CAn't Touch This: Software-only Mitigation against Rowhammer Attacks targeting Kernel Memory

Ferdinand Brasser, ***Technische Universität Darmstadt;*** **Lucas Davi,** ***University of Duisburg-Essen;*** **David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi,** ***Technische Universität Darmstadt***

https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/brasser

This paper is included in the Proceedings of the 26th USENIX Security Symposium

August 16–18, 2017 • Vancouver, BC, Canada

ISBN 978-1-931971-40-9

Open access to the Proceedings of the 26th USENIX Security Symposium is sponsored by USENIX

CAn't Touch This: Software-only Mitigation against Rowhammer Attacks targeting Kernel Memory

Ferdinand Brasser[1], Lucas Davi[2], David Gens[1], Christopher Liebchen[1], and Ahmad-Reza Sadeghi[1]

[1]Technische Universität Darmstadt, Germany
{ferdinand.brasser,david.gens,christopher.liebchen,ahmad.sadeghi}@trust.tu-darmstadt.de

[2]University of Duisburg-Essen, Germany
lucas.davi@wiwinf.uni-due.de

Abstract

Rowhammer is a hardware bug that can be exploited to implement privilege escalation and remote code execution attacks. Previous proposals on rowhammer mitigations either require hardware changes or follow heuristic-based approaches (based on CPU performance counters). To date, there exists no instant protection against rowhammer attacks on legacy systems.

In this paper, we present the design and implementation of a practical and efficient software-only defense against rowhammer attacks. Our defense, called CATT, prevents the attacker from leveraging rowhammer to corrupt kernel memory from user mode. To do so, we extend the physical memory allocator of the OS to physically isolate the memory of the kernel and user space. We implemented CATT on x86 and ARM to mitigate rowhammer-based kernel exploits. Our extensive evaluation shows that our mitigation (i) can stop available real-world rowhammer attacks, (ii) imposes virtually no runtime overhead for common user and kernel benchmarks as well as commonly used applications, and (iii) does not affect the stability of the overall system.

1 Introduction

CPU-enforced memory protection is fundamental to modern computer security: for each memory access request, the CPU verifies whether this request meets the memory access policy. However, the infamous rowhammer attack [11] undermines this access control model by exploiting a hardware fault (triggered through software) to flip targeted bits in memory. The cause for this hardware fault is due to the tremendous density increase of memory cells in modern DRAM chips, allowing electrical charge (or the change thereof) of one memory cell to affect that of an adjacent memory cell. Unfortunately, increased refresh rates of DRAM modules – as suggested by some hardware manufacturers – cannot eliminate this effect [3]. In fact, the fault appeared as a surprise to hardware manufacturers, simply because it does not appear during normal system operation, due to caches. Rowhammer attacks repetitively read (*hammer*) from the same physical memory address in very short time intervals which eventually leads to a bit flip in a physically co-located memory cell.

Rowhammer Attack Diversity. Although it might seem that single bit flips are not per-se dangerous, recent attacks demonstrate that rowhammer can be used to undermine access control policies and manipulate data in various ways. In particular, it allows for tampering with the isolation between user and kernel mode [20]. For this, a malicious user-mode application locates vulnerable memory cells and forces the operating system to fill the physical memory with page-table entries (PTEs), i.e., entries that define access policies to memory pages. Manipulating one PTE by means of a bit flip allows the malicious application to alter memory access policies, building a custom page table hierarchy, and finally assigning kernel permissions to a user-mode memory page. Rowhammer attacks have made use of specific CPU instructions to force DRAM access and avoid cache effects. However, prohibiting applications from executing these instructions, as suggested in [20], is ineffective because recent rowhammer attacks do no longer depend on special instructions [3]. As such, rowhammer has become a versatile attack technique allowing compromise of co-located virtual machines [18, 26], and enabling sophisticated control-flow hijacking attacks [6, 19, 22] without requiring memory corruption bugs [4, 7, 20]. Lastly, a recent attack, called Drammer [24], demonstrates that rowhammer is not limited to x86-based systems but also applies to mobile devices running ARM processors.

Rowhammer Mitigation. The common belief is that the rowhammer fault cannot be fixed by means of any

software update, but requires production and deployment of redesigned DRAM modules. Hence, existing legacy systems will remain vulnerable for many years, if not forever. An initial defense approach performed through a BIOS update to increase the DRAM refresh rate was unsuccessful as it only slightly increased the difficulty to conduct the attack [20]. The only other software-based mitigation of rowhammer, we are aware of, is a heuristic-based approach that relies on hardware performance counters [3]. However, it induces a worst-case overhead of 8% and suffers from false positives which impedes its deployment in practice.

Goals and Contributions. The goal of this paper is to develop the first practical software-based defense against rowhammer attacks that can instantly protect existing vulnerable legacy systems without suffering from any performance overhead and false positives. From all the presented rowhammer attacks [4, 7, 17, 18, 20, 24, 26], those which compromise the kernel memory to achieve privilege escalation are the most practical attacks and most challenging to mitigate. Other attacks can be either mitigated by disabling certain system features, or are impractical for real-world attacks: rowhammer attacks on virtual machines [18, 26] heavily depend on *memory deduplication* which is disabled in most production environments by default. Further, the browser attacks shown by Bosman et al. [4] require 15 to 225 minutes. As such, they are too slow for browser attacks in practice. Hence, we focus in this paper on practical kernel-based rowhammer attacks.

We present the design and implementation of a practical mitigation scheme, called CATT, that does not aim to prevent bit flips but rather remove the dangerous effects (i.e., exploitation) of bit flips. This is achieved by limiting bit flips to memory pages that are already in the address space of the malicious application, i.e., memory pages that are per-se untrusted. For this, we extend the operating system kernel to enforce a strong physical isolation of user and kernel space.

In detail, our main contributions are:

- We present a practical software-based defense against rowhammer. In contrast to existing solutions, our defense requires no hardware changes [11], does not deploy unreliable heuristics [3], and still allows legacy applications to execute instructions that are believed to alleviate rowhammer attacks [20].

- We propose a novel enforcement-based mechanism for operating system kernels to mitigate rowhammer attacks. Our design isolates the user and kernel space in physical memory to ensure that the attacker cannot exploit rowhammer to flip bits in kernel memory.

- We present our prototype implementation for the Linux kernel version 4.6, and demonstrate its effectiveness in mitigating all previously presented rowhammer attacks [7, 20].

- We successfully applied our Linux kernel patch for CATT to the Android version 4.4 for Google's Nexus devices. This allows us to also mitigate Drammer [24], a recent rowhammer-based privilege escalation exploit targeting ARM.

- We extensively evaluate the performance, robustness and security of our defense against rowhammer attacks to demonstrate the effectiveness and high practicality of CATT. In particular, our performance measurements indicate no computational overhead for common user and kernel benchmarks.

