Another Flip in the Row: Bypassing Rowhammer Defenses and Making Remote-Rowhammer Attacks Practical

Daniel Gruss, Moritz Lipp, Michael Schwarz

daniel.gruss@iaik.tugraz.at, moritz.lipp@iaik.tugraz.at, michael.schwarz@iaik.tugraz.at

Abstract

The Rowhammer bug is an issue in most DRAM modules [15, 23] which allows software to cause bit flips in DRAM cells, consequently manipulating data. Although only considered a reliability issue by DRAM vendors, research has showed that a single bit flip can subvert the security of an entire computer system.

In the introduction of the talk, we will outline the developments around Rowhammer since its presentation at Black Hat USA 2015. We discuss attacks [2, 3, 5, 8, 9, 11, 14, 15, 16, 17, 19, 20, 21, 22, 23, 24, 25, 27] and defenses that researchers came up with. The defenses against Rowhammer either try to prevent Rowhammer bit flips from occurring [1, 4, 7, 13, 15, 18], or at least ensure that Rowhammer attacks cannot exploit the bug anymore [6, 10, 12, 26].

We will present a novel Rowhammer attack [11] that undermines all existing assumptions on the requirements for such attacks. With one-location hammering, we show that Rowhammer does not necessarily require to alternating accesses to two or more addresses. We explain that modern CPUs rely on memory-controller policies that enables an attacker to use this new hammering technique. Moreover, we introduce new building blocks for exploiting Rowhammer-like bit flips which circumvent all currently proposed countermeasures. In addition to classical privilege escalation attacks, we also demonstrate a new, easily mountable denial-of-service attack which can be exploited in the cloud.

We will also show that despite all efforts, the Rowhammer bug is still not prevented. We conclude that more research is required to fully understand this bug to subsequently be able to design efficient and secure countermeasures.

1 Overview

In this whitepaper we cover the topics of our talk and also provide technical background. It consists of two parts.

The first part was published as a paper at the 39th IEEE Symposium on Security and Privacy 2018 with the title "Another Flip in the Wall of Rowhammer Defenses" [11]. The paper first systematically analyzes all proposed Rowhammer defenses and groups them into different classes. Then it shows how an attacker can bypass each defense class. Based on these insights, two attacks are presented. A native privilege escalation attack, and a denial-of-service attack in the cloud.

The second part is a pre-print of the paper "Nethammer: Inducing Rowhammer Faults through Network Requests" [17]. Rowhammer was assumed to be a local attack. However, as we find, remote attacks, attacks without a single line of attacker code on the system, are feasible. We performed such attacks and evaluated what we can do with the bit flips we obtained. While not as easy to exploit as bit flips in a local attack, bit flips do allow to perform persistent and non-persistent denial-of-service attacks. Furthermore, they can even allow attacks where data is maliciously modified by an attacker, enabling follow up attacks.

The main takeaways of both the talk and the whitepaper are as follows.

1. None of the previously known defenses solve the Rowhammer problem completely.
2. We need a better understanding of Rowhammer to find all attack variants and be able to design good defenses.
3. Even systems that never run any attacker-controlled code can be attacked, simply by flooding it with network packets.

References

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

[2] 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.

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

[4] 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.

[5] Y. Cheng, Z. Zhang, and S. Nepal, "Still hammerable and exploitable: on the effectiveness of software-only physical kernel isolation," *arXiv:1802.07060*, 2018.

[6] 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.

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

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

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

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

[11] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom, "Another Flip in the Wall of Rowhammer Defenses," in *S&P*, 2018.

[12] N. Herath and A. Fogh, "These are Not Your Grand Daddys CPU Performance Counters – CPU Hardware Performance Counters for Security," in *Black Hat Briefings*, Aug. 2015. [Online]. Available: https://www.blackhat.com/docs/us-15/materials/us-15-Herath-These-Are-Not-Your-Grand-Daddys-CPU-Performance-Counters-CPU-Hardware-Performance-Counters-For-Security.pdf

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

[14] Y. Jang, J. Lee, S. Lee, and T. Kim, "Sgx-bomb: Locking down the processor via rowhammer attack," in *SysTEX*, 2017.

[15] 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'14*, 2014.

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

[17] M. Lipp, M. T. Aga, M. Schwarz, D. Gruss, C. Maurice, L. Raab, and L. Lamster, "Nethammer: Inducing rowhammer faults through network requests," *arXiv:1711.08002*, 2017.

[18] M. Payer, "HexPADS: a platform to detect "stealth" attacks," in *ESSoS'16*, 2016.

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

[20] D. Poddebniak, J. Somorovsky, S. Schinzel, M. Lochter, and P. Rösler, "Attacking deterministic signature schemes using fault attacks," in *EuroS&P*, 2018.

[21] R. Qiao and M. Seaborn, "A new approach for Rowhammer attacks," in *International Symposium on Hardware Oriented Security and Trust*, 2016.

[22] 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.

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

[24] A. Tatar, R. Krishnan, E. Athanasopoulos, C. Giuffrida, H. Bos, and K. Razavi, “Throwhammer: Rowhammer Attacks over the Network and Defenses,” in *USENIX ATC*, 2018.

[25] 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’16*, 2016.

[26] V. van der Veen, M. Lindorfer, Y. Fratantonio, H. P. Pillai, G. Vigna, C. Kruegel, H. Bos, and K. Razavi, “Guardion: Practical mitigation of dma-based rowhammer attacks on arm,” in *DIMVA*, 2018.

[27] 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.

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 [44], 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 [65], from within a browser's sandbox [24], and from within virtual machines running on third-party compute clouds [70], mounting fault attacks on cryptographic primitives [10, 59], and obtaining root privileges on mobile phones [68]. 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 [44], ECC-RAM is intended for server computers and is typically not supported on consumer-grade machines.

Preprint of the work accepted at the 39th IEEE Symposium on Security and Privacy 2018 (https://www.ieee-security.org/TC/SP2018/index.html).

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 [44], removing unprivileged access to the `pagemap` interface [45, 62, 65], and prohibiting the `clflush` instruction [65]. However, recent works have already bypassed these countermeasures [6, 24, 68]. Other ad-hoc attempts, such as disabling page deduplication by default [52, 60], only prevent specific Rowhammer attacks exploiting these features [11, 59], 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 [28, 35].
- **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, 17, 25, 28, 35, 56, 75].
- **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, 18].
- **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 [24, 68].
- **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 [12].

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 [38, 57, 70], allowing us to perform Rowhammer attacks with even fewer requirements.

Page Placement Without Memory Exhaustion. Page deduplication is usually disabled for security reasons [52, 60, 68] as a response to page deduplication attacks [8, 22, 66], including deduplication-based Rowhammer attacks [11, 59], Hence, attacks can only use memory exhaustion [24, 65, 68, 70] to surgically place a target page on a vulnerable physical memory location. Consequently, countermeasures aim to prevent adversarial memory exhaustion [24, 68]. We introduce *memory waylaying*, a reliable technique exploiting the Operating System (OS) page cache to influence the physical location of a target page. 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. SGX is an x86 instruction-set extension to securely and confidentially run 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 OSs and hardware, the memory of the enclave is encrypted to prevent any modification or inspection of the enclave's memory contents, even by the OS's kernel and hardware components [19]. Furthermore, enclaves are excluded from the CPU performance counters [64]. Hence, this approach defeats countermeasures which rely on monitoring performance counters [6, 25, 28, 56] or on analyzing the application code or instruction stream for Rowhammer attacks [28, 35].

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

Section II provides background. Section III introduces a new categorization of Rowhammer defenses. Section IV defines our attacker model. Section V overviews our attack and its building blocks, which are detailed in Section VI (opcode flipping), Section VII (one-location hammering), and Section VIII (memory waylaying). Section IX evaluates our attacks in practical scenarios. Section X discusses limitations and additional observations. We conclude in Section XI.

II. Background

In this section, we overview the Rowhammer bug and defenses, discuss the prefetch side-channel attack which we use in Section VIII, and provide background on Intel SGX.

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 [54].

In 2014, Kim et al. [44] 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* [29]). To achieve these rapid DRAM accesses, data-caching mechanisms need to be bypassed, either by flushing the cache, e.g., using `clflush` [44], cache eviction [1, 6, 24], or uncached memory accesses [58]. 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.

Although the name single-sided hammering may suggest that only a single memory location is hammered, Seaborn and Dullien [65], who introduced this technique, hammer 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 [65]. NaCl executes arbitrary generated code directly on the CPU but sanitizes it using a blacklist, e.g., no 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 [10] exploited random Rowhammer bit flips in random memory locations to produce faulty RSA signatures, to 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 [24, 65, 70], grooming [68], or page deduplication [11, 59].

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. Static code analysis could be used to detect microarchitectural attacks in binaries in a fully automated way, e.g., when tested before loading them into an app store [35]. Several works detect ongoing attacks using hardware- and software-based performance counters [17, 18, 25, 28, 56, 75]. Herath and Fogh [28] detect attacks by monitoring suspicious cache activity of processes using performance counters and then searching for `clflush` instructions near the instruction pointer.

Rowhammer Neutralization Countermeasures. The system's memory allocator only places kernel pages near userspace pages in near-out-of-memory situations. Hence, modifying the allocator to prefer the out-of-memory situation over the proximate placement of kernel and userspace pages, effectively prevents memory exhaustion in turn of spraying and grooming [24, 68]. 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. [12] 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 [11, 52, 59, 60].

Rowhammer Elimination Countermeasures. ANVIL [6] uses performance counters to detect and subsequently mitigate Rowhammer attacks. More specifically, ANVIL uses the CPU's performance counters in order to continuously monitor the amount of cache misses. When the amount of cache misses exceeds a predetermined threshold, ANVIL's second stage is initiated, logging the addresses of cache misses. Finally, ANVIL mitigates Rowhammer effects by selectively refreshing nearby memory rows. However, as refreshing a row imposes some performance penalties, ANVIL avoids having a large number of false positives by discarding all logged cases that do have a significant amount of accesses to at least two rows in the same memory bank. While this optimization improves ANVIL's performance, as we discuss in Section III, it also prevents ANVIL from detecting one-location hammering, thus facilitating our attack. Similarly to ANVIL's detection approach, Corbet [18] discusses halting the CPU when cache-miss rates exceed a threshold, slowing down not only Rowhammer attacks but the entire system.

Brasser et al. [12] also presented B-CATT, a bootloader extension blacklisting vulnerable locations, thus, effectively reducing the amount of usable memory, but fully eliminating the Rowhammer bug. However, Kim et al. [44] observed that this approach is not practical as it would block almost the entire memory. We validated this observation and found more than $95\,\%$ of the memory would be blocked, on several of our systems. Eliminating Rowhammer by blacklisting the `clflush` instruction [65] was shown ineffective with cache-eviction-based Rowhammer attacks [1, 6, 24].

Besides building more reliable chips or employing ECC modules, Kim et al. [44] and Kim et al. [43] 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 [37] 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. [21] 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. [23] 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. Prefetch side-channel attacks also exploit the OS design. In most OSs, 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 OS 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 [19]. 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 OS, 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 [26]. 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., replacing a new encrypted page with an old encrypted page.

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 [26]. 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. [72] demonstrated a page fault side-channel attack from a malicious OS to extract sensitive information, e.g., text documents and images. Brasser et al. [13] demonstrated a Prime+Probe cache side-channel attack, extracting 70 % of an RSA private key in an enclave. Furthermore, Schwarz et al. [64] 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. [71] mounted control-flow inference attacks on recent SSL libraries running in secure enclaves. Moghimi et al. [53] 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. [69] 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 [73]) cannot be mounted, as enclave memory is inaccessible for other enclaves, processes, and the OS. But as DRAM rows are shared, Wang et al. [69] showed a cross-enclave DRAMA attack (cf. [57]) on other enclaves.

In a concurrent and independent work, Jang et al. [36] propose a denial-of-service attack running Rowhammer in an SGX enclave. We compare their and our observations in Section IX-A, where we describe a very similar attack.

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 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:

TABLE I: Rowhammer defenses for commodity systems.

Methodology \ Defense	MASCAT [35]	Chiappetta et al. [17]	Zhang et al. [75]	Herath and Fogh [28]	HexPADS [56]	Gruss et al. [25]	ANVIL [6]	Corbet [18]	No OOM [24, 68]	G-CATT [12]	B-CATT [12]	TRR [37]	MAC [37]	PARA/CRA/PRA [43, 44]	ARMOR [21]	ECC/Chipkill [30, 44]	Refresh Rate [44]
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 (○).

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 hardware-based defenses 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. [35] in MASCAT, an automated static code analysis tool to detect microarchitectural attacks on a large scale. Herath and Fogh [28] 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 [25, 28, 56]. The underlying Flush+Reload loop of Rowhammer is also detected by cache attack defenses [17, 75].

Memory Access Patterns Monitoring. The underlying assumptions of defenses based on memory access patterns (**D3**) are that Rowhammer attacks require a large number of cache misses on one row, and a large cumulative number of accesses on other rows in the same DRAM bank. Assuming this, Rowhammer attacks can be detected and stopped before they cause bit flips [6, 18]. ANVIL [6] detects Rowhammer in two stages: First, it monitors the last-level cache miss ratio. Next, if the cache miss ratio exceeds a threshold, ANVIL uses Intel PEBS to monitor the addresses of cache misses and distinguish Rowhammer attacks from legitimate work loads. For every candidate row, "other row access samples from the same DRAM bank" are checked (cf. Section 3.3 in [6]). Only if there are enough accesses to other rows of the same bank, an attack is detected and victim rows are refreshed [6].

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 [12] 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 [24, 65] and grooming [68] easily exhaust the entire memory in a way that gets the attacker process killed by the OS. 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 [24, 68]. 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 OS, 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 OS is up-to-date and fully patched, and no known software vulnerabilities exist that an attacker could exploit to elevate privileges.

Defenses. The system is protected with state-of-the-art Rowhammer defenses. Specifically, at least one defense from each defense class is deployed, including static analysis [35], hardware performance counters [6, 17, 25, 28, 56, 75], memory access pattern analysis [6], physical proximity prevention [12], and prevention of near-out-of-memory situations [24, 68].

