RIP-RH: Preventing Rowhammer-based Inter-Process Attacks

Carsten Bock
Technische Universität Darmstadt
carsten.bock@stud.tu-darmstadt.de

Ferdinand Brasser
Technische Universität Darmstadt
f.brasser@trust.tu-darmstadt.de

David Gens
Technische Universität Darmstadt
d.gens@trust.tu-darmstadt.de

Christopher Liebchen
Technische Universität Darmstadt
c.liebchen@trust.tu-darmstadt.de

Ahmad-Reza Sadeghi
Technische Universität Darmstadt
a.sadeghi@trust.tu-darmstadt.de

ABSTRACT

Run-time attacks pose a continuous threat to the security of computer systems. These attacks aim at hijacking the operation of a computer program by subverting its execution at run time. While conventional run-time attacks usually require memory-corruption vulnerabilities in the program, hardware bugs represent an increasingly popular attack vector. Rowhammer represents a vulnerability in the design of DRAM modules that allows an adversary to modify memory locations in physical proximity to attacker-controlled memory on the module without accessing them. This is a serious threat to real-world systems, since DRAM is used as main memory on virtually all platforms.

Recent research proposed defenses against rowhammer, such by patching the memory controller in hardware, or statically partitioning physical memory to protect the operating system kernel from a user space adversary. However, sharing DRAM memory securely between a number of different entities currently remains as an open problem. In this paper, we present RIP-RH, a DRAM-aware memory allocator that allows for dynamic management of multiple user-space processes. RIP-RH ensures that the memory partitions belonging to individual processes are physically isolated. In our detailed evaluation we demonstrate that our prototype implementation of RIP-RH incurs a modest run-time overhead of 3.17% for standard benchmarks and offers practical performance in a number of real-world scenarios.

ACM Reference Format:
Carsten Bock, Ferdinand Brasser, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2019. RIP-RH: Preventing Rowhammer-based Inter-Process Attacks. In *ACM Asia Conference on Computer and Communications Security (AsiaCCS '19), July 9–12, 2019, Auckland, New Zealand*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3321705.3329827

1 INTRODUCTION

Memory-corruption vulnerabilities pose a continuous threat to the security of modern computer systems. Attackers exploit these vulnerabilities to introduce malicious behavior to benign applications. Traditionally, these vulnerabilities are caused by programming errors [4, 6, 26, 31]. In the infamous case of *rowhammer*, however, researchers demonstrated an attack that exploits a weakness in DRAM chips to corrupt memory—even in the absence of any software vulnerabilities [14]. While initially termed by manufacturers to describe a glitch caused by electromagnetic coupling effects, they found that the repetitive access of the same memory cells can influence the content of physically neighboring memory cells which leads to bit flips, i.e., modification without access.

Rowhammer is particularly dangerous because it enables attackers to bypass any virtual memory isolation or memory-access permissions. Hence, it comes with no surprise that rowhammer is an effective technique to implement privilege escalation attacks. Seaborn and Dullien [27] were the first to demonstrate how the attacker can exploit the rowhammer vulnerability to flip bits in page table, which is a data structure to configure memory permissions. As a consequence, the attacker gains full control over the page table, and thus, the attacked system. Subsequently, researchers demonstrated the power of rowhammer attacks by exploiting the vulnerability to attack virtual machines in the cloud [25, 35], and even demonstrated that the rowhammer bug can be triggered from the browser [2, 11], and on other architectures than x86 [33].

Due to the nature of the rowhammer vulnerability it can only be fixed by modifying the hardware design. However, in the meantime, researchers suggested a number of software-based defenses to mitigate the effects of rowhammer. Aweke et al. [1] proposed to detect rowhammer-based attacks by monitoring the system's hardware-performance counters to detect a high number of cache misses. Although the approach incurs only a small run-time overhead it represents only a probabilistic defense, and hence, suffers from false positives.