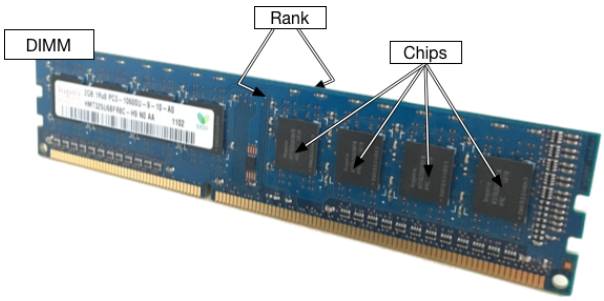

Figure 1: Organization of a DRAM module.

For a more comprehensive version of this paper with other rowhammer defense solutions, options and more technical details we refer to our full technical report available online [5].

2 Background

In this section, we provide the basic background knowledge necessary for understanding the remainder of this paper.

2.1 Dynamic Random Access Memory (DRAM)

A DRAM module, as shown in Figure 1, is structured hierarchically. The hardware module is called Dual Inline Memory Module (DIMM), which is physically connected through a channel to the memory controller. Modern desktop systems usually feature two channels facilitating parallel accesses to memory. The DIMM can be divided into one or two ranks corresponding to its front-

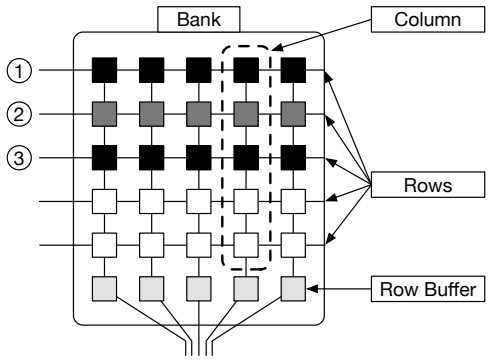

Figure 2: Organization of a Bank.

and backside. Each rank contains multiple chips which are comprised of one or multiple banks that contain the memory cells. Each bank is organized in columns and rows, as shown in Figure 2.

An individual memory cell consists of a capacitor and a transistor. To store a bit in a memory cell, the capacitor is electrically charged. By reading a bit from a memory cell, the cell is discharged, i.e., read operations are destructive. To prevent information loss, read operations also trigger a process that writes the bit back to the cell. A read operation always reads out the bits from a whole row, and the result is first saved in the row buffer before it is then transferred to the memory controller. The row buffer is also used to write back the content into the row of memory cells to restore their content.

It is noteworthy to mention that there exists the mapping between physical memory address and the rank-, bank- and row-index on the hardware module is non-linear. Consequently, two consecutive physical memory addresses can be mapped to memory cells that are located on different ranks, banks, or rows. For example, on Intel Ivy Bridge CPUs the 20th bit of the physical address determines the rank. As such, the consecutive physical addresses 0x2FFFFF and 0x300000 can be located on front and back side of the DIMM for this architecture. The knowledge of the physical memory location on the DIMM is important for both rowhammer attacks and defenses, since bit flips can only occur on the same bank. For Intel processors, the exact mapping is not officially documented, but has been reverse engineered [15, 26].

2.2 Rowhammer Overview and Challenges

As mentioned before, memory access control is an essential building block of modern computer security, e.g., to achieve process isolation, isolation of kernel code and data, and manage read-write-execute permission on memory pages. Modern systems feature a variety of mechanisms to isolate memory, e.g., paging [10], virtualization [1, 9], IOMMU [2], and special execution modes like SGX [10] and SMM [10]. However, these mechanisms enforce their isolation through hardware that mediates the physical memory accesses (in most cases the CPU). Hence, memory assigned to isolated entities can potentially be co-located in physical memory on the same bank. Since a rowhammer attack induces bit flips in co-located memory cells, it provides a subtle way to launch a remote attack to undermine memory isolation.

Recently, various rowhammer-based attacks have been presented [4, 7, 17, 18, 20, 24, 26]. Specifically, rowhammer was utilized to undermine isolation of operating system and hypervisor code, and escape from application sandboxes leveraged in web browsers. As discussed before, only the attacks that perform privilege escalation from user to kernel mode are considered as practical. In the following, we describe the challenges and workflow of rowhammer attacks. A more elaborated discussion on real-world, rowhammer-based exploits will be provided in Section 9.

The rowhammer fault allows an attacker to influence the electrical charge of individual memory cells by activating neighboring memory cells. Kim et al. [11] demonstrate that repeatedly activating two rows separated by only one row, called *aggressor rows* (① and ③ in Figure 2), lead to a bit flip in the enclosed row ②, called *victim row*.[1] To do so, the attacker has to overcome the following challenges: (i) undermine memory caches to directly perform repetitive reads on physical DRAM memory, and (ii) gain access to memory co-located to data critical to memory isolation.

Overcoming challenge (i) is complicated because modern CPUs feature different levels of memory caches which mediate read and write access to physical memory. Caches are important as processors are orders of magnitude faster than current DRAM hardware, turning memory accesses into a bottleneck for applications [25]. Usually, caches are transparent to software, but many systems feature special instructions, e.g., `clflush` or `movnti` for x86 [17, 20], to undermine the cache. Further, caches can be undermined by using certain read-access patterns that force the cache to reload data from physical memory [3]. Such patterns exist, because CPU caches are much smaller than physical memory, and system engineers have to adopt an eviction strategy to effectively utilize caches. Through alternating accesses to addresses which reside in the same cache line, the attacker can force the memory contents to be fetched from physical memory.

The attacker's second challenge (ii) is to achieve the physical memory constellation shown in Figure 2. For this, the attacker needs access to the aggressor rows in

[1] This rowhammer approach is called *double-sided* hammering. Other rowhammer techniques are discussed in Section 8

order to activate (*hammer*) them (rows ① and ③ in Figure 2). In addition, the victim row must contain data which is vulnerable to a bit flip (② in Figure 2). Both conditions cannot be enforced by the attacker. However, this memory constellation can be achieved with high probability using the following approaches. First, the attacker allocates memory hoping that the aggressor rows are contained in the allocated memory. If the operating system maps the attacker's allocated memory to the physical memory containing the aggressor rows, the attacker has satisfied the first condition. Since the attacker has no influence on the mapping between virtual memory and physical memory, she cannot directly influence this step, but she can increase her probability by repeatedly allocating large amounts of memory. Once control over the aggressor rows is achieved, the attacker releases all allocated memory except the parts which contain the aggressor rows. Next, victim data that should be manipulated has to be placed on the victim row. Again, the attacker cannot influence which data is stored in the physical memory and needs to resort to a probabilistic approach. The attacker induces the creation of many copies of the victim data with the goal that one copy of the victim data will be placed in the victim row. The attacker cannot directly verify whether the second step was successful, but can simply execute the rowhammer attack and validate whether the attack was successful. If not, the second step is repeated until the rowhammer successfully executes.