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	●	●	●	●	●

Attacker Capabilities. We assume that an attacker can start an arbitrary unprivileged user program and 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 in enclaves cannot be read or inspected, as the processor prevents all accesses to the enclave memory. By encrypting the code and only decrypting it after the enclave is launched, a developer can hide arbitrary code within SGX enclaves. Consequently, MASCAT [35] is incapable of detecting any microarchitectural or Rowhammer attack we perform inside the enclave. Furthermore, the instruction stream cannot be searched for `clflush` instructions [28].

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 [31]. Confirming this, Schwarz et al. [64] observed that performance counters are not influenced by cache attacks running in SGX enclaves. Hence, performance counters do not detect our attack.

One-location Hammering. To defeat defense class **D3** (memory access patterns), we introduce a new attack primitive, which we call *one-location hammering*. As older systems used an "open-page" memory controller policy where a memory row is kept open and buffered until the next memory row is accessed, double-sided and single-sided hammering cause frequent activations of rows by inducing cache misses on different rows of the same bank [44]. Recently, however, modern systems employ more sophisticated memory controller policies, preemptively closing rows earlier than necessary, to optimize performance (cf. Appendix C). We conjecture that this change in policy creates a previously unknown Rowhammer effect, which we exploit with one-location hammering.

With one-location hammering, the attacker only runs a Flush+Reload loop on a single memory address at the maximum frequency. This continuously re-opens the same DRAM row, whenever the memory controller closes the row. We observed that one-location hammering drains enough charge from the DRAM cells to induce bit flips. As one-location hammering does not access different rows in the same bank, **D3** defenses, such as the second stage of ANVIL [6], do not detect the ongoing attack (cf. Section III). 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 [24, 65, 68, 70].

With opcode flipping, we propose a novel way to exploit bit flips. In the x86 instruction set, bit flips in opcodes yield other, 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 [12].

Previous attacks on unprivileged code [65] (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 detail opcode flipping 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.

We observed that page cache pages, after being discarded from DRAM, are loaded to a new random physical location upon access, on both Linux and Windows. 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 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) induced bit flips in the page number field of an attacker-generated page table, in order to change the memory page reference by some page table entry. Seaborn and Dullien [65] (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 previous 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. In order to find suitable bit flips in system binaries, we used the following methodology. First, we manually define ranges within in the binary for which bits could be flipped. We then automatically test every single bit flip in these ranges, grouping the modified binaries by the result of their corresponding execution. Finally we manually analyze the results, looking for devastating outcomes (such a obtaining root permissions without knowing the root password) and target these bits via our Rowhammer attack.

Opcode flipping exploits that bit flips in opcodes can yield other, 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 and discuss the effect of such bit flips, before we show in Section VIII how a file can be 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`. Only 21 out of 255 two-byte sequences starting with the prefix `0x64` are illegal 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 analyzed one of the most prominent targets for privilege escalation, the `sudo` binary and `sudoers.so` shared library (henceforth *`sudo` binary*).

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 induce 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.

One-location hammering truly hammers only one memory location, i.e., the attacker does not directly induce row conflicts but only re-opens one row permanently. The core of one-location hammering is a Flush+Reload loop hammering a single randomly chosen address, voiding the assumptions of defense class **D3**. Both, one-location hammering and single-sided hammering are oblivious to virtual-to-physical address mappings. Hence, we can also apply both hammering techniques if 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 a Skylake i7-6700K with two 8 GB Crucial DDR4-2133 DIMMs. We tested each technique for eight hours and scanned for bit flips after each hammering attempt (i.e., after 5 000 000 rounds of Flush+Reload on two or one address, respectively). Each hammering attempt hammers random memory locations (randomly-chosen offsets on more than 100 000 randomly-chosen 4 kB pages).

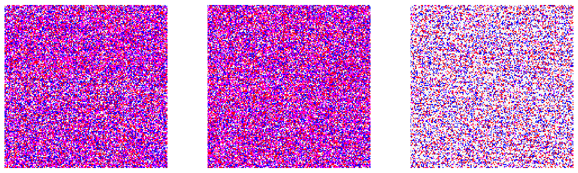


Fig. 1: Flippable bit offsets over $4\,\mathrm{kB}$-aligned memory regions for different hammering techniques. Bit flips from 0 to 1 (blue) and bit flips from 1 to 0 (red) may occur at any bit offset.

Figure 1 shows the distribution of bit flip offsets accumulated over $4\,\mathrm{kB}$-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 $4\,\mathrm{kB}$-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 $4\,\mathrm{kB}$-offset. $51.6\,\%$ of the bit flips were from 0 to 1. This is worse than double-sided hammering and single-sided hammering. 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 a Haswell i7-4790 with two Kingston DDR3-1600 DIMMs. We observe bit flips for all hammering techniques, including one-location hammering. On an Ivy Bridge i5-3230M with two Samsung DDR3-1600 SO-DIMMs we observe a significantly higher number of bit flips for double-sided hammering than for single-sided hammering, while bit flips from one-location hammering were rare and not reliably reproducible. Our measurements indicate that this machine uses an open-page memory controller policy, as opposed to the more efficient policies used on the other two systems (cf. Appendix C). However, 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 $4\,\mathrm{kB}$ 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 [74]. However, the page cache keeps every binary page only once in memory. Linux 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 [23] can be leveraged as an oracle. In Section VIII-B, we present a technique to evict a target page from the page cache, forcing relocation at the next access. In Section VIII-C, we describe how the prefetch attack 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 [23] 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 [73], 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. [64] demonstrated that accurate timing can be obtained by using counting threads and Wang et al. [69] 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 offsets in the direct-physical map with 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. [23], 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*

Both on Windows and Linux, 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 [24, 65, 68]. 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.

A fundamental observation we made is that the replacement algorithm of the Linux 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. On Windows, executable and non-executable file-backed pages can be used equally. 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 [24, 65, 68].

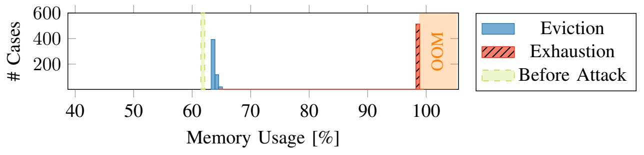


Fig. 2: Our replacement-aware page cache eviction only leads to negligible memory increase, whereas existing techniques are close to an out-of-memory situation.

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 OS 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 2 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 OS 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 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. On Windows, the page cache eviction takes on average 10.10 seconds, as we cannot rely on the Linux `mincore` function to abort the eviction process.

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.

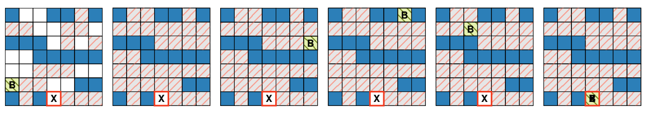

(a) Start (b) Our Eviction (c) Access Binary (d) Repeat: Evict + Access (e) Repeat: Evict + Access (f) Stop if target reached

Fig. 3: Memory waylaying. In step (a) some pages are free (☐). Our eviction (b) allocates all free pages for the page cache (▨), but leaves occupied pages (🟦) untouched. Repeating the eviction, the target page B (B) is relocated, but the occupied memory remains the same. Eventually, B is placed on the target physical location X (x) as illustrated (f).

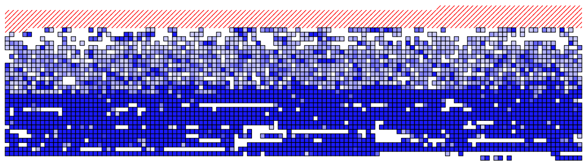


Fig. 4: Distribution of placements of a page in the physical memory of our test system (12 GB). Each square represents 4 MB. Hatched (red) areas are unavailable to the system (e.g., graphics memory). The darker (blue) an area, the more physical pages were in this area. Even a small number of relocations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

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 works the same way on Windows and Linux, as illustrated in Figure 3.

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 4 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) covered most of the physical memory, with the exception of occupied memory regions. Thus, eventually the target binary page is placed at a physical memory location where 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 OS 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 µs, whereas memory waylaying requires 2.68 s. On Windows, we could not test memory chasing as there is no equivalent to the `fork` function. With 10.10 s, memory waylaying requires slightly more time on Windows. 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 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 [57]. In the cloud scenario, the attacker uses our attack to identify vulnerable servers and take these 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. Note that only a tiny fraction of $4\,\text{kB}$ pages are adjacent to the $128\,\text{MB}$ EPC memory area. For instance, on a system with $16\,\text{GB}$ 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 [27] or Xen [7] 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 [32, 33, 34]. Recently, Microsoft [51] 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 the position of bit flips is uniformly distributed, an attacker learns from any bit flip while templating, that the DRAM very likely 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 bit flips in EPC memory. The runtime of this phase is in the range of seconds to minutes.

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* [51] 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 [51]. 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.

In a concurrent independent work, Jang et al. [36] propose a similar attack, making the same observations as we did: the system reset does not work properly following bit flips in SGX; any bit flip in the $128\,\text{MB}$ region causes the system to halt, making the attack easier than other Rowhammer attacks; all detection mechanisms are bypassed by hiding the Rowhammer code inside an enclave; and that just locking down the processor in case of a bit flip might not be the best defense scheme. As a defense, they propose that future work should investigate whether there are non-process-specific performance counters which allow detection of suspicious activity in SGX enclaves.

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**.

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

In our example attack, we apply opcode flipping (cf. 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 completed, the attacker continues with the main attack, consisting of four phases:

- **Templating phase.** Our Rowhammer enclave templates 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.
  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 OS 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 [15]. It is very well possible to slip malware into app stores [15]. 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, 49, 63].

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 OS. The OS 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 [47], operations on large encrypted database files [14, 42, 55, 63]. Hence, it is unclear whether memory allocation patterns alone are enough to give away a Rowhammer attack. There is no further interaction between the enclave and the non-enclave sides that could be monitored to detect the attack. Finally, future software defenses may still prevent our attack, e.g., by checking the integrity of binaries and terminating processes when an integrity check fails.

SGX enclaves should only be run if they are signed by Intel or a trusted partner. If Intel or one of the trusted partners do not thoroughly review the code before singing it, our attack might slip through the signing process. However, as this enclave signing process has not yet been deployed, it is unclear whether such a code review would actually happen. Perhaps more devastating is that fact that users and businesses can deliberately run non-signed enclaves. In fact, Microsoft already does this on the *Azure confidential computing* cloud [51]. Hence, it is unclear whether a signing process would pose any limitation for our attack.

Currently, in our opcode flipping technique, the identification of target bit flip locations in binaries requires some manual work. That is, manually defining a range where bit flips should be tested and manually selecting the groups of successful execution results. While this is certainly feasible for a small number of binaries, fully automating this process would allow a complete analysis of the attack surface. Similarly, compilers could generate code which guarantees that an attacker requires at least N bit flips to successfully manipulate the control flow, i.e., N is a security parameter (cf. [9, 16]). We consider this an interesting direction of future work, not only for research on Rowhammer attacks but also on fault attacks in general.

B. Rowhammer mitigations in hardware

While it might be possible to design a practical software-based Rowhammer countermeasure, the results of our paper indicate that this is difficult, since not all variants of triggering the Rowhammer bug are known. Furthermore, future Rowhammer defenses should also be designed with related fault attacks in mind [40, 46]. We now discuss proposed and existing countermeasures implemented that require hardware modifications, but tackle the problem at its root.

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 [30] allows to successfully recover from 3-bit errors. However, uncorrectable multi-bit flips can be exploitable [2, 3, 48] or can result in a denial-of-service attack similar as described in Section IX-A depending on how the OS 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 [37] implements TRR and MAC, van der Veen [67] still reported bit flips on a Google Pixel phone with 4 GB LPDDR4 memory. Doubling the refresh rate has been shown to be insufficient [6, 44] and a further increase would incur a too high performance penalty [44].

Meaney et al. [50] 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. For an uncorrectable error, an attacker would have to induce multiple bit flips in different rows of different modules, making Rowhammer attacks infeasible.

Kim et al. [44] and Kim et al. [43] proposed to eliminate bit flips in hardware by probabilistically opening adjacent or non-adjacent rows, whenever a row is opened or closed. As ongoing Rowhammer attacks open and close a certain row repeatedly, the vulnerable adjacent rows would be refreshed before bit flips occur. Their approaches are possible solutions to mitigate Rowhammer attacks in future hardware.

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. Taking down entire clouds, possibly in a coordinated and distributed way, 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. A similar design change was also proposed by Jang et al. [36].

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 OS into relocating any binary page takes 2.68 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 our anonymous reviewers and Mark Seaborn for their valuable feedback as well as Thomas Schuster for help with some experiments. This work has been supported by the Austrian Research Promotion Agency (FFG) via the K-project DeSSnet, which is funded in the context of COMET – Competence Centers for Excellent Technologies by BMVIT, BMWFW, Styria and Carinthia. This project has received funding from the European Union's Horizon 2020 research and innovation programme under European Research Council (ERC) grant agreement No 681402 and under grant agreement No 644052 (HECTOR). Yuval Yarom performed part of this work as a visiting scholar at the University of Pennsylvania, supported by an Endeavour Research Fellowship from the Australian Department of Education and Training. Daniel Genkin was supported by NSF awards #1514261 and #1652259, financial assistance award 70NANB15H328 from the U.S. Department of Commerce, National Institute of Standards and Technology, the 2017-2018 Rothschild Postdoctoral Fellowship, and the Defense Advanced Research Project Agency (DARPA) under Contract #FA8650-16-C-7622.

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 *International Symposium on Hardware Oriented Security and Trust*, 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] T. Barry, D. Couroussé, and B. Robisson, "Compilation of a countermeasure against instruction-skip fault attacks," in *Workshop on Cryptography and Security in Computing Systems*, 2016.

[10] S. Bhattacharya and D. Mukhopadhyay, "Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis," in *CHES*, 2016.

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

[12] 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.

[13] 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.