Partitioning physical memory was proposed as an alternative approach, and related work demonstrated an isolation policy that protects the operating system from an user-space adversary. In particular, CATT [3] proposed a kernel extension that enforces physical isolation of user and kernel to ensure that attacker-controlled user memory cannot corrupt kernel memory (e.g., page tables). However, their mitigation is limited to two domains, kernel and user, and physically isolating a higher number of entities in system memory dynamically at run time currently remains an open problem. ZebRAM [15] extends the idea of physical memory isolation by statically inserting guard rows in memory.

Recently Gruss et al. [10] demonstrated that breaking the isolation between user and kernel memory might not be required in practice for an user-space adversary to escalate privileges: in particular, they showed that flipping bits in one of the setuid binaries

AsiaCCS '19, July 9–12, 2019, Auckland, New Zealand
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6752-3/19/07…$15.00
https://doi.org/10.1145/3321705.3329827

allows malicious users to elevate their privileges to root. We can already see such attacks becoming increasingly relevant, as attackers start to improve their primitives, e.g., by accelerating remote rowhammer attacks using GPUs to attack the browser on mobile devices [8].

For this reason, we present RIP-RH, the first DRAM-aware memory allocator that enables physical isolation of individual user processes running concurrently on the system. While security-aware memory allocators have been studied for a long time to harden existing systems against software-based memory-corruption attacks [16, 20, 29], we propose to harden existing allocators by dynamically safe-guarding adjacent pages based on the allocating process.

Dynamically managing multiple isolation domains poses a number of difficult challenges such as management of a potentially high number of processes, frequent and dynamic reallocation, and minimizing fragmentation. As we will show our design tackles all of these challenges by carefully adopting an over-allocation of memory pages to effectively protect against bit flips from surrounding rows by using *guard pages* located directly above and below the target row in DRAM.

We implemented our prototype of RIP-RH for Linux and thoroughly evaluated our implementation using a number of benchmarks and real-world applications. We tested our prototype in various real-world scenarios, for instance, by marking all setuid programs on a recent Ubuntu 16.04 machine as *critical*. We were able to verify that this successfully prevents physical co-location of attacker-controlled memory in the kernel's page cache. Our solution is compatible with standard system configurations and orthogonal to existing isolation defenses, e.g., against speculative execution attacks. As our isolation works in the physical memory domain, it does not interfere in any way with defenses against Meltdown [18], which operate in virtual memory.

To summarize, our main contributions are:

- We present RIP-RH, a DRAM-aware memory allocator to physically isolate process memory and deterministically defend against rowhammer-based attacks without requiring hardware changes. Our design is transparent to user space and does not disrupt legacy applications.
- We implement our isolation-based allocator as an extension to the physical page allocator in Linux to enforce our mitigation for all processes on the system. RIP-RH enables an isolation between different user processes in general and between user and kernel as a special case. This ensures that attackers cannot flip bits in memory that does not belong to them.
- We rigorously analyze RIP-RH's security by reimplementing the exploit steps that are necessary to force *physical co-location* (e.g, as described by Gruss et al. [10] in their recent user-level root exploit). In our extensive tests we demonstrate that the attacks no longer succeed in allocating neighboring memory when isolating the victim processes (such as the setuid binaries).
- We thoroughly evaluate the performance, robustness and usability of RIP-RH in a number of benchmarking scenarios and real-world applications. Our performance measurements indicate that our prototype already offers practical performance for typical use cases such as server applications and web browsers.

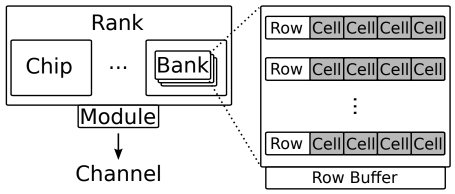

Figure 1: DRAM modules are highly standardized, consisting of multiple hierarchically components.

2 BACKGROUND

In this Section we briefly cover the concepts necessary for the understanding of the remainder of this paper. Firstly, we introduce technical background on DRAM, and secondly give an overview of existing rowhammer-based attacks and defenses.