Seaborn et al. [20] successfully implemented this approach to compromise the kernel from an unprivileged user process. They gain control over the aggressor rows and then let the OS create huge amounts of page table entries with the goal of placing one page table entry in the victim row. By flipping a bit in a page table entry, they gained control over a subtree of the page tables allowing them to manipulate memory access control policies.

3 Threat Model and Assumptions

Our threat model is in line with related work [4, 7, 17, 18, 20, 26]:

- We assume that the operating system kernel is not vulnerable to software attacks. While this is hard to implement in practice it is a common assumption in the context of rowhammer attacks.
- The attacker controls a low-privileged user mode process, and hence, can execute arbitrary code but has only limited access to other system resources which are protected by the kernel through mandatory and discretionary access control.
- We assume that the system's RAM is vulnerable to rowhammer attacks. Many commonly used systems (see Table 1) include vulnerable RAM.

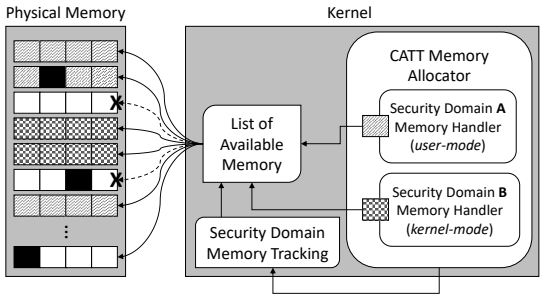

Figure 3: CATT constrains bit flips to the process' security domain.

4 Design of CATT

In this section, we present the high-level idea and design of our practical software-based defense against rowhammer attacks. Our defense, called CATT,[2] tackles the malicious *effect* of rowhammer-induced bit flips by instrumenting the operating system's memory allocator to constrain bit flips to the boundary where the attacker's malicious code executes. CATT is completely transparent to applications, and does not require any hardware changes.

Overview. The general idea of CATT is to tolerate bit flips by confining the attacker to memory that is already under her control. This is fundamentally different from all previously proposed defense approaches that aimed to prevent bit flips (cf. Section 9). In particular, CATT prevents bit flips from affecting memory belonging to higher-privileged *security domains*, e.g., the operating system kernel or co-located virtual machines. As discussed in Section 2.2, a rowhammer attack requires the adversary to bypass the CPU cache. Further, the attacker must arrange the physical memory layout such that the targeted data is stored in a row that is physically adjacent to rows that are under the control of the attacker. Hence, CATT ensures that memory between these two entities is physically separated by at least one row.[3]

To do so, CATT extends the physical memory allocator to partition the physical memory into security domains.

Figure 3 illustrates the concept. Without CATT, the attacker is able to craft a memory layout, where two ag-

[2] CAn't Touch This

[3] Kim et al. [11] mention that the rowhammer fault does not only affect memory cells of directly adjacent rows, but also memory cells of rows that are next to the adjacent row. Although we did not encounter such cases in our experiments, CATT supports multiple row separation between adversary and victim data memory. Further detailed discussion can be found in Section 8.

gressor rows enclose a victim row of a higher-privileged domain such as row ② in Figure 2. With CATT in place, the rows which are controlled by the attacker are grouped into the security domain A, whereas memory belonging to higher-privileged entities resides with their own security domain (e.g., the security domain B). Both domains are physically separated by at least one row which will not be assigned to any security domain.

Security Domains. Privilege escalation attacks are popular and pose a severe threat to modern systems. In particular, the isolation of kernel and user-mode is critical and the most appealing attack target. If a user-space application gains kernel privileges, the adversary can typically compromise the entire system. We define and maintain two security domains: a security domain for kernel memory allocations, and one security domain for user-mode memory allocations (see also Figure 3).

Challenges. The physical isolation of data raises the challenge of how to effectively isolate the memory of different system entities. To tackle this challenge, we first require knowledge of the mapping between physical addresses and memory banks. Since an attacker can only corrupt data within one bank, but not across banks, CATT only has to ensure that security domains of different system entities are isolated within each bank. However, as mentioned in Section 2.1, hardware vendors do not specify the exact mapping between physical address and banks. Fortunately, Pessl et al. [15] and Xiao et al. [26] provide a methodology to reverse engineer the mapping. For CATT, we use this methodology to discover the physical addresses of rows.

We need to ensure that the physical memory management component is aware of the isolation policy. This is vital as the memory management components have to ensure that newly allocated memory is adjacent only to memory belonging to the same security domain. To tackle this challenge, we instrumented the memory allocator to keep track of the domain association of physical memory and serve memory requests by selecting free memory from different pools depending on the security domain of the requested memory.

5 Implementation

Our software-based defense is based on modifications to low-level system software components, i.e., the physical memory allocator of the operating system kernel. In our proof-of-concept implementation of CATT, we focus on hardening Linux against rowhammer-based attacks. We successfully applied the mentioned changes to the x86-kernel version 4.6 and the Android kernel for Nexus devices in version 4.4. We chose Linux as our target OS for our proof-of-concept implementations for two reasons: (1) its source code is freely available, and (2) it is widely used on workstations and mobile devices. In the following we will explain the implementation of CATT's policy enforcement mechanism in the Linux kernel which allows for the partitioning of physical memory into isolated security domains. We note that CATT targets both x86 and ARM-based systems. Until today, rowhammer attacks have only been demonstrated for these two prominent architectures. However, our concept can be applied to other architectures, as well.

The basic idea underlying our software-based rowhammer defense is to physically separate rows that belong to different security domains. Operating systems are not per-se aware of the notions of cells and rows, but rather build memory management based on paging. Commodity operating systems use paging to map virtual addresses to physical addresses. The size of a page varies among architectures. On x86 and ARM, the page size is typically 4096 bytes (4K). As we described in Section 2.1, DRAM hardware consists of much smaller units of memory, i.e., individual memory cells storing single bits. Eight consecutive memory cells represent a byte, 4096 consecutive bytes a page frame, two to four page frames a row. Hence, our implementation of CATT changes low-level components of the kernel to make the operating system aware of the concept of memory rows.

In the following, we describe how we map individual memory pages to domains, keep track of different domains, modify the physical memory allocator, and define partitioning policies for the system's DRAM hardware.

5.1 Mapping Page Frames to Domains

To be able to decide whether two pages belong to the same security domain we need to keep track of the security domain for each page frame. Fortunately, the kernel already maintains meta data about each individual page frame. More specifically, each individual page frame is associated with exactly one meta data object (`struct page`). The kernel keeps a large array of these objects in memory. Although these objects describe physical pages, this array is referred to as *virtual memory map*, or `vmemmap`. The Page Frame Number (PFN) of a physical page is used as an offset into this array to determine the corresponding `struct page` object. To be able to associate a page frame with a security domain, we extend the definition of `struct page` to include a field that encodes the security domain. Since our prototype implementation targets rowhammer attacks that aim at violating the separation of kernel and user-space, we encode security domain 0 for kernel-space, and 1 for user-space.