[14] H. Brekalo, R. Strackx, and F. Piessens, "Mitigating password database breaches with Intel SGX," in *Workshop on System Software for Trusted Execution*, 2016.

[15] 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.

[16] Z. Chen, J. Shen, A. Nicolau, A. Veidenbaum, N. F. Ghalaty, and R. Cammarota, "CAMFAS: A compiler approach to mitigate fault attacks via enhanced SIMDization," in *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2017.

[17] 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.

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

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

[20] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu, "Memory power management via dynamic voltage/frequency scaling," in *ACM International Conference on Autonomic Computing*, 2011.

[21] 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

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

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

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

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

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

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

[28] 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.

[29] 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 *Annual Design Automation Conference (DAC)*, 2012.

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

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

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

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

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

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

[36] Y. Jang, J. Lee, S. Lee, and T. Kim, "SGX-Bomb: Locking down the processor via Rowhammer attack," in *SysTEX*, 2017.

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

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

[39] O. D. Kahn and J. R. Wilcox, "Method for dynamically adjusting a memory page closing policy," Sep. 28 2004, uS Patent 6,799,241.

[40] 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.

[41] D. Kaseridis, J. Stuecheli, and L. K. John, "Minimalist open-page: A DRAM page-mode scheduling policy for the many-core era," in *International Symposium on Microarchitecture (MICRO)*, 2011.

[42] 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.

[43] 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.

[44] 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.

[45] 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

[46] 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.

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

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

[49] 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.

[50] 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.

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

[52] ——, "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

[53] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "CacheZoom: How SGX amplifies the power of cache attacks," in *CHES 2017*, 2017, pp. 69–90.

[54] 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.

[55] 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.

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

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

[58] R. Qiao and M. Seaborn, "A new approach for Rowhammer attacks," in *International Symposium on Hardware Oriented Security and Trust*, 2016.

[59] 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.

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

[61] H. G. Rotithor, R. B. Osborne, and N. Aboulenein, "Method and apparatus for out of order memory scheduling," Oct. 24 2006, uS Patent 7,127,574.

[62] 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

[63] 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," in *S&P*, 2015.

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

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

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

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

[68] 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.

[69] 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.

[70] 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.

[71] 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.

[72] Y. Xu, W. Cui, and M. Peinado, "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems," in *S&P*, May 2015.

[73] Y. Yarom and K. Falkner, "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *USENIX Security Symposium*, 2014.

[74] K. S. Yim, "The rowhammer attack injection methodology," in *IEEE 35th Symposium on Reliable Distributed Systems (SRDS)*, 2016.

[75] 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 $4\,\text{kB}$ 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 $4\,\text{kB}$ page (2^{12} bytes) which can flip in both directions each.

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

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

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

Figure 5 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.

C. Memory Basics, Policies, and their Influence on One-Location Hammering

DRAM is organized in multiple banks, e.g., for a dual-channel dual-rank configuration 32 banks on DDR3 and 64 banks on DDR4. Each bank consists of an array of rows of $8\,\text{kB}$ each. Thus, the number of rows is typically in the range of 2^{14} to 2^{16}. Since the DRAM cells lose their charge over time, the DDR standard defines that every row must be refreshed once per $64\,\mu\text{s}$. When accessing a memory location,

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`

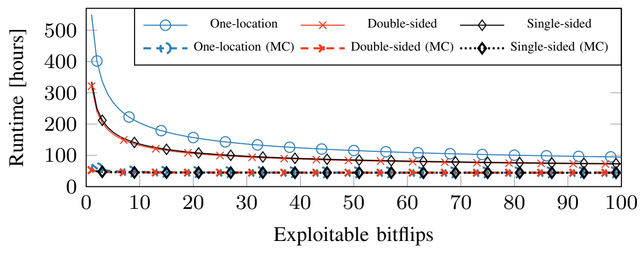


Fig. 5: Expected total runtime (templating and waylaying) until the attacker has the target page at the target physical location.

the corresponding row is opened, i.e., copied into an internal array called the *row buffer*. Closing a row copies the data from the row buffer back into the actual DRAM cells.

Before a row can be opened, the bank has to be precharged. Consequently, when accessing a memory location in the currently opened row, i.e., a row hit, the latency is comparably low. Accessing a memory location in a different row, i.e., a row conflict, incurs first closing the DRAM row, then precharging the bank, and finally opening the new row, copying the data into the row buffer. The latency in this case is significantly higher, e.g., 200 % of the latency of a row hit.
recently accessed row open and buffered. This is beneficial

The memory controller can optimize the memory performance by cleverly deciding when to close a row preemptively. The two most basic memory controller policies are “open page” and “closed page”. An open-page policy keeps the for memory access latency, power consumption, and bank utilization when the number of memory accesses is low [41]. However, when the number of memory accesses increases the situation is more complex. A closed-page policy can then achieve a better system performance, since the row is immediately closed and the bank is precharged and ready to open a new row [41].

With modern processors having huge caches and complex algorithms for spatial and temporal prefetching, the probability that further memory accesses go to the same row decreases. Consequently, more complex memory controller policies have been proposed and are implemented in modern processors [41]. David et al. [20] noted that closed-page policies perform especially better on multi-core systems and hence they assumed that these are implemented in current processor architectures. Intel also holds patents for dynamically adjusting memory controller policies [39]. A closed-page policy, but also other policies which preemptively close rows, would allow one-location hammering.

Besides these memory controller policies, the memory controller can also reorder and combine memory accesses [61]. Since the Rowhammer bug is related to the number of row activations [44], a lower number of activations due to reordering and combining also reduces the probability of bit flips. In one-location hammering most of the accesses can be expected to be reordered and combined to reduce the overall number of row activations, leading to a lower number of bit flips than with other hammering techniques.

Nethammer: Inducing Rowhammer Faults through Network Requests

Moritz Lipp
Graz University of Technology

Misiker Tadesse Aga
University of Michigan

Michael Schwarz
Graz University of Technology

Daniel Gruss
Graz University of Technology

Clémentine Maurice
Univ Rennes, CNRS, IRISA

Lukas Raab
Graz University of Technology

Lukas Lamster
Graz University of Technology

ABSTRACT

A fundamental assumption in software security is that memory contents do not change unless there is a legitimate deliberate modification. Classical fault attacks show that this assumption does not hold if the attacker has physical access. Rowhammer attacks showed that local code execution is already sufficient to break this assumption. Rowhammer exploits parasitic effects in DRAM to modify the content of a memory cell without accessing it. Instead, other memory locations are accessed at a high frequency. All Rowhammer attacks so far were local attacks, running either in a scripted language or native code.

In this paper, we present Nethammer. Nethammer is the first truly remote Rowhammer attack, without a single attacker-controlled line of code on the targeted system. Systems that use uncached memory or flush instructions while handling network requests, e.g., for interaction with the network device, can be attacked using Nethammer. Other systems can still be attacked if they are protected with quality-of-service techniques like Intel CAT. We demonstrate that the frequency of the cache misses is in all three cases high enough to induce bit flips. We evaluated different bit flip scenarios. Depending on the location, the bit flip compromises either the security and integrity of the system and the data of its users, or it can leave persistent damage on the system, *i.e.,* persistent denial of service.

We investigated Nethammer on personal computers, servers, and mobile phones. Nethammer is a security landslide, making the formerly local attack a remote attack. With this work we invalidate all defenses and mitigation strategies against Rowhammer build upon the assumption of a local attacker. Consequently, this paradigm shift impacts the security of millions of devices where the attacker is not able to execute attacker-controlled code. Nethammer requires threat models to be re-evaluated for most network-connected systems. We discuss state-of-the-art countermeasures and show that most of them have no effect on our attack, including the target-row-refresh (TRR) countermeasure of modern hardware.

Disclaimer: This work on Rowhammer attacks over the network was conducted independently and unaware of other research groups working on truly remote Rowhammer attacks. Experiments and observations presented in this paper, predate the publication of the Throwhammer attack by Tatar et al. [81]. We will thoroughly study the differences between both papers and compare the advantages and disadvantages in a future version of this paper.

1 INTRODUCTION

Hardware-fault attacks have been considered a security threat since at least 1997 [12, 13]. In such attacks, the attacker intentionally brings devices into physical conditions which are outside their specification for a short time. For instance, this can be achieved by temporarily using incorrect supply voltages, exposing them to high or low temperature, exposing them to radiation, or by dismantling the chip and shooting at it with lasers. Fault attacks typically require physical access to the device. However, if software can bring the device to the border or outside of the specified operational conditions, software-induced hardware faults are possible [50, 80].

The most prominent hardware fault which can be induced by software is the Rowhammer bug, caused by a hardware reliability issue of DRAM. An attacker can exploit this bug by repeatedly accessing (*hammering*) DRAM cells at a high frequency, causing unauthorized changes in physically adjacent memory locations. Since its initial discovery as a security issue [50], Rowhammer's ability to defy abstraction barriers between different security domains has been improved gradually to develop more powerful attacks on various systems. Examples of previous attacks include privilege escalation, from native environments [27, 78], from within a browser's sandbox [14, 25, 28], and from within virtual machines running on third-party compute clouds [86], mounting fault attacks on cryptographic primitives [11, 73], and obtaining root privileges on mobile phones [84].

Most Rowhammer attacks assume that two DRAM rows must be hammered to induce bit flips. The reason is that they assume that an "open-page" memory controller policy is used, *i.e.,* a DRAM row is kept open until a different row is accessed. However, modern CPUs employ more sophisticated memory controller policies that preemptively close rows [27]. Based on this observation, Gruss et al. [27] described a technique called *one-location* hammering.

In 2016, Intel introduced Cache Allocation Technology (CAT) to address quality of service in multi-core server platforms [32]. Intel CAT allows restricting cache allocation of cores to a subset of cache ways of the last-level cache, with the aim of optimizing workloads in shared environments, e.g., protecting virtual machines against performance degradation due to cache thrashing of a co-located virtual machine. However, with a lower number of cache ways available to the process, the probability to evict an address by accessing other addresses increases significantly. Aga et al. [4] showed that this facilitates eviction-based Rowhammer attacks.

All previously known Rowhammer attacks required some form of local code execution, e.g., JavaScript [14, 25, 28] or native code [4, 9, 11, 27, 50, 62, 69, 73, 78, 84, 86]. Moreover, all works on Rowhammer defenses assume that some form of local code execution is required [9, 14, 15, 17, 18, 26, 29, 31, 43, 49, 50, 59, 67, 74, 91]. In particular, we found that none of these works even mentions the theoretical possibility of truly non-local Rowhammer attacks. Consequently, it was a widely accepted assumption that remote Rowhammer attacks are not possible. More specifically, devices where an attacker could not obtain local code execution were so far considered to be safe. Yet, the following questions arise:

Are remote Rowhammer attacks possible? More specifically, is it possible for an attacker to induce bit flips and exploit them, without any local code execution on the system?

In this paper, we answer these questions and confirm that truly remote Rowhammer attacks are possible. We present Nethammer, the first Rowhammer attack that does not require local code execution. Nethammer requires only a fast network connection between the attacker and victim. It sends a crafted stream of size-optimized packets to the victim which causes a high number of memory accesses to the same set of memory locations. If the network driver or other parts of the network stack use uncached memory or flush instructions, e.g., for interaction with the network device, an attacker can induce bit flips. Furthermore, if Intel CAT is activated, e.g., as an anti-DoS mechanism, memory accesses lead to fast cache eviction and thus frequent DRAM accesses. This enables attacks even if there are no accesses to uncached memory or flush instructions while handling the network packet. Thus, the attacker implicitly hammers the DRAM through the code executed for processing the network packets. While an attacker cannot control the addresses of the bit flips, we demonstrate how an attacker can still exploit them.

Nethammer has several building blocks that we systematically developed. First, we measure whether handling network packets could at least, in theory, induce bit flips, and the influence of real-world memory-controller page policies. For this purpose, we present a new algorithm to observe and classify the memory-controller page policy. Second, based on these insights, we demonstrate that one-location hammering [27] does not require a closed-page policy, but instead, adaptive policies may also allow one-location hammering. Third, we investigate memory operations that occur while handling network requests. Fourth, we show that the time windows we observe between memory accesses from subsequent network requests enable Rowhammer attacks.

As previous work on Rowhammer showed, once a bit flips in a system, its security can be subverted. We present different attacks exploiting bit flips on victim machines to compromise various services, in particular, version-control systems, DNS servers and OCSP servers. In all cases, the triggered bit flips may induce persistent denial-of-service attacks by corrupting the persistent state, e.g., the file system on the remote machine. In our experiments, we observed bit flips using Nethammer already after 300 ms of running the attack and up to 10 000 bit flips per hour. Nethammer represents a significant paradigm shift, from local to remote attacks. Previous fault attacks required physical access or local code execution in the case of Rowhammer. Making Rowhammer possible over the network requires re-evaluating the threat model of virtually every network-connected system. We discuss state-of-the-art countermeasures and show that most of them do not affect our attack, including the target-row-refresh (TRR) countermeasure in hardware. Furthermore, we evaluate the performance of different other proposed Rowhammer countermeasures against Nethammer. Nethammer is difficult to detect on systems where high network traffic is commonplace. Finally, we discuss how attacks like Nethammer can be mitigated.

Contributions. The contributions of this work are:

- We present Nethammer, the first truly remote Rowhammer attack, with not even a single line of attacker-controlled code running on the target device.
- We demonstrate Nethammer on devices that either use uncached memory or `clflush` while handling network packets.
- We demonstrate that even without uncached memory and `clflush`, attacks on cloud systems can still be practical.
- We illustrate how our attack invalidates assumptions from previous works, marking a paradigm shift, and requiring re-evaluation of the threat models of most network-connected systems.
- We show that many previously proposed defenses, e.g., TRR, do not work against our new attack.

Outline. The remainder of the paper is structured as follows. In Section 2, we provide background information. In Section 3, we overview the Nethammer attack. In Section 4, we describe the building blocks and obtain insights we need for Nethammer. In Section 5, we demonstrate how bit flips induced over the network can be exploited. In Section 6, we evaluate the performance of Nethammer in different scenarios on several different systems. In Section 7, we discuss and propose countermeasures. In Section 8, we discuss limitations of Nethammer. We conclude our work in Section 9.