2.1 DRAM Basics

Synchronous Dynamic Random Access Memory (SDRAM, or simply DRAM) is a clocked integrated circuit component that implements parallelizable volatile memory on a chip. Due to its comparatively low manufacturing costs, DRAM chips represent the most widely used implementation of main memory for computer systems today. As such, the design for DRAM is highly standardized, with both the DDR3 and DDR4 standards combined accounting for 97% of the market share in 2017 [12].

While there are different form factors for servers, desktops, laptops, and mobile devices, the conceptual design is the same, as depicted in Figure 1: one or two DRAM modules are connected via a *channel* to the mainboard. Each module consists of 1 or 2 *ranks*, which correspond to the front and back of a Dual Inline Memory Module (DIMM). Each rank in turn consists of a number of chips, which contain *banks*. Each bank comprises *rows*, which contain the individual *memory cells* that hold the electrical charge corresponding to the stored bit. A memory cell can be implemented in hardware by connecting a transistor (as switch) to a capacitor (as single-bit storage).

Memory accesses to separate banks can be parallelized and interleaved, however, within a bank access is only possible per row. During a read operation the charge from all cells in a row is transferred to the row buffer of that bank. The bits are then forwarded to the rank, channel, and finally the memory bus. Afterwards, the charge is written back to the row to restore its original state. Because the charge of the capacitor also degrades naturally over time, DRAM requires a continuous refresh operation, with typical refresh intervals of 32ms or 64ms. Since there can be only one row buffer for every bank, it represents a bottle neck for the access latency of the module.

The physical address space of the memory as seen by devices and the processor serves as an abstraction from these low-level implementation details of the hardware, and hence, they are usually not visible to the software. The memory controller implements

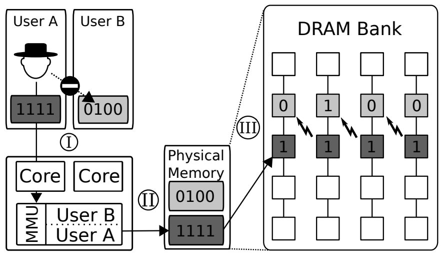

Figure 2: An adversary controlling one process can modify physical memory of other processes by exploiting the Rowhammer bug despite an isolation policy being enforced in virtual memory.

a physical-to-DRAM mapping that is intended to distribute the load between different DRAM structures and improve performance. This means, that while physical memory is addressed sequentially, consecutive memory pages can be mapped onto different rows, banks, etc.

2.2 Existing Attacks and Defenses

Similar to processor feature sizes, DRAM chip area has been decreased by manufacturers through increasing density of memory cells per chip to offer higher capacity, parallelism, and throughput. While memory errors have always been a randomly occurring phenomenon in microelectrical circuitry,[1] recent feature sizes approach less than 10nm, and electromagnetic coupling effects in the form of capacitive or inductive crosstalk are becoming increasingly problematic.

Kim et al. [14] were the first to systematically analyze what was considered mostly a reliability problem by manufacturers and discovered that by repeatedly accessing one row they were able to influence contents of memory cells from *neighboring* rows within the same bank (hence, the term *row*-hammer). In their study they found the behavior to be reproducible from software on a large number of (mostly 2GB) memory modules across different manufacturers.

Yet, rowhammer is not just a reliability issue, but has significant repercussions for security as well, as it leaves systems vulnerable to a range of dangerous attacks. The general workflow of such an attack is depicted in Figure 2: an adversary with some limited access[2] to the system is confined Ⓘ, e.g., by means of de-privileging and MMU-based isolation in virtual memory. Hence, any attempted access to isolated system entities (such as the sandbox, another process, the kernel, or the hypervisor) is prevented. However, by allocating Ⓘⓘ or otherwise obtaining access to memory that is physically adjacent to victim memory in the DRAM module, the attacker can exploit the rowhammer effect to corrupt its contents ⓘⓘⓘ.