5.2 Tracking Security Domains

The extension of the page frame meta data objects enables us to assign pages to security domains. However, this assignment is dynamic and changes over time. In particular, a page frame may be requested, allocated, and used by one domain, after it has been freed by another domain. Note that this does not violate our security guarantees, but is necessary for the system to manage physical memory dynamically. Yet, we need to ensure that page frames being reallocated continue to obey our security policy. Therefore, we reset the security domain upon freeing a page.

Upon memory allocation, CATT needs to correctly set the security domain of the new page. To do so, we require information about the requesting domain. For our case, where we aim at separating kernel and user-space domains, CATT utilizes the call site information, which is propagated to the memory allocator by default. Specifically, each allocation request passes a set of flags to the page allocator. These flags encode whether an allocation is intended for the kernel or the user-space. We leverage this information and separate the two domains by setting the domain field of the respective page frame.

When processes request memory, the kernel initially only creates a virtual mapping without providing actual physical page frames for the process. Instead, it only assigns physical memory on demand, i.e., when the requesting process accesses the virtual mapping a page fault is triggered. Thereafter, the kernel invokes the physical page allocator to search for usable pages and installs them under the virtual address the process attempted to access. We modified the page fault handler, which initiates the allocation of a new page, to pass information about the security domain to the page allocator. Next, the page is allocated according to our policy and sets the domain field of the page frame's meta data object to the security domain of the interrupted process.

5.3 Modifying the Physical Page Allocator

The Linux kernel uses different levels of abstraction for different memory allocation tasks. The physical page allocator, called *zoned buddy allocator*, is the main low-level facility handling physical page allocations. It exports its interfaces through functions such as `alloc_pages` which can be used by other kernel components to request physical pages. In contrast to higher-level allocators, the buddy allocator only allows for allocating sets of memory pages with a cardinality which can be expressed as a power of two (this is referred to as the *order* of the allocation). Hence, the buddy allocator's smallest level of granularity is a single memory page. The buddy allocator already partitions the system RAM into different *zones*. We modify the implementation of the physical page allocator in the kernel to include a dedicated memory zone for the kernel. This enables CATT to separate kernel from user pages according to the security domain of the origin of the allocation request. Hence, any requests for kernel pages will be served from the dedicated memory zone. We additionally instrument a range of maintenance checks to make them aware of our partitioning policy before the allocator returns a physical page. If any of these checks fail, the page allocator is not allowed to return the page in question.

5.4 Defining DRAM Partitioning Policies

Separating and isolating different security domains is essential to our proposed mitigation. For this reason, we incorporate detailed knowledge about the platform and its DRAM hardware configuration into our policy implementation. While our policy implementation for a target system largely depends on its architecture and memory configuration, this does not represent a fundamental limitation. Indeed, independent research [15, 26] has provided the architectural details for the most prevalent architectures, i.e., it shows that the physical address to DRAM mapping can be reverse engineered automatically for undocumented architectures. Hence, it is possible to develop similar policy implementations for architectures and memory configurations beyond x86 and ARM. We build on this prior research and leverage the physical address to DRAM mapping information to enforce strict physical isolation. In the following, we describe our implementation of the partitioning strategy for isolating kernel and user-space.

Kernel-User Isolation. To achieve physical separation of user and kernel space we adopt the following strategy: we divide each bank into a top and a bottom part, with a separating row in-between. Page frames for one domain are exclusively allocated from the part that was assigned to that domain. The part belonging to the kernel domain is determined by the physical location of the kernel image.[4] As a result, user and kernel space allocations may be co-located within one bank, but never within adjacent rows.[5] Due to this design memory allocated to the kernel during early boot is allocated from a memory region which is part of the kernel's security domain, hence, the isolation covers *all* kernel memory. Different partitioning policies would be possible in theory: for instance, we could confine the kernel to a certain DRAM

[4] This is usually at 1MB, although Kernel Address Space Layout Randomization (KASLR) may slightly modify this address according to a limited offset.

[5] The exact location for the split can be chosen at compile time. Hence, the partitioning is not fixed but can be chosen arbitrarily (e.g., 20-80, 50-50, 75-25, etc.).

System	Operating System	System Model
S1	Ubuntu 14.04.4 LTS	Dell OptiPlex 7010
S2	Debian 8.2	Dell OptiPlex 990
S3	Kali Linux 2.0	Lenovo ThinkPad x220

Table 1: Model numbers of the vulnerable systems used for our evaluation.

bank to avoid co-location of user domains within a single bank. However, this would likely result in a severe increase of memory latency, since reads and writes to a specific memory bank are served by the bank's row buffer. The benefit of our partitioning policy stems from the fact that we distribute memory belonging to the kernel security domain over multiple banks thereby not negatively impacting performance. Additionally, the bank split between top and bottom could be handled at run time, e.g., by dynamically keeping track of the individual bank-split locations similar to the watermark handling already implemented for different zones in the buddy allocator. In our current prototype, we only need to calculate the row index of a page frame for each allocation request. More specifically, we calculate this index from the physical address (PA) in the following way:

$$Row(PA) := \frac{PA}{PageSize \cdot PagesPerDIMM \cdot DIMMs}$$

Here, we calculate the number of pages per DIMM as $PagesPerDIMM := PagesPerRow \cdot BanksPerRank \cdot RanksPerDIMM$. Because all possible row indices are present once per bank, this equation determines the row index of the given physical address.[6] We note that this computation is in line with the available rowhammer exploits [20] and the reported physical to DRAM mapping recently reverse engineered [15, 26]. Since the row size is the same for all Intel architectures prior to Skylake [7], our implementation for this policy is applicable to a wide range of system setups, and can be adjusted without introducing major changes to fit other configurations as well.

6 Security Evaluation

The main goal of our software-based defense is to protect legacy systems from rowhammer attacks. We tested the effectiveness of CATT on diverse hardware configurations. Among these, we identified three hardware configurations, where we observed many reproducible bit flips. Table 1 and Table 2 lists the exact configurations of the three platforms we use for our evaluation. Our effectiveness evaluation of CATT is based on two attack scenarios. For the first scenario, we systematically search for reproducible bit flips based on a tool published by Gruss et al.[7] Our second attack scenario leverages a real-world rowhammer exploit published by Google's Project Zero.[8] We compared the outcome of both attacks on our vulnerable systems before and after applying CATT. Next, we elaborate on the two attack scenarios and their mitigation in more detail.

6.1 Rowhammer Testing Tool

We use a slightly modified version of the double-sided rowhammering tool, which is based on the original test by Google's Project Zero [20]. Specifically, we extended the tool to also report the aggressor physical addresses, and adjusted the default size of the fraction of physical memory that is allocated for the test. The tool scans the allocated memory for memory cells that are vulnerable to the rowhammer attack. To provide comprehensive results, the tool needs to scan the entire memory of the system. However, investigating the entire memory is hard to achieve in practice since some parts of memory are always allocated by other system components. These parts are therefore not available to the testing tool, i.e., memory reserved by operating system. To achieve maximum coverage, the tool allocates a huge fraction of the available memory areas. However, due to the lazy allocation of Linux the allocated memory is initially not mapped to physical memory. Hence, each mapped virtual page is accessed at least once, to ensure that the kernel assigns physical pages. Because user space only has access to the virtual addresses of these mappings, the tool exploits the `/proc/pagemap` kernel interface to retrieve the physical addresses. As a result, most of the systems physical memory is allocated to the rowhammering tool.