2 BACKGROUND

In this section, we provide the necessary background information on DRAM, memory controller policies, and the Rowhammer attack. Furthermore, we discuss caches and cache eviction as well as the Intel CAT technology.

2.1 DRAM and Memory Controller Policies

DRAM Organization. Modern computers use DRAM as the main memory. To maximize data transfer rates, DRAM is organized for a high degree of parallelism, in a hierarchy of channels, DIMMs, ranks, bank groups, and banks. Most processors today support dual-channel or quad-channel configurations. The DIMMs are assigned to one of the channels. Each DIMM has one or more ranks, e.g., the two sides of the DIMM may form two ranks. Every rank is further subdivided into so-called *banks*, with each bank spanning over multiple chips. The number of banks in a rank is standardized [45], e.g., 8 banks on DDR3 and 16 banks on DDR4. Each bank is an array of *cells*, organized in *rows* and *columns*, storing the actual memory content. The row size, *i.e.*, the amount of data that can be stored in all cells of one row, is defined to be 8 kB [45]. Each cell is made out of a capacitor and an access transistor. The charge of the capacitor represents the binary data value of the cell. Each cell in the grid is connected to the neighboring cells with a wire forming horizontal and vertical bit lines.

When accessing a physical address, the memory controller translates the physical address to channel, DIMM, rank, bank group, bank, row, and column addresses. While AMD publicly documents these addressing functions [3], Intel and ARM do not. Pessl et al. [68] reverse-engineered these addressing functions using an automated technique for several Intel and ARM processors.

As DRAM cells lose their charge over time, they must be refreshed periodically. The maximum time interval between refreshes is defined through the *row refresh rate*, standardized by the JEDEC group for the different DRAM technologies [45]. Typically, the refresh interval is 64 ms but can vary depending on the device, on-the-fly adjustments due to the current temperature, or other external influences. With a 64 ms refresh interval, the memory controller issues the refresh command every 7.8 µs for each bank.

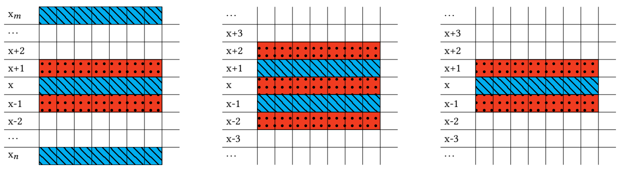
Memory Controller Policies. Each bank has a *row buffer*, acting as a directly-mapped cache for the rows. To read data, the data is moved from the cells of a row to the row buffer before it is sent to the processor. Similarly, write accesses go to the row buffer instead of directly to the row. By raising the word line of a row, all access transistors in that row are activated to connect all capacitors to their respective bit line. This transfers the charge representing the data from the row to the row buffer. If the requested data from this bank is already stored in the row buffer, the data can be transmitted to the processor immediately, resulting in a fast access time (a *row hit*). However, if the requested data is not in the row buffer, a so-called *row conflict* occurs, and the bit lines must be *pre-charged* before the data can be read from the new target row (row-activate).

Consequently, there are three different cases leading to distinct access times: row hits are the fastest, an access to a row in a pre-charged bank is a few nanoseconds slower, row conflicts are significantly slower (*i.e.*, several nanoseconds). Hence, the memory controller can optimize the memory performance by deciding when to close a row preemptively and pre-charge the bank. Typically, memory controllers employ one of the three following page policies:

(1) **Closed-page policy**: the page is immediately closed after every read or write request, and the bank is pre-charged and, thus, ready to open a new row (page-empty). If subsequent accesses are likely to be from other rows, a closed-page policy can achieve a better average system performance.

(2) **Fixed open-page policy**: the page is left open for a fixed amount of time after a read or write request. If temporal locality is given, subsequent accesses are served with a low latency. This policy is also beneficial for power consumption and bank utilization [48].

(3) **Adaptive open-page policy**: the adaptive open-page policy by Intel [21] is similar to the fixed open-page policy but dynamically adjusts the page timeout interval. Each row buffer has a timeout counter and a timeout register. A row remains open until the timeout counter reaches the value of the timeout register. As the initial timeout register value might not be the most efficient, an additional mistake counter is introduced to update the timeout register dynamically [26]. If a row conflict occurs, the memory controller kept the row open for too long and hence, the mistake counter is decremented. Whenever a page-empty access could have been a hit as the requested row is the same as the last accessed one, the mistake counter is incremented. Periodically, the value of the mistake counter is checked to decide if a less or more aggressive close-page policy should be used. If the mistake counter is higher than a certain threshold, the timeout register is incremented to keep the row open for a longer period of time, and conversely, if the mistake counter is lower than a certain threshold, the timeout register is decremented to close the row earlier.



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

Figure 1: Different hammering strategies: blue rectangles (▨) represent the hammered location, while red rectangles (🟥) represent the most likely location for bit flips to occur.

As modern processors have many cores running independently as well as deploy large caches and complex algorithms for spatial and temporal prefetching, the probability that subsequent memory accesses go to the same row decreases. Awasthi et al. [8] proposed an access-based page policy that assumes a row receives the same number of accesses as the last time it was activated. Shen et al. [79] proposed a policy taking past memory accesses into account to decide whether to close a row preemptively. Intel suggested predicting how long a row should be kept open [47, 82]. Consequently, more complex memory controller policies have been proposed and are implemented in modern processors [26, 48]. Besides these memory controller policies, the memory controller can also reorder and combine memory accesses [76].

2.2 Rowhammer

With increasing DRAM cell density, the physical size of DRAM cells and their capacitance decreases. While this has the advantage of higher storage capacity and lower power consumption, cells may be more susceptible to disturbance errors. Disturbance errors are interferences between cells that cause memory corruption by unintentionally flipping the bit-value of a DRAM cell [62].

In 2014, Kim et al. [50] demonstrated that such bit flips could be reliably triggered in a DRAM row by accessing memory locations in adjacent DRAM rows in a high frequency, a technique known as *row hammering* [35]. Typically, subsequent memory accesses would be served from the CPU cache. However, in a Rowhammer attack, the cache is bypassed by either using specific instructions [50], cache eviction [4, 9, 25, 28] or uncached memory [69, 84].

To reliably induce bit flips, different techniques have been proposed using different memory access patterns as illustrated in Figure 1. While the name *single-sided hammering* suggests that only one memory location is accessed, Seaborn and Dullien [78] accessed 8 randomly chosen memory locations simultaneously. Seaborn and Dullien [78] focused on a typical DDR3 setup with 32 DRAM banks. Following the birthday paradox, the probability is quite high that at least 2 out of 8 random memory locations map into the same DRAM bank. By repeatedly accessing these 8 memory locations, the attacker induces row conflicts at a high frequency. With single-sided hammering, bit flips most likely occur in some proximity to one of the 8 hammered rows.

With *double-sided hammering*, the attacker chooses three rows, where the two outer rows are hammered. Bit flips most likely occur in the row between the two rows. Double-sided hammering requires at least partial knowledge of virtual-to-physical mappings.

Finally, Gruss et al. [27] proposed *one-location hammering*, in which the attacker only accesses one single location at a high frequency. The attacker does not directly induce row conflicts but instead keeps re-opening one row permanently. As modern processors do not use strict open-page policies anymore, the memory controller preemptively closes rows earlier than necessary, causing row conflicts on the subsequent accesses of the attacker. Bit flips most likely occur in proximity to the hammered row.

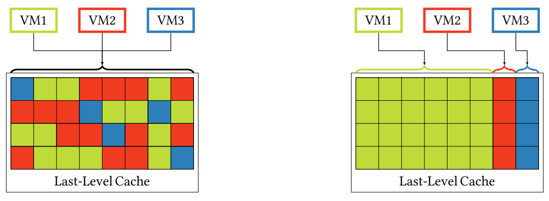
Using these techniques, the Rowhammer bug has been exploited in different scenarios. Bhattacharya and Mukhopadhyay [11] exploited untargeted bit flips at random locations to produce faulty RSA signatures, allowing the recovery of the secret keys. However, as bit flips can be reproduced quite reliably, more deterministic attacks have been mounted. These attacks include privilege-escalation attacks, sandbox escapes and the compromise of cryptographic algorithms. They have been mounted from sandboxed environments [78], from native environments [27, 78], from virtual machines in the cloud [73, 86], as well as from within a web browser running JavaScript [14, 28]. Furthermore, attacks from native code [84] and JavaScript within the browser sandbox [25] have been demonstrated on mobile devices. To reliably induce a bit flip on a specific page, memory spraying [28, 78, 86], grooming [84], and page deduplication [14, 73] have been used.

To develop countermeasures, a large body of research focused on detecting [17, 18, 29, 31, 43, 67, 91], neutralizing [14, 15, 28, 73, 84], or eliminating [9, 15, 18, 26, 49, 50] Rowhammer attacks in software or hardware. Furthermore, the LPDDR4 standard [46] specifies two features to mitigate Rowhammer attacks: with Target Row Refresh (TRR) the memory controller refreshes adjacent rows of a certain row and with Maximum Activation Count (MAC) the number of times a row can be activated before adjacent rows have to be refreshed is specified. However, in 2018, Gruss et al. [27] showed that an attacker can bypass all software-based countermeasures and gain root privileges by mounting a one-location hammering Rowhammer attack from inside an Intel SGX enclave.

2.3 Caches and Cache Eviction

Caching is a fundamental concept that is used to reduce the latency of various operations, in particular computations and accesses to slower storage. Hardware caches keep frequently used data from main memory in smaller but faster memories.

Cache Organization. Modern CPUs have multiple levels of caches, varying in size and latency, where the level-1 (L1) cache is the smallest and fastest, and the L3 or last-level cache is the biggest but slowest cache. Modern caches are organized in cache sets consisting of a fixed number of cache ways. The cache set is determined by either the virtual or physical address. Addresses are called *congruent* if they map to the same cache set. The cache replacement policy decides which of the cache ways is replaced (evicted) when new data has to be loaded into the cache.



(a) CAT disabled **(b) CAT enabled**

Figure 2: When Intel CAT is disabled in (a), the cache is shared among the virtual machines. In (b), Intel CAT is configured with 6 ways for VM1, and 1 way for VM2 and VM3.

On most Intel CPUs, the last-level cache is inclusive, *i.e.*, data present in L1 or L2 cache must also be present in the last-level cache. Furthermore, the last-level cache is shared among all cores and divided into so-called cache slices. The hash function that maps physical addresses to slices is not publicly documented but has been reverse-engineered [39, 57, 88].

Cache Eviction. To mount a Rowhammer attack, memory accesses need to be directly served by the main memory. Thus, an attacker needs to make sure that the data is not stored in the cache. An attacker can use the unprivileged `clflush` instruction [87] to invalidate the cache line or use uncached memory if available [84]. On devices where no uncached memory and no unprivileged cache flush instruction is available, an attacker can instead evict a cache line by accessing congruent memory addresses [25, 28, 52], *i.e.*, addresses that map to the same cache set and same cache slice. Merely accessing a large number of different but congruent addresses in an arbitrary order typically does not lead to a high eviction rate. Gruss et al. [28] observed that to evict the victim address, a so-called eviction set of attacker-chosen congruent addresses has to be accessed in a specific pattern. The eviction set does not contain the victim address, which is consequently evicted from the cache.

Intel CAT. In 2016, Intel introduced Cache Allocation Technology (CAT) [41] to address quality of service in multi-core server platforms [32, 40]. Intel CAT allows system software to partition the last-level cache to optimize workloads in shared environments as well as to isolate applications or virtual machines in the cloud. When a virtual machine in the cloud thrashes the cache and therefore decreases the performance of other co-located machines, the hypervisor can restrict this virtual machine to a subset of the cache to retain the performance of other tenants. More specifically, Intel CAT allows restricting the number of cache ways available to processes, virtual machines, and containers, as illustrated in Figure 2. However, Aga et al. [4] showed that Intel CAT allows improving eviction-based Rowhammer attacks as it reduces the number of accesses required for cache eviction and consequently reduces the time required to evict an address from the cache.

3 NETHAMMER ATTACK

All previously published Rowhammer attacks rely on some form of code execution on the targeted device, be it the execution of a native binary [4, 27, 73, 78], an application [84] or using a scripted language in the web browser, like JavaScript [14, 25, 28]. In this section, we present Nethammer, the first Rowhammer attack that does not rely on any attacker-controlled code on the victim machine.

3.1 Attack Overview

Nethammer sends a crafted stream of network packets to the target device to mount a one-location or single-sided Rowhammer attack by exploiting quality-of-service technologies deployed on the device. For each packet received on the target device, a set of addresses is accessed, either in the kernel driver or a user-space application processing the contents. By repeatedly sending packets, this set of addresses is hammered and, thus, bit flips may be induced. As frequently-used addresses are served from the cache for performance, the cache must be bypassed such that the access goes directly into the DRAM to cause the row conflicts required for hammering. This can be achieved in different ways if the code that is executed (in kernel space or user space) when receiving a packet,

(1) *flushes* (and later on reloads) an address;
(2) uses *uncached* memory;
(3) *evicts* (and later on reloads) an address.

All three scenarios are plausible. *Uncached* memory is often used on ARM-based devices for interaction with the hardware, e.g., access buffers used by the network controller. Intel x86 processors have the `clflush` instruction for the same purpose. We verified that an attack is practical in both scenarios, as we describe in Section 6.2.

As caches are large, and cache replacement policies try to keep frequently-used data in the cache, it is not trivial to mount an eviction-based attack without executing attacker-controlled code on the device. However, to address quality of service in multi-core server platforms, Intel introduced CAT (cf. Section 2.3), allowing to control the amount of cache available to applications or virtual machines dynamically as illustrated in Figure 2. If a virtual machine is thrashing the cache, the hypervisor limits the number of cache ways available to this virtual machine to meet performance guarantees given to other tenants on the same physical machine. Thus, if an attacker excessively uses the cache, its virtual machine is restricted to a low number of ways, possibly only one, leading to a fast self-eviction of addresses.