Consequently, rowhammer was quickly picked up by the security community and researchers demonstrated two row-hammer-based exploits [27] by breaking out of the Chrome sandbox and gaining kernel privileges from a user process. The sandbox escape was achieved by flipping bits in the target address of a restricted `jmp` instruction. The kernel privilege escalation exploited bit flips in kernel memory by creating a huge number of virtual mappings to spray the physical memory with page table entries. A bit flip at the correct position in a page table entry then allows an adversary to re-map the kernel as writable and take over the system.

Subsequently, several possible defenses were proposed, however, most require changes to the hardware or suffer from false positives and we discuss them in detail in section 9. To thwart rowhammer attacks on kernel memory Brasser et al. [3] recently presented an isolation-based defense against rowhammer by partitioning the physical memory into two security domains purely in software. By leveraging knowledge about the mapping between physical addresses and DRAM structures it is possible to isolate kernel memory within a dedicated region in DRAM. Consequently, the attacker can no longer induce bit flips in kernel memory from user space processes. However, their defense is designed for the special case of two domains, i.e., kernel and user land, and there are significant challenges when isolating a larger number of domains.

3 THREATMODEL

Our threat model is derived from and in line with the related work in this field [1–3, 11, 14, 24, 25, 27, 33, 35]:

- The attacker has full control over a user-space process and can execute code, allocate memory, and utilize kernel services through system calls.
- The physical-to-DRAM mapping is publicly documented or has been reverse engineered.
- The operating system has sole privileged access to the machine and cannot be corrupted through rowhammer, since its memory is isolated (e.g., through the partitioning scheme described in [3]).
- The OS and processes are isolated from each other, which is enforced through different address spaces in virtual memory, and the memory-management units (MMU, I/O-MMU) are configured accordingly.
- The main memory of the system is vulnerable to rowhammer.

In this setting the goal of an adversary is to compromise another process, e.g., to elevate privileges, gain control over some service, or corrupt its data. While run-time attacks through exploitation of memory-corruption vulnerabilities represent real-world threats, we treat software-based attacks as an orthogonal problem [5–7, 9, 17, 21, 23, 30].

4 DESIGN

In this section we present the design of RIP-RH. We first give a brief overview of its structure and workflow. Secondly, we present how to isolate dynamic process memory physically. Finally, we highlight the main challenges that RIP-RH tackles.

4.1 Overview

The main goal of RIP-RH is to enable physical isolation for user space processes. To this end, we devise an interface for managing the startup, isolation policy to be used, and tear-down of *critical*

[1]For instance, through thermal conditions, alpha particles from natural decay, or cosmic rays [19].
[2]In this case the attacker has access to a user process, but in principle this could also be a VM-guest or a sandboxed script in a document viewer or browser.

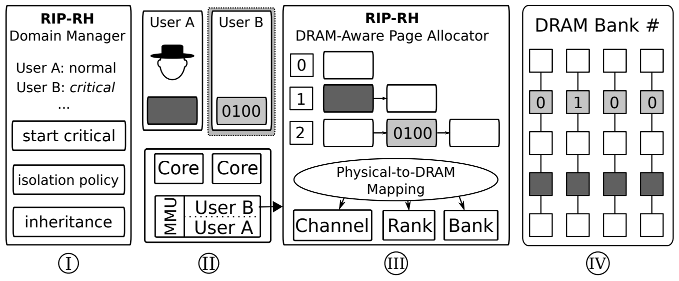

Figure 3: Our design of RIP-RH incorporates components for (1) managing process-owned memory and isolation policies, (2) an interface for running isolated processes, (3) a DRAM-aware physical page allocator, and (4) a partitioning scheme to thwart inter-process rowhammer attacks, such as *opcode flipping* [10]. Critical processes can be started from the domain manager, in this case user process B is configured to be isolated from the attacking process A. Consequently, its allocated pages will be allocated from the free lists in such a way, that guard pages remain above and below its physical row index on the DRAM module.

processes, i.e., programs that should be isolated *physically* in DRAM by the system at run time. Figure 3 depicts a high-level overview of RIP-RH.