Afterwards, the tool analyzes the memory in order to identify potential victim and aggressor pages in the physical memory. As the test uses the double-sided rowhammering approach two aggressor pages must be identified for every potential victim page. Next, all potential victim pages are challenged for vulnerable bit flips. For this, the potential victim page is initialized with a fixed bit pattern and "hammered" by accessing and flushing the two associated aggressor pages. This ensures that all of the

[6] The default values for DDR3 on x86 are 4K for the page size, 2 pages per row, 8 banks per rank, 2 ranks per DIMM and between 1 and 4 DIMMs per machine. For DDR4 the number of banks per rank was doubled. DDR4 is supported on x86 starting with Intel's Skylake and AMD's Zen architecture.

[7] `https://github.com/IAIK/rowhammerjs/tree/master/native`

[8] `https://bugs.chromium.org/p/project-zero/issues/detail?id=283`

	CPU			RAM			
System	Version	Cores	Speed	Size	Speed	Manufacturer	Part number
S1	i5-3570	4	3.40GHz	2x2GB	1333 MHz	Hynix Hyundai	HMT325U6BFR8C-H9
				1x4GB	1333 MHz	Corsair	CMV4GX3M1A1600C11
S2	i7-2600	4	3.4GHz	2x4GB	1333 MHz	Samsung	M378B5273DH0-CH9
S3	i5-2520M	4	2.5GHz	2x4GB	1333 MHz	Samsung	M471B5273DH0-CH9

Table 2: Technical specifications of the vulnerable systems used for our evaluation.

	Rowhammer Exploit: Success (avg. # of tries)	
	Vanilla System	CATT
S1	✓(11)	✗(3821)
S2	✓(42)	✗(3096)
S3	✓(53)	✗(3768)

Table 3: Results of our security evaluation. We found that CATT mitigates rowhammer attacks. We executed the rowhammer test on each system three times and averaged the amount of bit flips.

accesses activate a row in the respective DRAM module. This process is repeated 10^6 times.[9] Lastly, the potential victim address can be checked for bit flips by comparing its memory content with the fixed pattern bit. The test outputs a list of addresses for which bit flips have been observed, i.e., a list of victim addresses.

Preliminary Tests for Vulnerable Systems. Using the rowhammering testing tool we evaluated our target systems. In particular, we were interested in systems that yield reproducible bit flips, as only those are relevant for practical rowhammer attacks. This is because the attack requires two steps. First, the attacker needs to allocate chunks of memory, and test each chunk to identify vulnerable memory. Second, the attacker needs to exploit the vulnerable memory. Since the attacker cannot force the system to allocate page tables at a certain physical position in RAM, the attacker has to repeatedly spray the memory with page tables to increase the chances of hitting the desired memory location. Both steps relay on reproducible bit flips.

Hence, we configured the rowhammering tool to only report memory addresses where bit flips can be triggered repeatedly. We successively confirmed that this list indeed yields reliable bit flips by individually triggering the reported addresses and checking for bit flips within an interval of 10 seconds. Additionally, we tested the bit flips across reboots through random sampling.

The three systems mentioned in Table 1 and Table 2 are highly susceptible to reproducible bit flips. Executing the rowhammer test on these three times and rebooting the system after each test run, we found 133 pages with exploitable bit flips for S1, 31 pages for S2, and 23 pages for S3.

To install CATT, we patched the Linux kernel of each system to use our modified memory allocator. Recall that CATT does not aim to prevent bit flips but rather constrain them to a security domain. Hence, executing the rowhammer test on CATT-hardened systems still locates vulnerable pages. However, in the following, we demonstrate based on a real-world exploit that the vulnerable pages are not exploitable.

6.2 Real-world Rowhammer Exploit

To further demonstrate the effectiveness of our mitigation, we tested CATT against a real-world rowhammer exploit. The goal of the exploit is to escalate the privileges of the attacker to kernel privileges (i.e., gain root access). To do so, the exploit leverages rowhammer to manipulate the page tables. Specifically, it aims to manipulate the access permission bits for kernel memory, i.e., reconfigure its access permission policy. A second option is to manipulate page table entries in such a way that they point to attacker controlled memory thereby allowing the attacker to install new arbitrary memory mappings.[10]

To launch the exploit, two conditions need to be satisfied: (1) a page table entry must be present in a vulnerable row, and (2) the enclosing aggressor pages must be allocated in attacker-controlled memory.

Since both conditions are not directly controllable by the attacker, the attack proceeds as follows: the attacker allocates large memory areas. As a result, the operating system needs to create large page tables to maintain the newly allocated memory. This in turn increases the probability to satisfy the aforementioned conditions, i.e., a page table entry will eventually be allocated to a victim

[9] This value is the hardcoded default value. Prior research [11, 12] reported similar numbers.

[10] The details of this attack option are described by Seaborn et al. [20].

page. Due to vast allocation of memory, the attacker also increases her chances that aggressor pages are co-located to the victim page.

Once the preconditions are satisfied, the attacker launches the rowhammer attack to induce a bit flip in victim page. Specifically, the bit flip modifies the page table entry such that a subtree of the paging hierarchy is under the attacker's control. Lastly, the attacker modifies the kernel structure that holds the attacker-controlled user process privileges to elevate her privileges to the superuser root. Since the exploit is probabilistic, it only succeeds in five out of hundred runs (5%). Nevertheless, a single successful run allows the attacker to compromise the entire system.

Effectiveness of CATT. Our defense mechanism does not prevent the occurrence of bit flips on a system. Hence, we have to verify that bit flips cannot affect data of another security domain. Rowhammer exploits rely on the fact that such a cross domain bit flip is possible, i.e., in the case of our exploit it aims to induce a bit flip in the kernel's page table entries.

However, since the exploit by itself is probabilistic, an unsuccessful attempt does not imply the effectiveness of CATT. As described above, the success rate of the attack is about 5%. After deploying CATT on our test systems, we repeatedly executed the exploit to minimize the probability of the exploit failing due to the random memory layout rather than due to our protection mechanism. We automated the process of continuously executing the exploit and ran this test for 48 h, on all three test systems. In this time frame the exploit made on average 3500 attempts of which on average 175 should have succeeded. However, with CATT, none of the attempts was successful. Hence, as expected, CATT effectively prevents rowhammer-based exploits.