3.2 Attack Setup

In our attack setup, the attacker has a fast network connection to the victim machine, e.g., a gigabit connection. We assume that the victim machine has DDR2, DDR3, or DDR4 memory that is susceptible to one-location (or single-sided) hammering.

Personal Computers. For our attack on personal computers, tablets, smartphones, or devices with similar hardware configuration, we make no further assumptions.

Cloud Systems. For our attack on cloud systems, we assume that the victim is running a virtual machine on a cloud server providing an interface or API accessible over the network. Furthermore, to prevent denial-of-service situations due to cache thrashing, we assume that the hypervisor on the cloud server uses Intel CAT to constrain the virtual machine of the victim to a subset of the cache.

Note that there are overlaps between the two attack setups. A personal computer can be susceptible to the attack we describe for the cloud scenario. Even more likely a cloud system can be susceptible to the attack we describe for personal computers.

3.3 Inducing Bit Flips over Network

To induce bit flips remotely, one requirement is to send as many packets as possible over the network in a short time frame. As defined in Section 3.2, we assume that either uncached memory or `clflush` is used when receiving a network packet or alternatively, that Intel CAT is active on the victim machine. Thus, every single packet processed by the network stack actively evicts and reloads data from the cache. By sending many packets, the corresponding addresses are hammered efficiently.

As an example, UDP packets without content can be used, allowing an overall packet size of 64 B, which is the minimum packet size for an Ethernet packet. This allows to send up to 1 024 000 packets per second over a 500 Mbit/s connection.

4 FROM REGULAR MEMORY ACCESSES TO ROWHAMMER

Naturally, several challenges need to be solved to induce Rowhammer bit flips over the network. Fundamentally, we need to investigate memory-controller page policies to determine whether regular memory accesses that occur while handling network packets could at least, in theory, induce bit flips. Note that these investigations are oblivious to the specific technique to access the DRAM row (*i.e.*, eviction, flushing, uncached memory). Hence, in this section, we do not discuss `clflush`, uncached memory, or eviction strategies with [4] or without Intel CAT [28, 52]. We defer comparisons of Nethammer with these techniques to Section 6. In this section, we focus on the underlying behavior of the memory controller and what this means for possible attacks.

Gruss et al. [27] found that the memory-controller page policy has a significant influence on the way the Rowhammer bug can be triggered. In particular, they found that one-location hammering works and deduced from this that the memory-controller page policy must be similar to a closed-page policy. Most previous work on Rowhammer assumed an open-row policy [4, 9, 11, 50, 62, 69, 73, 78, 84, 86]. In Section 4.1, we propose a method to determine the memory-controller page policy on real-world systems automatically. We show that one-location hammering does not necessarily need a closed-page policy, but instead, adaptive policies may allow one-location hammering.

Based on these insights, we demonstrate the first one-location Rowhammer attack on an ARM device in Section 4.2, and draw the connection to the attack presented by Aga et al. [4]. Finally, we investigate whether Rowhammer via network packets is theoretically possible. Network packets do not arrive with the same speed as the memory accesses in an optimized tight loop.

4.1 Automated Classification of Memory-Controller Page Policies

Gruss et al. [27] stated that a requirement for one-location hammering is a policy similar to a closed-page policy. To get a more in-depth understanding of the memory-controller page policy used on a specific system, we present an automated method to detect the used policy. This is a significant step forward for Rowhammer attacks, as it allows to deduce whether specific attack variants may or may not work without an empiric evaluation. Pessl et al. [68] reverse-engineered the undocumented mapping functions of physical memory addresses to DRAM channels, ranks and banks. These mapping functions allow selecting addresses located in the same bank but in a different row. If we access these addresses consecutively, we will cause a row conflict in the corresponding bank. This row conflict induces latency for the second access because the currently active row must be closed (written back), the bank must be pre-charged, and only then the new row can be fetched with an activate command. This side-channel information can not only be used to build a covert communication channel [68], but as we show, it can also be used to detect the page policy used by the memory controller.

Automated Classification of the Page-policy. We assume knowledge of processor and DRAM timings. For the DRAM this means in particular, the `tRCD` latency (the time to select a column address), and the `tRP` latency (the time between pre-charge and row activation). These three timings influence the observed latency as follows:
(1) we consider the case **page open / row hit** as the base line;
(2) in the case **page empty / bank pre-charged**, we observe an additional latency of `tRP` over a row hit;
(3) in the case **page miss / row conflict**, we observe an additional latency of (`tRP` + `tRCD`) over a row hit.
To compute the actual number of cycles we can expect, we have to divide the DRAM latency value by the DRAM clock rate. In case of DDR4, we have to additionally divide the latency value by factor two, as DDR4 is double-clocked. This yields the latency in nanoseconds. By dividing the nanoseconds by the processor clock speed, we obtain the latency in CPU cycles. Still, as we cannot obtain absolutely clean measurements due to out-of-order execution, prefetching, and other mechanisms that aim to hide the DRAM latency, the actually observed latency will deviate slightly.

As in our test we cannot simply measure the three different cases, we define an experiment that allows to distinguish the different policies. In the experiment we use for our automated classification, we select two addresses *A* and *B* that map to the same bank but different rows. Using the `clflush` instruction, we make sure that *A* and *B* are not cached, in order to load those addresses directly from main memory. We base our method on two observations for open-page policies:
Single By loading address *A* an increasing number of times ($n = 1..10\,000$) before measuring the time it takes to load the same address on a subsequent access, we can measure the access time of an address in DRAM if the corresponding row is already active. For an open-page policy the access time should be the same for any *n*.
Conflict By accessing address *A* and subsequently measuring the access time to address *B*, we can measure the access time of an address in DRAM in the occurrence of a row conflict.

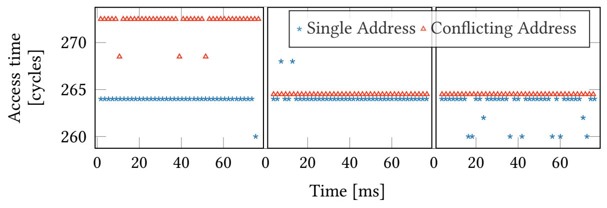


Figure 3: Measured access times over a period of time for a single address (blue) and an address causing a row conflict (red) for different page policies on the Intel Xeon D-1541: open policy (left), closed policy (middle), adaptive policy (right).

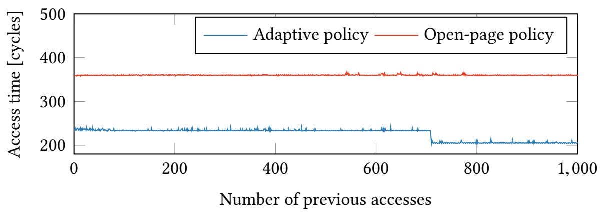


Figure 4: Open-page policy and adaptive page policy can be distinguished by testing increasing numbers of accesses to the same row. The open-page policy (Intel Core i7-4790) always has the same timing for subsequent accesses, since the row always remains open. The adaptive page policy (Intel Xeon E5-1630v4) only leaves the row open for a longer time after a larger number of accesses.

Our classification now works by running the following checks:
(1) If there is no timing difference between the two cases described above (**Single** with a large *n* and **Conflict**), the system uses a closed-page policy. The closed-page policy immediately closes the row after every read or write request. Thus, there is no timing difference between these two cases. The timing observed corresponds to the row-pre-charged state.
(2) Otherwise, if the timing for the **Single** case is the same, regardless of the value of *n*, but differs from the timing for **Conflict**, the system uses an open-page policy. The timing difference corresponds to the row hits and row conflicts. Following the definition of the open-page policy, the timing for row hits is always the same.
(3) Otherwise, the timing for the **Single** case will have a jump at some *n* after which the page policy is adapted to cope better with our workload. Consequently, the timing differences we observe correspond to row hit and row-pre-charged states.

Figure 3 shows the memory access time measured on an Intel Xeon D-1541 with different page policies. The plot shows that closed-page policy can be distinguished from the other two using our method. We also verified our results by reading out the `CLOSE_PG` bit in the `mcmtr` configuration register of the integrated memory controller [42].

We validated that we can distinguish open-page policy and adaptive page policy by running our experiments on two systems with the corresponding page policies. Figure 4 shows the results of these experiments. The difference between open-page policy and adaptive policy is clearly visible.

Our experiments show that adaptive page policies often behave like closed-page policies. This indicates the possibility of one-locating hammering on systems using an adaptive page policy.

4.2 One-location Hammering on ARM

To make Nethammer a more generic attack, it is essential to demonstrate it not only on Intel CPUs but also on ARM CPUs. This is particularly interesting as ARM CPUs dominate the mobile market, and ARM-based devices are predominant also in IoT applications. Gruss et al. [27] only demonstrated one-location hammering on Intel CPUs. However, as one-location hammering is the most plausible hammering variant for Nethammer, we need to investigate whether it is possible to trigger one-location hammering bit flips on ARM.

In our experiments, we used a LG Nexus 4 E960 mobile phone equipped with a Qualcomm Snapdragon 600 (APQ8064) [71] SoC and 2GB of LPDDR2 RAM, susceptible to bit flips using double-sided hammering. The page policy used by the memory controller is selected via the `DDR_CMD_EXEC_OPT_0` register: if the bit is set to 1, which is the recommended value [72], a closed-page policy is used. If the bit is set to 0, an open-page policy is used. Hence, we can expect the memory controller to preemptively close rows, enabling one-location hammering.

So far, bit flips on ARM-based devices have only been demonstrated in the combination of double-sided hammering, and uncached memory [84] or access via the GPU [25]. Even in the presence of a flush instruction [7] or optimal cache eviction strategies [52], the access frequency to the two neighboring rows is too low to induce bit flips. Furthermore, devices with the ARMv8 instruction set that allows exposing a flush instruction to unprivileged programs are usually equipped with LPDDR4 memory.

In our experiment, we allocated uncached memory using the Android ION memory allocator [90]. We hammered a single random address within the uncached memory region at a high frequency and then checked the memory for occurred bit flips. We were able to observe 4 bit flips while hammering for 10 hours. Thus, we can conclude that there are ARM-based devices that are vulnerable to one-location hammering.

4.3 Minimal Access Frequency for Rowhammer Attacks

A show stopper for Nethammer is if the frequency of memory accesses caused by processing network packets is not high enough to induce bit flips on one of our test systems successfully. As the system performs many memory accesses when handling a network packet, the attacker, in fact, cannot tell whether only one location in a bank is hammered (*i.e.*, one-location hammering) or multiple locations (*i.e.*, single-sided hammering). In particular, following the pigeon-hole principle, in our test setups with 32 bank (single DIMM) or 64 bank (dual DIMM) setups, we know that, if we access at least $n + 1$ different addresses, *i.e.*, 33 or 65 respectively, at least two addresses must be served from the same bank. Hence, we can assume that there is a good probability that the attacker actually does single-sided hammering. Moreover, some addresses are accessed multiple times.

Previous work has investigated the minimal number of accesses that are necessary within a 64 ms refresh interval to still obtain bit flips. Kim et al. [50] reported bit flips starting at 139 000 row activations per refresh interval, which can be, depending on the page policy, identical to the number of memory accesses. Gruss et al. [28] reported bit flips starting at 43 000 and Aweke et al. [9] at 110 000 memory accesses per refresh interval.

In our experiments, we send 500 Mbit/s (and more) over the network interface. With a minimum size of 64 B for Ethernet packets, we can send 1 024 000 packets per second over a 500 Mbit/s connection. As described in Section 6.2, we found functions which are called multiple times, e.g., 6 times in the case of once function. Hence, on a 500 Mbit/s connection, the attack can induce 6 144 000 accesses per second. Divided by the default refresh interval of 64 ms, we are at 393 216 accesses per refresh interval. This is clearly above the previously reported required number of memory accesses [9, 28, 50]. Hence, we conclude that in theory, if the system is susceptible to Rowhammer attacks, network packets can induce bit flips. In the following section, we will describe how an attacker can exploit such bit flips.

5 EXPLOITING BIT FLIPS OVER A NETWORK

In this section, we discuss Nethammer attack scenarios to exploit bit flips over the network in detail. We discuss the possible locations of bit flips in Section 5.1. We describe different Nethammer attacks in Section 5.2.

5.1 Bit Flip Location and Effect

As the Nethammer attack does not control where in physical memory a bit flip is induced and, thus, what is stored at that location, the bit flip can lead to different consequences. On a high level, we can divide bit flips into two groups, based on the location of the flip. We distinguish between bit flips in user memory, *i.e.*, memory pages that are mapped as `user_accessible` in at least one process, and bit flips in kernel memory, *i.e.*, memory pages that are never mapped as `user_accessible`. We can also distinguish the bit flips based on their high-level effect, again forming two groups. The first group consists of bit flips that lead to a denial-of-service situation. The second group consists of bit flips that do not lead to a denial-of-service situation. If a denial-of-service situation is temporary, a system reboot may be necessary. A denial-of-service situation can be persistent if the bit flip is written back to a permanent storage location. Then it may be necessary to reinstall the system software or parts of it from scratch, clearly taking more time than just a reboot. Denial-of-service attacks have a direct financial impact on companies due to unplanned downtimes and maintenance times. Moreover, studies show that their announcement can also have a negative impact on the stock prices [1]. Consequently, Nethammer poses a severe threat to servers vulnerable to the attack.

5.2 Bit Flip Targets

Nethammer may induce a bit flip in *kernel memory*. Depending on the modified location, parts of the operating system can behave unexpectedly, or the entire system may even halt. Bit flips in *user memory* may have similar consequences.

5.2.1 File System Data Structures. File system data structures, e.g., inodes, are not directly part of the kernel code or data but are also in the kernel memory. An inode is a data structure defining a file or a directory of a file system. Each inode contains metadata such as the size of the file, owner and permission data as well as the disk block location of its data. If a bit flips in the inode structure, it corrupts the file system and, thus, causes persistent loss of data. This may again crash the entire system.