The first component, the Domain Manager Ⓘ, extends the OS to add the notion of a *critical* process, i.e., a user-space process that should be isolated from all other processes.[3] It also provides services to user space for starting, stopping, and managing such processes. There are different possible configuration options for the isolation policy, and the forking behavior of such processes, which the owner of the process can choose prior to launching a critical process.

The second component Ⓘⅰ represents the run-time for critical processes. It handles the meta-data and book-keeping, e.g., for running processes, physical page ownership, and memory zones on the system. It further delegates all allocation requests to the physical page allocator of the OS.

The main component is our DRAM-aware physical page allocator Ⓘⅰⅰ, which integrates with the buddy allocator of Linux. It enhances its operation by embedding the physical-to-DRAM mapping and dynamically translating the location of memory in physical address space to physical locations in DRAM.

Lastly, RIP-RH comes with a set of pre-defined default isolation policies Ⓘⅴ that mitigate rowhammer-based inter-process attacks through co-located and adjacent memory pages. This also ensures that any two critical processes cannot interfere with each other. We discuss shared memory handling in Section 8.2.

4.2 Physically Isolating Processes

In principle, isolation between two physical pages in DRAM is ensured if one of the following conditions is met: the two pages reside in different modules, ranks, or banks. If two pages belong to the same bank, they can influence each other if their physical proximity allows for electromagnetic coupling effects between the charges of their memory cells. As Kim et al. observed in their original study [14], this affects *consecutive* rows, i.e., rows that are placed directly next to each other within a bank. However, because of the physical-to-DRAM mapping the distance between pages in neighboring rows within one bank can be very high in the physical address space, as pages are distributed across different banks.

For this reason, our isolation of dynamic memory must translate physical addresses of pages into the corresponding rank, bank, and row number, and perform neighborhood checks with respect to the underlying DRAM structures. To this end, RIP-RH incorporates (parts of) this mapping to make the physical page allocator aware of the DRAM-structure. While the physical-to-DRAM mapping is publicly documented for some architectures, this is not always the case. Yet, for the most prominent architectures (x86 and ARM), the mapping function has been reverse engineered previously by the related work [22, 35].[4]

4.3 Challenges

Realizing physical isolation for user-space processes comes with a number of difficult challenges, such as (1) supporting a potentially large number of processes, (2) mapping process memory and allocation requests to physical pages, (3) a strategy for avoiding fragmentation and over-allocation, while (4) minimizing run-time overhead and latency. In the following we briefly explain each of these challenges and how RIP-RH tackles them on a high level. We provide a detailed explanation of our prototype in the next section.

4.3.1 Large Number of Processes. While in theory there can be any number of processes per system, this number is limited in practice. For instance, on Linux the default maximum number of processes is 32767 [13]. Further, if the running processes consume more memory than available the system will start swapping out memory pages to disk. If disk space runs low, the kernel has no choice but to start killing processes. Since not all running processes might be security sensitive, or require the physical DRAM isolation, RIP-RH enforces its isolation upon request. This means that the owner of the process must notify the OS that the process should

[3]For sake of simplicity we assume the OS to be the most privileged software component in the system. However, in virtualized system the Domain Manager needs to be replicated in the Hypervisor.

[4]For special use cases, e.g., embedded programming, physical-to-DRAM-aware memory allocation might give additional performance, which could be an incentive for manufacturers to provide this information to the public.

be isolated. Our domain manager provides several interfaces for creating and managing such critical processes.

4.3.2 Attributing and Mapping Allocation Requests. To isolate critical processes, RIP-RH has to track allocation requests in the physical page allocator. The usual workflow for such allocations is as follows: a process creates a virtual memory mapping and subsequently accesses the mapping. This results in a page fault, that is handled by the OS, which delegates the page fault to the physical page allocator. We modify the OS to pass the process identifier along with the allocation request and also store it in the meta data of the allocated physical page. This enables the allocator to perform domain checks for each individual allocation request dynamically.