As we have demonstrated, CATT successfully prevents the original attack developed on x86 by physically isolating pages belonging to the kernel from the user-space domain. In addition to that, the authors of the Drammer exploit [24] confirm that CATT prevents their exploit on ARM. The reason is, that they follow the same strategy as in the original kernel exploit developed by Project Zero, i.e., corrupting page table entries in the kernel from neighboring pages in user space. Hence, CATT effectively prevents rowhammer exploits on ARM-based mobile platforms as well.

7 Performance Evaluation

One of our main goals is practicability, i.e., inducing negligible performance overhead. To demonstrate practicability of our defense, we thoroughly evaluated the performance and stability impact of CATT on different benchmark and testing suites. In particular, we used the SPEC CPU2006 benchmark suite [8] to measure the impact on CPU-intensive applications, LMBench3 [14] for measuring the overhead of system operations, and the Phoronix test suite [16] to measure the overhead for common applications. We use the Linux Test Project, which aims at stress testing the Linux kernel, to evaluate the stability of our test system after deploying CATT. We performed all of our performance evaluation on system S2 (cf. Table 2).

7.1 Run-time Overhead

Table 4 summarizes the results of our performance benchmarks. In general, the SPEC CPU2006 benchmarks measure the impact of system modifications on CPU intensive applications. Since our mitigation mainly affects the physical memory management, we did not expect a major impact on these benchmarks. However, since these benchmarks are widely used and well established we included them in our evaluation. In fact, we observe a minimal performance improvement for CATT by 0.49% which we attribute to measuring inaccuracy. Such results have been reported before when executing a set of benchmarks for the same system with the exact same configuration and settings. Hence, we conclude that CATT does not incur any performance penalty.

LMBench3 is comprised of a number of micro benchmarks which target very specific performance parameters, e.g., memory latency. For our evaluation, we focused on micro benchmarks that are related to memory performance and excluded networking benchmarks. Similar to the previous benchmarks, the results fluctuate on average between -0.4% and 0.11%. Hence, we conclude that our mitigation has no measurable impact on specific memory operations.

Finally, we tested the impact of our modifications on the Phoronix benchmarks. In particular, we selected a subset of benchmarks[11] that, on one hand, aim to measure memory performance (IOZone and Stream), and, on the other hand, test the performance of common server applications which usually rely on good memory performance.

To summarize, our rigorous performance evaluation with the help of different benchmarking suites did not yield any measurable overhead. This makes CATT a highly practical mitigation against rowhammer attacks.

[11] The Phoronix benchmarking suite features a large number of tests which cover different aspects of a system. By selecting a subset of the available tests we do not intend to improve our performance evaluation. On the contrary, we choose a subset of tests that is likely to yield measurable performance overhead, and excluded tests which are unrelated to our modification, e.g., GPU or machine learning benchmarks.

SPEC2006	CATT
perlbench	0.29%
bzip2	0.00%
gcc	-0.71%
mcf	-1.12%
gobmk	0.00%
hmmer	0.23%
sjeng	0.19%
libquantum	-1.63%
h264ref	0.00%
omnetpp	-0.28%
astar	-0.45%
xalan	-0.14%
milc	-1.79%
namd	-1.82%
dealII	0.00%
soplex	0.00%
povray	-0.46%
lbm	-1.12%
sphinx3	-0.58%
Mean	-0.49%

Phoronix	CATT
IOZone	0.05%
Unpack Kernel	-0.50%
PostMark	0.92%
7-Zip	1.18%
OpenSSL	-0.22%
PyBench	-0.59%
Apache	-0.21%
PHPBench	0.35%
stream	1.96%
ramspeed	0.00%
cachebench	0.05%
Mean	0.27%

LMBench3	CATT
Context Switching:	
2p/0K	-2.44%
2p/16K	0.00%
2p/64K	2.00%
8p/16K	-1.73%
8p/64K	0.00%
16p/16K	-1.33%
16p/64K	0.99%
Mean	-0,36%
File & VM Latency:	
0K File Create	0.27%
0K File Delete	0.89%
10K File Create	-0.35%
10K File Delete	0.47%
Mmap Latency	-1.81%
Mean	-0,12%

LMBench3	CATT
Local Bandwidth:	
Pipe	0.18%
AF UNIX	-0.30%
File Reread	-0.38%
Mmap reread	0.00%
Bcopy (libc)	0.08%
Bcopy (hand)	0.34%
Mem read	0.00%
Mem write	0.43%
Mean	0.04%
Memory Latency:	
L1 $	0.00%
L2 $	0.00%
Main mem	-2.09%
Rand mem	1.66%
Mean	0.11%

Table 4: The benchmarking results for SPEC CPU2006, Phoronix, and LMBench3 indicate that CATT induce no measurable performance overhead. In some cases we observed negative overhead, hence, performance improvements. However, we attribute such results to measuring inaccuracy.

7.2 Memory Overhead

CATT prevents the operating system from allocating certain physical memory.

The memory overhead of CATT is constant and depends solely on number of memory rows per bank. Per bank, CATT omits one row to provide isolation between the security domains. Hence, the memory overhead is $1/\#rows$ (*#rows* being rows per bank). While the number of rows per bank is dependent on the system architecture, it is commonly in the order of 2^{15} rows per bank, i.e., the overhead is $2^{-15} \triangleq 0,003\%$.[12]

7.3 Robustness

Our mitigation restricts the operating system's access to the physical memory. To ensure that this has no effect on the overall stability, we performed numerous stress tests with the help of the Linux Test Project (LTP) [13]. These tests are designed to stress the operating system to identify problems. We first run these tests on a vanilla Debian 8.2 installation to receive a baseline for the evaluation of CATT. We summarize our results in Table 5, and report no deviations for our mitigation compared to the baseline. Further, we also did not encounter any problems during the execution of the other benchmarks. Thus, we conclude that CATT does not affect the stability of the protected system.

[12] https://lackingrhoticity.blogspot.de/2015/05/how-physical-addresses-map-to-rows-and-banks.html

8 Discussion

Our prototype implementation targets Linux-based systems. Linux is open-source allowing us to implement our defense. Further, all publicly available rowhammer attacks target this operating system. CATT can be easily ported to memory allocators deployed in other operating systems. In this section, we discuss in detail the generality of our software-based defense against rowhammer. For a detailed discussion of possible extensions and additional policies we refer to our technical report [5].

Linux Test Project	Vanilla	CATT
clone	✓	✓
ftruncate	✓	✓
prctl	✓	✓
ptrace	✓	✓
rename	✓	✓
sched_prio_max	✓	✓
sched_prio_min	✓	✓
mmstress	✓	✓
shmt	✗	✗
vhangup	✗	✗
ioctl	✗	✗

Table 5: Result for individual stress tests from the Linux Test Project.