5.2.2 SGX Enclave Page Cache. If the victim machine supports Intel SGX [19], an x86 instruction-set extension that allows the execution of programs in so-called *secure enclaves* to run with integrity and confidentiality guarantees in untrusted environments, a bit flip easily causes a denial of service. Enclave memory is stored in a physically contiguous block of memory that is encrypted using a Memory Encryption Engine [30]. Jang et al. [44] and Gruss et al. [27] demonstrated that if a bit flip in enclave memory is induced, the Memory Encryption Engine locks the memory controller, preventing any future memory operations and thus, halting the entire system. While such a bit flip is not persistent itself, the unsafe halting of the entire system can leave permanent damage leading to a persistent denial-of-service.

5.2.3 Application Memory in General. If a bit flip occurs in memory of a user-space application, e.g., code or data, a possible outcome is the crash of the program. Such a flip may render the affected service unavailable.

Another outcome of a bit flip in the data of a user-space application, e.g., in the database of a service, is that the service delivers modified, possibly invalid, content. Depending on the service, its users cannot distinguish if the data is correct or has been altered.

Altering DNS Entries to redirect to Malicious Services. To resolve domain names to the corresponding IP address, a DNS request [60] is sent to a DNS server. DNS servers are organized in a tree-like structure, building a distributed system to store DNS records. A record consists of a type, a name, a class code, a time-to-live for caching, and the value. For instance, the `A` record holds a 32-bit IPv4 address for a specific domain. However, DNS allows defining aliases to map one domain name to another. This is used to define message transfer agents for a domain or to redirect domains.

In this attack, the attacker leverages Nethammer to induce a bit flip in a character of a DNS entry to make it point to a different domain. For instance, `domain.com` changes to `dnmain.com` if the least-significant bit of the "o" character is flipped from '1' to '0'. Such an attack is also referred to as bitsquatting [20]. Such bit flips in domains have been successfully exploited before using Rowhammer attacks [73]. DNS zone transfers (AXFR queries) allow replicating DNS databases across different servers. Using zone transfers, an attacker can retrieve entries of an entire zone at once. The attacker queries the DNS server for its entries, mounts the attack and then verifies whether a bit flip at an exploitable position has occurred by monitoring changes in the queried entries. If so, the attacker can register the changed domain and host a malicious service on the domain, e.g., a fake website to steal login credentials or a mail server intercepting email traffic. Users querying the DNS server for said entry connect to the server controlled by the attacker and are thus exposed to data theft. A flip might also change an MX entry, pointing it to a different domain. The attacker can then again register the domain and intercept connections that were intended to go to the original mail server.

Rebuilding Trust in Revoked Certificates. An attacker can also target OCSP servers. The Online Certificate Status Protocol (OCSP) is a protocol to retrieve the revocation status of a certificate [77]. In contrast to a certificate revocation list, where all revoked certificates are enumerated, the OCSP protocol enables to query the status of a single certificate. This protocol shifts the workload from the user to the OCSP server, so that users, or more specifically browsers, do not have to store huge revocation lists. Instead, the OCSP server manages a list of revoked certificate fingerprints.

Digital certificates are used to generate digital signatures that present the authenticity of digital documents or messages. They are typically obtained from a trusted party, e.g., a certificate authority. The certificate allows verifying that a specific signature was indeed issued by the signer. However, if the corresponding private key of a certificate is exposed to the public, everyone can sign data in the name of the signer. Hence, a user can revoke a certificate to avoid any abuse. Liu et al. [53] evaluated 74 full IPv4 HTTPS scans and found that 8% of 38 514 130 unique SSL certificates served have been revoked.

To process a certificate validity request, the server queries its database for the requested certificate identifier. The result can either be that the certificate is revoked, not revoked (*i.e.*, valid), or that the state is unknown (*i.e.*, it is not in the database). If a client tries to establish a secure connection to a server or check the validity of a signed document, it queries the OCSP server provided by the certificate. If the certificate has been revoked, the client aborts the connection or marks the signature as invalid.

In this attack, the attacker flips a bit in the memory of an OCSP server of a certificate authority where private keys of certificates have become public, and the certificates have thus been revoked. The attacker can either flip the status or the identifier of the certificate. As the status of the certificate is stored as an ASCII character in the OpenSSL OCSP responder [65], one bit flip is sufficient to flip the "R" (revoked) to "V" (valid). Assuming the memory is filled with revocation list entries, which are on average 100 B for this specific responder, an attacker has a chance of 0.125 % *per bit flip* to make a random certificate valid again. Thus, an attacker can again reuse that certificate (with the known private key) to sign documents or data and, thus, impersonate the original signer.

A weaker, but more likely attack scenario, is to flip a bit in the certificate identifier. Such a bit flip leads to the OCSP server not finding the certificate in its database anymore, thus, returning "unknown" as the state. Most browsers fall back to their own certificate revocation list in such a case [2, 56, 66]. However, only high-value revocations are kept in the browser's list, making it very unlikely that the certificate is in the certificate revocation list of the browser [2]. Hence, an attacker can again reuse that certificate.

Other attacks. The attack scenarios described above are by far not exhaustive. With bit flips in applications, attackers have numerous possibilities to modify random data, yielding different, disastrous consequences. However, the outlined attacks highlight the severity of remotely induced bit flips by Nethammer.

5.2.4 *Cryptographic Material.*

Cryptographic material as part of the application memory is particularly interesting for attacks. In the past, it has been demonstrated that fault attacks on RSA public keys result in broken keys which are susceptible to key factorization [10, 16]. Therefore, also public key material has to be protected against faults. Muir [61] remarked that a bit flip in an RSA public key allows an attacker with a non-negligible probability to compute a private key corresponding to the modified key in a reasonable amount of time. Thus, an attacker can flip a bit of a public RSA key in memory using Nethammer, giving the attacker the same privileges and permissions as the owner of the original key. These permissions are only temporary, e.g., until the key is reloaded from the hard drive.

Distribution of Malicious Software on Version-Control Hosting Services. An attacker can compromise a hosting service to distribute malicious software. The number of organizations using hosting services for revision control to manage changes to their source code, documents or other information is increasing steadily. These services can either be subscription based, e.g., GitHub [36], or self-hosted, e.g., GitLab [37], and, thus, are deployed on many web servers to distribute their software.

To commit changes to a version-controlled repository, users authenticate with the service using public-key cryptography. Typically, users generate an SSH key pair [89], e.g., using RSA [75], upload the public key to the service, and store the private key securely on their local system. As the position of the bit flip cannot be controlled using Nethammer, an attacker can improve the probability to induce a bit flip in the modulus of a public key by loading as many keys as possible into the main memory of the server. Some APIs, e.g., the GitLab API [38], allow enumerating the users registered for the service as well as their public keys. By enumerating and, therefore, accessing all public keys of the service, the attacker loads the public keys into the DRAM.

In the first step of the attack, the attacker enumerates all keys of all users and stores them locally. In the second step, the attacker mounts Nethammer to induce bit flips on the targeted system. The more keys the attacker loaded into memory, the more likely it is that the bit flip corrupts the modulus of a public key of a user. For instance, with 80 % of the memory filled with 4096-bit keys, the chance to hit a bit of a modulus is 79.7 %. As the attacker does not know which key was affected by the bit flip, the attacker enumerates all keys again and compares them with the locally stored keys. If a modified key has been found, the attacker computes a new corresponding private key [61, 73]. The attacker uses this new key to authenticate with the service, impersonating the user. Consequently, the attacker can make changes to the software repository as that user and, thus, introduce bugs that can later be exploited if the software is distributed. The original public key will be restored after a while when the key is evicted from the page cache and has to be reloaded from the hard drive. As the correct key is restored, the attack leaves no traces. Furthermore, it also breaks the non-repudiation guarantee provided by the public-key authentication, making the victim whose public key was attacked the prime suspect in possible investigations.

6 EVALUATION

In this section, we evaluate Nethammer and its performance. We show that the number of bit flips induced by Nethammer depends on how the cache is bypassed and the memory-controller's page policy. We evaluate which kernel functions are executed when handling a UDP network packet. We describe the bit flips we obtained when running Nethammer in different attack scenarios. Finally, we show that TRR does not protect against Nethammer or Rowhammer in general.

6.1 Environment

In our evaluation, we used the test systems listed in Table 1. We used the first system for our experiments with a non-default network driver implementation that uses `clflush` in the process of handling a network packet, and the second and third system for our experiments with Intel CAT. To mount Nethammer, we used a Gigabit switch to connect two other machines with the victim machine. The two other machines were used to flood the victim machine with network packets triggering the Rowhammer bug. We used the fourth system for our experiments on an ARM-based device that uses uncached memory in the process of handling a network packet.

6.2 Evaluation of the Different Cache Bypasses for Nethammer

In Section 4, we investigated the requirements to trigger the Rowhammer bug over the network. In this section, we evaluate Nethammer for the three cache-bypass techniques (see Section 3.1): a kernel driver that flushes (and reloads) an address whenever a packet is received, Intel Xeon CPUs with Intel CAT for fast cache eviction, and uncached memory on an ARM-based mobile device.

Driver with `clflush`. To verify that Nethammer can induce bit flips, we used a non-default network driver implementation that uses `clflush` in the process of handling a network packet on an Intel i7-6700K CPU. We sent UDP packets with up to 500 Mbit/s and scanned memory regions where we expected bit flips. We observed a bit flip every 350 ms showing that hammering over the network is feasible if at least two memory accesses are served from main memory, due to flushing an address while handling a network packet. Thus, in this scenario, up to 10 000 bit flips per hour can be induced.

Eviction with Intel CAT. The operating system will handle every network packet received by the network card. The operating system parses the packets depending on their type, validates their checksum and copies and delivers every packet to each registered socket queue. Thus, for each received packet quite some code is executed before the packet finally arrives at the application destined to handle its content.

We tested Nethammer on Intel Xeon CPUs with Intel CAT. The number of cache ways has been limited to a single one for code handling the processing of UDP packets, resulting in fast cache eviction. If a function is called multiple times for one packet, the same addresses are accessed and loaded from DRAM with a high probability, thus, hammering this location. To estimate how many different functions are called and how often they are called, we

Table 1: List of test systems that were used for the experiments.

Device	CPU	DRAM	Network card	Operating system
Desktop	Intel i7-6700K @ 4 GHz	8 GB DDR4 @ 2133 MHz	Intel 10G X550T	Ubuntu 16.04
Server	Intel Xeon E5-1630v4 @ 3.7 GHz	8 GB DDR4 @ 2133 MHz	Intel i210/i218-LM Gigabit	Xubuntu 17.10
Server	Intel Xeon D-1541 @ 2.1 GHz	8 GB DDR4 @ 2133 MHz	Intel i350-AM2 Gigabit	Ubuntu 16.04
LG Nexus 4	Qualcomm APQ8064 @ 1.5 GHz	2 GB LPDDR2 @ 533 MHz	USB Adapter	Android 5.1.1

use the *perf* framework [22] to count the number of function calls related to UDP packet handling. Appendix A shows the results of a system handling UDP packets. Out of 27 different functions we identified, most were called only once for each received packet. The function `__udp4_lib_lookup` is called twice. In a more extensive profiling scan, we found that `nf_hook_slow` is called 6 times while handling UDP packets on some kernels.

With this knowledge, we analyzed how many bit flips can be induced from this code execution. We observed 45 bit flips per hour on the Intel Xeon E5-1630v4. As TRR is active on this system (see Section 6.5), fewer bit flips occur in comparison to systems without TRR. In Section 6.3, we evaluate the number of bit flips on the Intel Xeon D-1541 depending on the configured page policy.

Uncached Memory. In Section 4.2, we demonstrated that ARM-based devices are vulnerable to one-location hammering in general. To investigate whether bit flips can also be induced over the network, we connect the LG Nexus 4 using an OTG USB ethernet adapter to a local network. Using a different machine, we send as many network packets as possible to the mobile phone. An application on the phone allocates memory and repeatedly checks the allocated memory for occurred bit flips. However, we were not able to observe any bit flips on the device within 12 hours of hammering. As the device does not deploy a technology like Intel CAT (Section 2.3), the cache is not limited for certain applications and, thus, the eviction of code or data used by handling memory packets has a low probability. As network drivers often use DMA memory and, thus, uncached memory, bit flips induced by the network are more likely if the network driver itself uses uncached memory. While we identified a remarkable number of around 5500 uncacheable pages used by the system, we were not able to induce any bit flips over the network. However, we found that the USB ethernet adapter only allowed for a network capacity of less than 16 Mbit/s, which is clearly too slow for a Nethammer attack. It is very likely that with a faster network connection, e.g., more than 200 Mbit/s, it is possible to induce bit flips. Nevertheless, we were successfully able to induce bit flips using Nethammer on the Intel Xeon E5-1630v4 where one uncached address is accessed for every received UDP packet.

6.3 Influence of Memory-Controller Page Policies on Rowhammer

In order to evaluate the actual influence of the used memory-controller page policy on Nethammer, *i.e.*, how many bit flips can be induced depending on the policy used, we mounted the Nethammer in different settings. The experiment was conducted on our Intel Xeon D-1541 test system, as the BIOS of its motherboard allowed to

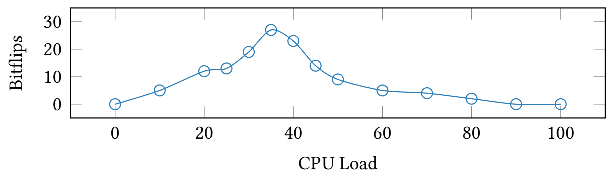


Figure 5: Number of bit flips depending on the CPU load with a closed-page policy after 15 minutes (Xeon D-1541).

chose between different page policies: *Auto*, *Closed*, *Open*, *Adaptive*. For each run, we configured the victim machine with one of the policies and Intel CAT, and, mounted a Nethammer attack for at least 4 hours. To detect bit flips, we ran a program on the victim machine that mapped a file into memory. The program then repeatedly scans the content of all allocated pages and reports bit flips if the content has changed.

We detected 11 bit flips in 4 hours with the *Closed* policy, with the first one after 90 minutes. We did not observe any bit flips with the *Open* policy within the first 4 hours. However, when running the experiment longer, we observed 46 bit flips within 10 hours. With the *Adaptive* policy, we observed 10 bit flips in 4 hours, with the first one within the second hour of the experiment. While this experiment was conducted without any additional load on the system, we see in Figure 5 that additional CPU utilization increases the number of bitflips drastically. Using the *Closed* policy, we observed 27 bitflips with a load of 35% within 15 minutes.

These results do not immediately align with the assumption that a policy that preemptively closes rows is required to induce bit flips using one-location hammering. However, depending on the addresses that are accessed and the constant eviction through Intel CAT, it is possible that two addresses map to the same bank but different rows and, thus, bit flips can be induced through single-sided hammering. In fact, the attacker cannot know whether the hammering was actually one-location hammering or single-sided hammering. However, as long as a bit flip occurs, the attacker does not care how many addresses mapped to the same bank. Finally, depending on the actual parameters used by a fixed-open-page policy, a row can still be closed early enough to induce bit flips.

6.4 Bit Flips induced by Nethammer

As described in Section 5.1, a bit flip can occur in user space or kernel space leading to different effects depending on the memory it corrupts. In this section, we present bit flips that we have observed in our experiments and the effects they have caused.

Kernel image corruption and kernel crashes. We observed Nethammer bit flips that caused the system not to boot anymore. It stopped responding after the bootloader stage. We inspected the kernel image and compared it to the original kernel image distributed by the operating system. As the kernel image differed blockwise at many locations, we assume that the Nethammer caused a bit flip in an inode of the file system. The inode of a program that wanted to write data did not point to the correct file any longer but to the kernel image and, thus, corrupted the kernel image.

Furthermore, we observed several bit flips immediately halting the entire system such that interaction with it was not possible any longer. By debugging the operating system over a serial connection, we detected bit flips in certain modules such as the keyboard or network driver. In these cases, the system was still running but did not respond to any user input or network packets anymore. We also observed bit flips that were likely in the SGX EPC region, causing an immediate permanent locking of the memory controller.

Bit flips in user space libraries and executables. We observed that bit flips crashed running processes and services or prevented the execution of others as the bit flip triggered a segmentation fault when functions of a library were executed. On one occasion, a bit flip occurred either in the SSH daemon or the stored passwords of the machine, preventing any user to login on the system. The system was restored to a stable state only by rebooting the machine and thus reloading the entire code from disk.

We also validated that an attacker can increase chances to flip a bit in a target page by increasing the memory usage of a user program. In fact, this was the most common scenario, overlapping with our general test setup to detect bit flips for our evaluation. Unsurprisingly, these bit flips equally occur when filling the memory with actual contents that the attacker targets.

6.5 Target Row Refresh (TRR)

Previous assumptions on the Rowhammer bug lead to the conclusion that only bit flips in the victim row adjacent to the hammering rows would occur. While the probability for bit flips to occur in directly adjacent rows is much higher, Kim et al. [50] already showed rows further away (even a distance of 8 rows and more) are affected as well. Still, the hardware vendors opted for implementing countermeasures focusing on the directly adjacent rows.

With the Low Power Double Data Rate 4 (LPDDR4) standard, the JEDEC Solid State Technology Association [46] defines a reliability feature called Target Row Refresh (TRR). The idea of TRR is to refresh adjacent rows if the targeted row is accessed at a high frequency. More specifically, TRR works with a maximum number of activations allowed during one refresh cycle, the maximum active count. Thus, if a double-sided Rowhammer attack (Section 2.2) is mounted, and two hammered rows are accessed more than the defined maximum active count, the adjacent rows (in particular the victim row of the attack) will be refreshed. As the potential victim rows are refreshed, in theory, no bit flip will occur, and the attack is mitigated. However, in practice, bit flips can be further away from the hammered rows and thus TRR may be ineffective.

With the Ivy Bridge processor family, Intel introduced Pseudo Target Row Refresh (pTRR) for Intel Xeon CPUs to mitigate the Rowhammer bug [55]. On these systems pTRR-compliant DIMMs must be used; otherwise, the system will default into double refresh mode, where the time interval in which a row is refreshed is halved [55]. However, Kim et al. [50] showed that a reduced refresh period of 32 ms is not sufficient enough to impede bit flips in all cases. While pTRR is implemented in the memory controller [54], DRAM module specifications theoretically allow automatically running TRR in the background [58].

In our experiments, we were able to induce bit flips on a pTRR-supporting DDR4 module using double-sided hammering on an Intel i7-6700K. The bit flips occurred in directly adjacent rows and rows further away. We observed that when using the same DDR4 DRAM on the Intel Xeon E5-1630 v4 CPU, no bit flips occurred in the directly adjacent rows, but we observed no statistically significant difference in the number of bit flips for the rows further away. This indicates that TRR is active on the second machine but also that TRR does not prevent the occurrence of exploitable bit flips in practice. Thus, we conclude that the TRR hardware countermeasure is insufficient in mitigating Rowhammer attacks.

7 COUNTERMEASURES

Since Nethammer does not require any attack code in contrast to a regular Rowhammer attack, e.g., no attacker-controlled code on the system, most countermeasures do not prevent our attack.

Countermeasures based on static code analysis aim to detect attack code in binaries [43]. However, as our attack does not use any suspicious code and does not execute a program, these countermeasures do not detect the ongoing attack. Other countermeasures detect on-going attacks using hardware- and software-based performance counters [17, 18, 29, 31, 67, 91] and subsequently stop the corresponding programs. However, when hammering over the network, the large amount of memory accesses are executed by the kernel itself, and the kernel cannot just be terminated or stopped like a regular program. Hence, these countermeasures cannot cope with our attack. Modifying the system memory allocator to hinder the exploitability of bit flips [15, 28, 84] may generally work against Nethammer. However, the hammering is in practice done by the kernel, so the proposed isolation schemes are ineffective, and new schemes have to be proposed.

ANVIL [9] uses performance counters to detect and subsequently mitigate Rowhammer attacks. Since ANVIL, in its current form, does not detect one-location hammering [27], it also does not detect our attack. While we believe an adapted version of ANVIL could detect our attack, it would require evaluating whether the false positive and false negative rates allow for an application in practice. B-CATT [15] blacklists vulnerable locations, thus, effectively reducing the amount of usable memory, but fully eliminating the Rowhammer bug. B-CATT would work against Nethammer, but previous work has found that it is not practical as it would block too much memory [27, 50].

In general, we recommend reviewing any network stack and network services code. Uncached memory and `clflush` instructions should only be used with extreme care, and it may even be necessary to add artificial slow downs such that they cannot be exploit for Nethammer attacks anymore. If this is not possible for technical reasons, the threat model of the device should be revisited and reevaluated. Mitigating our eviction-based Nethammer attack might be more straight-forward, as it requires a specific configuration for Intel CAT. Either avoiding the restriction to a low number of cache ways via Intel CAT on network-connected systems or installing ECC memory would likely be sufficient to make our attack very improbable to succeed. Hence, we also recommend using Intel CAT very carefully in network-connected systems.

8 DISCUSSION

Hardware requirements. To induce the Rowhammer bug, one needs to access memory in the main memory repeatedly and, thus, needs to circumvent the cache. Therefore, either native flush instructions [87], eviction [4, 28] or uncached memory [84] can be used to remove data from the cache. In particular, for eviction-based Nethammer, the system must use Intel CAT as described in Section 2.3 in a configuration that restricts the number of ways available to a virtual machine in a cloud scenario to guarantee performance to other co-located machines [40]. If none of these capabilities are available over the network, an attacker could not mount Nethammer in practice.

One limitation of our attack is that only DRAM susceptible to bit flips can be exploited using a Rowhammer attack and, thus, Nethammer. To reduce the risk of bit flips on servers, one would assume that cloud providers tend to deploy ECC RAM usually. However, many cloud providers offer to rent hardware without ECC RAM [23, 33, 34, 63, 64, 85], potentially allowing Nethammer attacks. DRAM with ECC can only be used in combination with Intel Xeon CPUs and can detect and correct 1-bit errors. Therefore it can deal with single bit flips. While non-correctable multi-bit flips can be exploitable [5, 6, 51], they often end up in a denial-of-service attack depending on the operating system's response to the error.

Network traffic. Nethammer sends as many network packets to the victim machine as possible, aiming to induce bit flips. Depending on the actual attack scenario (see Section 5), additional traffic, e.g., by enumerating the public keys of the service, is generated. If the victim uses network monitoring software, the attack might be detected and stopped, due to the highly increased amount of traffic. In our experiments, we sent a stream of UDP packets with up to 500 Mbit/s to the target system. We were able to induce a bit flip every 350 ms and, thus, if the first random bit flip already hits the target or causes a denial-of-service, the attack could already be successful. However, as the rows are periodically refreshed, an attacker only needs an extraordinary high burst of memory accesses to a row between two refreshes, *i.e.*, within a period of 64 ms. Hence, an attacker could mount Nethammer for a few hundred milliseconds and then pause the attack for a longer time. These short network spikes may circumvent network monitoring software that might otherwise detect and prevent the on-going attack, e.g., by null routing the victim server.

Gigabit LTE on Mobile Devices. While ethernet adapters in mobile phones are uncommon, many ARM-based embedded devices in IoT setups are equipped and connected with gigabit ethernet. However, we expect the maximum throughput of these network cards to be too low on many of these devices, e.g., the Raspberry Pi 3 Model B+ [24], and also WiFi chips typically offer too little capacity. However, on more recent processors, e.g., the Qualcomm Snapdragon 845 chipset [70], and modems like the Qualcomm X20 Gigabit LTE modem, throughputs up to 1.2 Gbit/s are possible in practice. This would enable to send enough packets to hammer specific addresses to induce bit flips on the device and, thus, to successfully mount Nethammer.

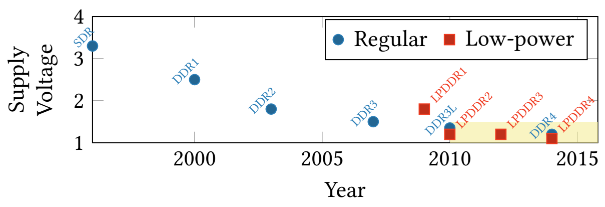


Figure 6: Minimum DRAM supply voltages for different DDR standards. The highlighted area marks the voltage and manufacturing years of DRAM modules where Rowhammer bit flips have been reported.

Influence of DRAM Supply Voltage on Rowhammer Effect. Kim et al. [50] identified voltage fluctuations as the root cause of DRAM disturbance errors, e.g., the Rowhammer bug. However, no study so far has investigated the direct effect of the DRAM supply voltage on Rowhammer bit flips. In fact, we can already observe a direct correlation between a low DRAM supply voltage by reviewing related work. Figure 6 shows how the DRAM voltage has been reduced over the past years. Previous work observed that the vulnerability of DRAM modules is related to the manufacturing date, *i.e.*, no bitflips before 2010 [50, 78]. However, as shown in Figure 6 there are at least two possible correlations with the Rowhammer bug, the manufacturing date, and the supply voltage.

Indeed, Rowhammer has only been reported on DRAM modules with a voltage below 1.5 volts [50, 78], *i.e.*, DDR3 [50, 78], DDR4 [68], LPDDR2 and LPDDR3 [84], and LPDDR4 [83].

We investigated the influence of the DRAM voltage on the occurrence of bit flips on two systems. We tested voltage increases in 0.01 V steps. On three tested systems (2× DDR4, 1× DDR3), we observed no significant change in the number of bit flips, *i.e.*, the number of bit flips stayed in the same order of magnitude, even when increasing the voltage by 0.2 V. Future work should investigate whether other voltage-related parameters could lead to a straightforward elimination of the Rowhammer bug.

9 CONCLUSION

In this paper, we presented Nethammer, the first truly remote Rowhammer attack, without a single attacker-controlled line of code on the targeted system. We demonstrate attacks on systems that use uncached memory or flush instructions while handling network requests, and systems that don't use either but are protected by Intel CAT. In all cases, we were able to induce multiple bit flips per hour on real-world systems, leading to temporary or persistent damage on the system. We showed that depending on the location, the bit flip compromises either the security and integrity of the system and the data of its users. In some cases, the system was rendered unbootable after the attack.

We presented a method to automatically identify the page policy used by the memory controller. Consequently, we found that adaptive page policies are also vulnerable to one-location hammering. While we were able to mount the first one-location hammering attack on an ARM device, the network capacity on this device was too low for Nethammer.

Transforming formerly local attacks into remote attacks is always a landslide in security. Assumptions that were true for the local scenario are largely invalid in a remote scenario. In particular, all defenses and mitigation strategies were designed against local Rowhammer attacks, *i.e.*, remote Rowhammer attacks were out of scope. Hence, Nethammer requires the re-evaluation of the security of millions of devices where the attacker is not able to execute attacker-controlled code. Finally, our work demonstrates that we need to develop countermeasures with the root cause of both local and remote Rowhammer attacks in mind.

ACKNOWLEDGEMENT

We thank Mattis Turin-Zelenko and Paul Höfler for help with some experiments. We thank Stefan Mangard, Anders Fogh, Thomas Dullien, and Jann Horn for fruitful discussions.

This work has been supported by the Austrian Research Promotion Agency (FFG) via the K-project DeSSnet, which is funded in the context of COMET – Competence Centers for Excellent Technologies by BMVIT, BMWFW, Styria and Carinthia. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402).

REFERENCES

[1] Abhishta, Reinoud Joosten, and Lambert J.M. Nieuwenhuis. 2017. Comparing Alternatives to Measure the Impact of DDoS Attack Announcements on Target Stock Prices. (2017).

[2] Adam Langley. 2014. Revocation still doesn't work. (2014). https://www.imperialviolet.org/2014/04/29/revocationagain.html

[3] Advanced Micro Devices. 2013. BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors. (2013). http://support.amd.com/TechDocs/42301_15h_Mod_00h-0Fh_BKDG.pdf

[4] Misiker Tadesse Aga, Zelalem Birhanu Aweke, and Todd Austin. 2017. When good protections go bad: Exploiting anti-DoS measures to accelerate Rowhammer attacks. In *International Symposium on Hardware Oriented Security and Trust.*

[5] Barbara Aichinger. 2015. DDR memory errors caused by Row Hammer. In *HPEC.*

[6] Barbara Aichinger. 2015. Row Hammer Failures in DDR Memory. In *memcon.*

[7] ARM Limited. 2013. *ARM Architecture Reference Manual ARMv8.* ARM Limited.