8.1 Applying CATT to Mobile Systems

The rowhammer attack is not limited to x86-based systems, but has been recently shown to also affect the ARM platform [24]. The ARM architecture is predominant in mobile systems, and used in many smartphones and tablets. As CATT is not dependent on any x86 specific properties, it can be easily adapted for ARM based systems. We demonstrate this by applying our extended physical memory allocator to the Android kernel for Nexus devices in version 4.4. Since there are no major deviations in the implementation of the physical page allocator of the kernel between Android and stock Linux kernel, we did not encounter any obstacles during the port.

8.2 Single-sided Rowhammer Attacks

From our detailed description in Section 4 one can easily follow that our proposed solution can defeat all known rowhammer-based privilege escalation attacks in general, and single-sided rowhammer attacks [24] in particular. In contrast to double-sided rowhammer attacks (see Figure 2), single-sided rowhammer attacks relax the adversary's capabilities by requiring that the attacker has control over only one row adjacent to the victim memory row. As described in more detail in Section 4, CATT isolates different security domains in the physical memory. In particular, it ensures that different security domains are separated by at least one buffer row that is never used by the system. This means that the single-sided rowhammer adversary can only flip bits in own memory (that it already controls), or flip bits in buffer rows.

8.3 Benchmarks Selection

We selected our benchmarks to be comparable to the related literature. Moreover, we have done evaluations that go beyond those in the existing work to provide additional insight. Hereby, we considered different evaluation aspects: We executed SPEC CPU2006 to verify that our changes to the operating system impose no overhead of user-mode applications. Further, SPEC CPU2006 is the most common benchmark in the field of memory-corruption defenses, hence, our solutions can be compared to the related work. LMBench3 is specifically designed to evaluate the performance of common system operations, and used by the Linux kernel developers to test whether changes to the kernel affect the performance. As such LMBench3 includes many tests. For our evaluation, we included those benchmarks that perform memory operations and are relevant for our defense. Finally, we selected a number of common applications from the Phoronix test suite as macro benchmarks, as well as the *pts/memory* tests which are designed to measure the RAM and cache performance. For all our benchmarks, we did not observe any measurable overhead (see Table 4).

8.4 Vicinity-less Rowhammering

All previous Rowhammer attacks exploit rows which are physically co-located [4, 7, 20, 24]. However, while Kim et al. [11] suggested that physical adjacency accounts for the majority of possible bit flips, they also noted that this was not always the case. More specifically, they attributed potential aggressor rows with a greater row distance to the *re-mapping* of faulty rows: DRAM manufacturers typically equip their modules with around 2% of spare rows, which can be used to physically replace failing rows by re-mapping them to a spare row [23]. This means, that physically adjacent spare rows can be assigned to arbitrary row indices, potentially undermining our isolation policy. For this, an adversary requires a way of determining pairs of defunct rows, which are re-mapped to physically adjacent spare rows. We note that such a methodology can also be used to adjust our policy implementation, e.g., by disallowing any spare rows to be assigned to kernel allocations. Hence, re-mapping of rows does not affect the security guarantees provided by CATT.

9 Related Work

In this section, we provide an overview of existing rowhammer attack techniques, their evolution, and proposed defenses. Thereafter, we discuss the shortcomings

of existing work on mitigating rowhammer attacks and compare them to our software-based defense.

9.1 Rowhammer Attacks

Kim et al. [11] were the first to conduct experiments and analyze the effect of bit flipping due to repeated memory reads. They found that this vulnerability can be exploited on Intel and AMD-based systems. Their results show that over 85% of the analyzed DRAM modules are vulnerable. The authors highlight the impact on memory isolation, but they do not provide any practical attack. Seaborn and Dullien [20] published the first practical rowhammer-based privilege-escalation attacks using the x86 `clflush` instruction. In their first attack, they use rowhammer to escape the Native Client (NaCl) [27] sandbox. NaCl aims to safely execute native applications by 3rd-party developers in the browser. Using rowhammer malicious developers can escape the sandbox, and achieve remote code execution on the target system. With their second attack, Seaborn and Dullien utilize rowhammer to compromise the kernel from an unprivileged user-mode application. Combined with the first attack, the attacker can remotely compromise the kernel without exploiting any software vulnerabilities. To compromise the kernel, the attacker first fills the physical memory with page-table entries by allocating a large amount of memory. Next, the attacker uses rowhammer to flip a bit in memory. Since the physical memory is filled with page-table entries, there is a high probability that an individual page-table entry is modified by the bit flip in a way that enables the attacker to access other page-table entries, modify arbitrary (kernel) memory, and eventually completely compromise the system. Qiao and Seaborn [17] implemented a rowhammer attack with the x86 `movnti` instruction. Since the `memcpy` function of `libc` – which is linked to nearly all C programs – utilizes the `movnti` instruction, the attacker can exploit the rowhammer bug with code-reuse attack techniques [21]. Hence, the attacker is not required to inject her own code but can reuse existing code to conduct the attack. Aweke et al. [3] showed how to execute the rowhammer attack without using any special instruction (e.g., `clflush` and `movnti`). The authors use a specific memory-access pattern that forces the CPU to evict certain cache sets in a fast and reliable way. They also concluded that a higher refresh rate for the memory would not stop rowhammer attacks. Gruss et al. [7] demonstrated that rowhammer can be launched from JavaScript. Specifically, they were able to launch an attack against the page tables in a recent Firefox version. Similar to Seaborn and Dullien's exploit this attack is mitigated by CATT. Later, Bosman et al. [4] extended this work by exploiting the memory deduplication feature of Windows 10 to create counterfeit JavaScript objects, and corrupting these objects through rowhammer to gain arbitrary read/write access within the browser. In their follow-up work, Razavi et al. [18] applied the same attack technique to compromise cryptographic (private) keys of co-located virtual machines. Concurrently, Xiao et al. [26] presented another cross virtual machine attack where they use rowhammer to manipulate page-table entries of Xen. Further, they presented a methodology to automatically reverse engineer the relationship between physical addresses and rows and banks. Independently, Pessl et al. [15] also presented a methodology to reverse engineer this relationship. Based on their findings, they demonstrated cross-CPU rowhammer attacks, and practical attacks on DDR4. Van der Veen et al. [24] recently demonstrated how to adapt the rowhammer exploit to escalate privileges in Android on smartphones. Since the authors use the same exploitation strategy of Seaborn and Dullien, CATT can successfully prevent this privilege escalation attack. While the authors conclude that it is challenging to mitigate rowhammer in software, we present a viable implementation that can mitigate practical user-land privilege escalation rowhammer attacks.

Note that all these attacks require memory belonging to a higher-privileged domain (e.g., kernel) to be physically co-located to memory that is under the attacker's control. Since our defense prevents direct co-location, we mitigate these rowhammer attacks.