[8] Manu Awasthi, David W. Nellans, Rajeev Balasubramonian, and Al Davis. 2011. Prediction Based DRAM Row-Buffer Management in the Many-Core Era. In *PACT'11.*

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

[10] Alexandre Berzati, Cécile Canovas, and Louis Goubin. 2008. Perturbating RSA Public Keys: An Improved Attack. In *Cryptographic Hardware and Embedded Systems – CHES 2008.*

[11] Sarani Bhattacharya and Debdeep Mukhopadhyay. 2016. Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis. In *Conference on Cryptographic Hardware and Embedded Systems (CHES).*

[12] Eli Biham. 1997. A fast new DES implementation in software. In *International Workshop on Fast Software Encryption.* 260–272.

[13] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. 1997. On the Importance of Checking Cryptographic Protocols for Faults. In *Advances in Cryptology - EUROCRYPT '97.*

[14] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2016. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *S&P.*

[15] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2017. CAn't Touch This: Software-only Mitigation against Rowhammer Attacks targeting Kernel Memory. In *USENIX Security Symposium.*

[16] Eric Brier, Benoît Chevallier-Mames, Mathieu Ciet, and Christophe Clavier. 2006. Why One Should Also Secure RSA Public Key Elements. In *Cryptographic Hardware and Embedded Systems - CHES 2006.*

[17] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. 2015. Real time detection of cache-based side-channel attacks using Hardware Performance Counters. Cryptology ePrint Archive, Report 2015/1034. (2015).

[18] Jonathan Corbet. 2016. Defending against Rowhammer in the kernel. (Oct. 2016). https://lwn.net/Articles/704920/

[19] Victor Costan and Srinivas Devadas. 2016. Intel SGX explained. (2016).

[20] Artem Dinaburg. 2011. Bitsquatting: DNS Hijacking without Exploitation. (2011). http://media.blackhat.com/bh-us-11/Dinaburg/BH_US_11_Dinaburg_Bitsquatting_WP.pdf

[21] James M. Dodd. 2003. Adaptive page management. (2003). https://encrypted.google.com/patents/US7076617B2 US Patent Grant 2006-07-11.

[22] Jake Edge. 2009. Perfcounters added to the mainline. (Jul 2009). http://lwn.net/Articles/339361/

[23] fasthosts. 2018. High performance dedicated servers. (May 2018). https://www.fasthosts.co.uk/dedicated-servers

[24] Raspberry Pi Foundation. 2018. Raspberry Pi 3 Model B+. (March 2018). https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus

[25] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2018. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In *IEEE S&P.*

[26] Mohsen Ghasempour, Mikel Lujan, and Jim Garside. 2015. ARMOR: A Run-time Memory Hot-Row Detector. (2015). http://apt.cs.manchester.ac.uk/projects/ARMOR/RowHammer

[27] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. 2018. Another Flip in the Wall of Rowhammer Defenses. In *S&P'18.*

[28] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2016. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *DIMVA'16.*

[29] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA.*

[30] Shay Gueron. 2016. A Memory Encryption Engine Suitable for General Purpose Processors. Cryptology ePrint Archive, Report 2016/204. (2016).

[31] Nishad Herath and Anders Fogh. 2015. These are Not Your Grand Daddys CPU Performance Counters – CPU Hardware Performance Counters for Security. In *Black Hat Briefings.* https://www.blackhat.com/docs/us-15/materials/us-15-Herath-These-Are-Not-Your-Grand-Daddys-CPU-Performance-Counters-CPU-Hardware-Performance-Counters-For-Security.pdf

[32] Andrew Herdrich, Edwin Verplanke, Priya Autee, Ramesh Illikkal, Chris Gianos, Ronak Singhal, and Ravi Iyer. 2016. Cache QoS: From concept to reality in the Intel Xeon processor E5-2600 v3 product family. In *IEEE HPCA'16.*

[33] Hetzner. 2018. Dedicated Root Server Hosting. (May 2018). https://www.hetzner.com/dedicated-rootserver/

[34] DefineQuality Hosting. 2018. Highend Dedicated Rootserver. (May 2018). https://definequality.net/dedicated.php

[35] Rei-Fu Huang, Hao-Yu Yang, Mango C.-T. Chao, and Shih-Chin Lin. 2012. Alternate hammering test for application-specific DRAMs and an industrial case study. In *Annual Design Automation Conference (DAC).*

[36] GitHub Inc. 2018. GitHub. (2018). https://github.com

[37] GitLab Inc. 2018. GitLab. (2018). https://gitlab.com

[38] GitLab Inc. 2018. GitLab Documentation: List SSH keys. (2018). https://docs.gitlab.com/ee/api/users.html#list-ssh-keys

[39] Mehmet S Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2016. Cache Attacks Enable Bulk Key Recovery on the Cloud. In *Cryptographic Hardware and Embedded Systems - CHES (LNCS)*, Vol. 9813. Springer, 368–388.

[40] Intel. 2015. Improving Real-Time Performance by Utilizing Cache Allocation Technology: Enhancing Performance via Allocation of the Processor's Cache. (April 2015). https://www.intel.com/content/www/us/en/communications/cache-allocation-technology-white-paper.html

[41] Intel. 2016. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide. 253665 (2016).

[42] Intel. 2016. Intel Xeon Processor E5 v4 Product Family: Datasheet Volume 2: Registers. 2 (2016).

[43] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2017. MASCAT: Stopping Microarchitectural Attacks Before Execution. Cryptology ePrint Archive, Report 2016/1196. (2017).

[44] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. 2017. SGX-Bomb: Locking Down the Processor via Rowhammer Attack. In *SysTEX.*

[45] Jedec Solid State Technology Association. 2013. Low Power Double Data Rate 3. (2013). http://www.jedec.org/standards-documents/docs/jesd209-4a

[46] JEDEC Solid State Technology Association. 2017. Low Power Double Data Rate 4. (2017). http://www.jedec.org/standards-documents/docs/jesd209-4b

[47] Suryaprasad Kareenahalli, Zohar B. Bogin, and Mihir D. Shah. 2003. Adaptive idle timer for a memory device. (2003). https://encrypted.google.com/patents/US7076617B2 US Patent Grant 2005-06-21.

[48] Dimitris Kaseridis, Jeffrey Stuecheli, and Lizy Kurian John. 2011. Minimalist open-page: A DRAM page-mode scheduling policy for the many-core era. In *International Symposium on Microarchitecture (MICRO)*.

[49] Dae-Hyun Kim, Prashant J Nair, and Moinuddin K Qureshi. 2015. Architectural support for mitigating row hammering in DRAM memories. *IEEE Computer Architecture Letters* 14, 1 (2015), 9–12.

[50] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ISCA'14*.

[51] Mark Lanteigne. 2016. How Rowhammer Could Be Used to Exploit Weaknesses in Computer Hardware. (March 2016). http://www.thirdio.com/rowhammer.pdf

[52] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium*.

[53] Yabing Liu, Will Tome, Liang Zhang, David Choffnes, Dave Levin, Bruce Maggs, Alan Mislove, Aaron Schulman, and Christo Wilson. 2015. An End-to-End Measurement of Certificate Revocation in the Web's PKI. In *IMC '15*.

[54] Sreenivas Mandava, Brian S. Morris, Suneeta Sah, Roy M. Stevens, Ted Rossin, Mathew W. Stefaniw, and John H. Crawford. 2017. Techniques for determining victim row addresses in a volatile memory. (2017). https://encrypted.google.com/patents/US9824754B2 US Patent Grant 2017-11-21.

[55] Marcin Kaczmarski. 2014. Thoughts on Intel Xeon E5-2600 v2 Product Family Performance Optimisation – component selection guidelines. (August 2014). http://infobazy.gda.pl/2014/pliki/prezentacje/d2s2e4-Kaczmarski-Optymalna.pdf Infobazy 2014.

[56] Mark Goodwin. 2015. Improving Revocation: OCSP Must-Staple and Short-lived Certificates. (2015). https://blog.mozilla.org/security/2015/11/23/improving-revocation-ocsp-must-staple-and-short-lived-certificates/

[57] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. 2015. Reverse Engineering Intel Complex Addressing Using Performance Counters. In *RAID*.

[58] Micron. 2014. DDR4 SDRAM. (2014). https://www.micron.com/~/media/documents/products/data-sheet/dram/ddr4/4gb_ddr4_sdram.pdf Retrieved on February 17, 2016.

[59] Microsoft. 2017. Cache and Memory Manager Improvements. (April 2017). https://docs.microsoft.com/en-us/windows-server/administration/performance-tuning/subsystem/cache-memory-management/improvements-in-windows-server

[60] Paul V Mockapetris. 1987. Domain names-concepts and facilities. (1987).

[61] James A Muir. 2006. Seifert's RSA fault attack: Simplified analysis and generalizations. In *International Conference on Information and Communications Security*.

[62] Onur Mutlu. 2017. The RowHammer problem and other issues we may face as memory becomes denser. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*.

[63] myLoc managed IT. 2018. The dedicated server in comparison. (May 2018). https://www.myloc.de/en/server-hosting/dedicated-server/dedicated-server-comparison.html

[64] netcup. 2018. Dedicated servers for professional applications. (May 2018). https://www.netcup.eu/professional/dedizierte-server/

[65] OpenSSL. 2015. Online Certificate Status Protocol utility. (Jan. 2015). https://www.openssl.org/docs/man1.0.2/apps/ocsp.html

[66] Paul Mutton. 2014. Certificate revocation: Why browsers remain affected by Heartbleed. (2014). https://news.netcraft.com/archives/2014/04/24/certificate-revocation-why-browsers-remain-affected-by-heartbleed.html

[67] Matthias Payer. 2016. HexPADS: a platform to detect "stealth" attacks. In *ESSoS'16*.

[68] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security Symposium*.

[69] Rui Qiao and Mark Seaborn. 2016. A New Approach for Rowhammer Attacks. In *International Symposium on Hardware Oriented Security and Trust*.

[70] Qualcomm. 2017. Snapdragon 845 Mobile Platform Product Brief. (Dec. 2017). https://www.qualcomm.com/documents/snapdragon-845-mobile-platform-product-brief

[71] Inc. Qualcomm Technologies. 2016. Qualcomm Snapdragon 600 APQ8064: Data Sheet. (2016).

[72] Inc. Qualcomm Technologies. 2016. Qualcomm Snapdragon 600E Processor APQ8064E: Recommended Memory Controller and Device Settings. (2016).

[73] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. 2016. Flip Feng Shui: Hammering a Needle in the Software Stack. In *USENIX Security Symposium*.

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

[75] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. 1977. Cryptographic communications system and method. (1977). https://patents.google.com/patent/US4405829 US Patent Grant 1983-09-20.

[76] Hemant G Rotithor, Randy B Osborne, and Nagi Aboulenein. 2006. Method and apparatus for out of order memory scheduling. (Oct. 2006). US Patent 7,127,574.

[77] Stefan Santesson, Michael Myers, Rich Ankney, Ambarish Malpani, Slava Galperin, and Dr. Carlisle Adams. 2013. X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP. RFC 6960. (2013). https://doi.org/10.17487/RFC6960

[78] Mark Seaborn and Thomas Dullien. 2015. Exploiting the DRAM rowhammer bug to gain kernel privileges. In *Black Hat Briefings*.

[79] X. Shen, F. Song, H. Meng, S. An, and Z. Zhang. 2014. RBPP: A row based DRAM page policy for the many-core era. In *IEEE ICPADS*.

[80] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. 2017. CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In *USENIX Security Symposium*.

[81] Andrei Tatar, Radhesh Krishnan, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2018. Throwhammer: Rowhammer Attacks over the Network and Defenses. In *USENIX ATC*.

[82] Chee Hak Teh, Suryaprasad Kareenahalli, and Zohar Bogin. 2006. Dynamic update adaptive idle timer. (2006). https://encrypted.google.com/patents/US7076617B2 US Patent Grant 2009-09-08.

[83] Victor van der Veen. 2016. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. (2016). http://vvdveen.com/publications/drammer.slides.pdf

[84] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. 2016. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *CCS'16*.

[85] webtropia. 2018. Dedicated Server. (May 2018). https://www.webtropia.com/en/dedicated-server/root-server-vergleich.html

[86] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. 2016. One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. In *USENIX Security Symposium*.

[87] Yuval Yarom and Katrina Falkner. 2014. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium*.

[88] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. 2015. Mapping the Intel Last-Level Cache. *Cryptology ePrint Archive, Report 2015/905* (2015), 1–12.

[89] Tatu Ylonen and Chris Lonvick. 2006. The secure shell (SSH) protocol architecture. (2006).

[90] Thomas M. Zeng. 2012. The Android ION memory allocator. (Feb. 2012). https://lwn.net/Articles/480055/

[91] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. 2016. CloudRadar: A Real-Time Side-Channel Attack Detection System in Clouds. In *RAID*.

A KERNEL ACCESSES FOR NETWORK PACKETS

Table 2 shows the results of the *funccount* script of the *perf* framework [22] for functions with udp in their name while the targeted system is flooded with UDP packets.

Table 2: Results of *funccount* on the victim machine for functions with udp in their name while the system is flooded with UDP packets.

Function	Number of calls
`__udp4_lib_lookup`	2 000 024
`__udp4_lib_rcv`	1 000 012
`udp4_gro_receive`	1 000 012
`udp4_lib_lookup_skb`	1 000 012
`udp_error`	1 000 012
`udp_get_timeouts`	1 000 013
`udp_gro_receive`	1 000 013
`udp_packet`	1 000 012
`udp_pkt_to_tuple`	1 000 012
`udp_rcv`	1 000 012
`udp_v4_early_demux`	1 000 012