9.2 Defenses against Rowhammer

Kim et al. [11] present a number of possible mitigation strategies. Most of their solutions involve changes to the hardware, i.e., improved chips, refreshing rows more frequently, or error-correcting code memory. However, these solutions are not very practical: the production of improved chips requires an improved design, and a new manufacturing process which would be costly, and hence, is unlikely to be implemented. The idea behind refreshing the rows more frequently (every 32ms instead of 64ms) is that the attacker needs to hammer rows many times to destabilize an adjacent memory cell which eventually causes the bit flip. Hence, refreshing (stabilizing) rows more frequently could prevent attacks because the attacker would not have enough time to destabilize individual memory cells. Nevertheless, Aweke et al. [3] were able to conduct a rowhammer attack within 32ms. Therefore, a higher refresh rate alone cannot be considered as an effective countermeasure against rowhammer. Error-correcting code (ECC) memory is able to detect and correct single-bit errors. As observed by Kim et al. [11] rowhammer can induce multiple bit flips which cannot be detected by ECC memory. Further, ECC memory has an additional space overhead of around 12% and is more

expensive than usual DRAM, therefore it is rarely used.

Kim et al. [11] suggest to use probabilistic adjacent row activation (PARA) to mitigate rowhammer attacks. As the name suggests, reading from a row will trigger an activation of adjacent rows with a low probability. During the attack, the malicious rows are activated many times. Hence, with high probability the victim row gets refreshed (stabilized) during the attack. The main advantage of this approach is its low performance overhead. However, it requires changes to the memory controller. Thus, PARA is not suited to protect legacy systems.

To the best of our knowledge Aweke et al. [3] proposed the only other software-based mitigation against rowhammer. Their mitigation, coined ANVIL, uses performance counters to detect high cache-eviction rates which serves as an indicator of rowhammer attacks [3]. However, this defense strategy has three disadvantages: (1) it requires the CPU to feature performance counters. In contrast, our defense does not rely on any special hardware features. (2) ANVIL's worst-case runtime overhead for SPEC CPU2006 is 8%, whereas our worst-case overhead is 0.29% (see Table 4). (3) ANVIL is a heuristic-based approach. Hence, it naturally suffers from false positives (although the FP rate is below 1% on average). In contrast, we provide a deterministic approach that is guaranteed to stop rowhammer-based kernel-privilege escalation attacks.

10 Conclusion

Rowhammer is a hardware fault, triggered by software, allowing the attacker to flip bits in physical memory and undermine CPU-enforced memory access control. Recently, researchers have demonstrated the power and consequences of rowhammer attacks by breaking the isolation between virtual machines, user and kernel mode, and even enabling traditional memory-corruption attacks in the browser. In particular, rowhammer attacks that undermine the separation of user and kernel mode are highly practical and critical for end-user systems and devices.

Contrary to the common belief that rowhammer requires hardware changes, we show the first defense strategy that is purely based on software. CATT is a practical mitigation that tolerates rowhammer attacks by dividing the physical memory into security domains, and limiting rowhammer-induced bit flips to the attacker-controlled security domain. To this end, we implemented a modified memory allocator that strictly separates memory rows of user and kernel mode. Our detailed evaluation of CATT demonstrates that our defense mechanism prevents all known rowhammer-based kernel privilege escalation attacks while neither affecting the run-time performance nor the stability of the system.

Acknowledgment

The authors thank Simon Schmitt for sacrificing his personal laptop to the cause of science, and Victor van der Veen, Daniel Gruss and Kevin Borgolte for their feedback.

This work was supported in part by the German Science Foundation (project P3, CRC 1119 CROSSING), the European Union's Horizon 2020 Research and Innovation Programme under grant agreement No. 643964 (SUPERCLOUD), the Intel Collaborative Research Institute for Secure Computing (ICRI-SC), and the German Federal Ministry of Education and Research within CRISP.

References

[1] AMD. Intel 64 and IA-32 architectures software developer's manual - Chapter 15 Secure Virtual Machine nested paging. http://developer.amd.com/resources/documentation-articles/developer-guides-manuals, 2012.

[2] I. AMD. I/O virtualization technology (IOMMU) specification. *AMD Pub*, 34434, 2007.

[3] 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. In *21st International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2016.

[4] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida. Dedup est machina: Memory deduplication as an advanced exploitation vector. In *37th IEEE Symposium on Security and Privacy*, S&P, 2016.

[5] F. Brasser, L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi. CAn't Touch This: Practical and Generic Software-only Defenses Against Rowhammer Attacks. https://arxiv.org/abs/1611.08396, 2016. arXiv:1611.08396 [cs.CR].

[6] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, M. Negro, M. Qunaibit, and A.-R. Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2015.

[7] D. Gruss, C. Maurice, and S. Mangard. Rowhammer.js: A cache attack to induce hardware faults from a website. In *13th Conference on Detection of Intrusions and Malware and Vulnerability Assessment*, DIMVA, 2016.

[8] J. L. Henning. SPEC CPU2006 memory footprint. *SIGARCH Computer Architecture News*, 35, 2007.

[9] Intel. Intel 64 and IA-32 architectures software developer's manual - Chapter 28 VMX support for address translation. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf.

[10] Intel. Intel 64 and IA-32 architectures software developer's manual. http://www-ssl.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html, 2015.

[11] 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 *41st Annual International Symposium on Computer Architecture*, ISCA, 2014.

[12] M. Lanteigne. How rowhammer could be used to exploit weaknesses in computer hardware. https://www.thirdio.com/rowhammer.pdf, 2016.

[13] LTP developer. The linux test project. https://linux-test-project.github.io/, 2016.

[14] L. McVoy and C. Staelin. Lmbench: Portable tools for performance analysis. In *USENIX Technical Conference*, ATEC, 1996.

[15] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard. Drama: Exploiting dram addressing for cross-cpu attacks. In *25th USENIX Security Symposium*, USENIX Sec, 2016.

[16] Phoronix. Phoronix test suite. http://www.phoronix-test-suite.com/, 2016.

[17] R. Qiao and M. Seaborn. A new approach for rowhammer attacks. In *IEEE International Symposium on Hardware Oriented Security and Trust*, HOST, 2016.

[18] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos. Flip feng shui: Hammering a needle in the software stack. In *25th USENIX Security Symposium*, USENIX Sec, 2016.

[19] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *36th IEEE Symposium on Security and Privacy*, S&P, 2015.

[20] M. Seaborn and T. Dullien. Exploiting the dram rowhammer bug to gain kernel privileges. https://googleprojectzero.blogspot.de/2015/03/exploiting-dram-rowhammer-bug-to-gain.html, 2016.

[21] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2007.

[22] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *34th IEEE Symposium on Security and Privacy*, S&P, 2013.

[23] A. J. Van De Goor and I. Schanstra. Address and data scrambling: Causes and impact on memory tests. In *The First IEEE International Workshop on Electronic Design, Test and Applications*, DELTA, 2002.

[24] 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 commodity mobile platforms. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2016.

[25] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.

[26] Y. Xiao, X. Zhang, Y. Zhang, and M.-R. Teodorescu. One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation. In *25th USENIX Security Symposium*, USENIX Sec, 2016.

[27] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *30th IEEE Symposium on Security and Privacy*, S&P, 2009.