4.3.3 Fragmentation and Over-allocation. Any memory partitioning scheme that aims to support an isolation in physical DRAM will inevitably cause some memory overhead, as the isolation requires buffer rows above and below the allocated memory. A typical size of a row, e.g., in DDR3 memory, is 8K, and hence, a row in a bank contains two physical memory pages. Hence, the buffer space consists of two pages on the boundary of a memory domain. To further minimize the memory overhead and avoid fragmentation, RIP-RH dynamically checks for pages that are physically adjacent to memory of non-critical processes and prevents allocation to those regions.[5]

4.3.4 Minimizing Run-Time Overhead. To minimize the overhead RIP-RH leverages the design of the physical page allocator, which is optimized for performance. Our additional security checks are integrated into the existing architecture of the allocation algorithm, which maintains separate freelists for different allocation sizes (called *order*). If an allocation for a particular order cannot be fulfilled the algorithm skips to the next higher order and splits the block of pages into smaller chunks, which are then assigned to the requesting process.

5 IMPLEMENTATION

In this section we describe our prototype implementation of RIP-RH, which is a patch to the Linux kernel version 4.10, in detail. The patch consists of 586 lines of code with changes in 13 files. We compile and install our custom kernel on top of an Ubuntu 16.10 installation, using a standard configuration. We further developed a custom system call wrapper to automatically launch new isolated processes in user space.

5.1 RIP-RH Domain Manager

The first component of our implementation is the domain manager, which allows a user to launch *critical* processes. For this, we extended the kernel interface by adding a new system call definition. Our new system call will invoke the `__do_fork_critical` function, which starts a new *critical* process in the kernel. We further added several flags to the process descriptor in Linux (called `task struct`), which our system call sets accordingly to mask a process as *critical.* The system call function then sets up the necessary parameters and delegates the rest of the process creation to the `do_fork` function in the kernel. One important property of *critical* processes is the inheritance policy of the isolation: to guarantee that a forked critical process does not create uncritical child processes we also modified the original fork function to checks the parent process descriptor for its isolation policy and critical flag. If these are present, a fork call from such a process will automatically pass these settings on to the child process. It is noteworthy to mention that isolating the kernel should be possible in a similar manner, by assigning it a process descriptor with id 0 (the first process, `init`, always has the id 1). However, we did not implement this for our prototype. We note, that our process-isolation defense against rowhammer attacks is orthogonal to defenses that operate in the virtual memory domain, and hence, supports a multitude of real-world system configurations (e.g., such as large page tables).

[5]As a possible optimization RIP-RH can be extended to reuse guard pages for non-critical or integrity protected data, similar to ZebRAM [15].

5.2 Buddy allocator modifications

The second component of the implementation modifies the standard physical page allocator of the Linux kernel to make it aware of the underlying DRAM structure and isolate privately mapped pages of processes that are configured accordingly with a domain isolation policy. To implement this, page allocations must be tracked according to the requesting process, which our prototype achieves by setting the identifier of the current process upon receiving a paging fault, and passing it to the page allocator. Generally, there are two types of private memory mappings, which can cause page faults: anonymous and file-backed mappings. The difference between the two is that anonymous mappings are satisfied by the buddy allocator directly, whereas file-backed mappings are normally satisfied from a block device (e.g., a hard disk or flash drive) and added to the kernel's *page cache.* If the allocation originates from a file-backed mapping, the respective `do_file_read` implementation will allocate a free page from the buddy allocator and initiate the disk IO before placing the page in the page cache. In both cases, the modified buddy allocator performs a series of checks based on the current configuration of the requesting process and allocates physical pages accordingly.

In addition to that, every physical page in the kernel is tracked through an individual page descriptor object (called `page struct`), that stores meta data about the page.

We supplement the page struct definition by adding the corresponding process id, as this information is required by the page allocator to determine if a particular page belongs to a critical process. We note that meta-data tagging is required since we need to perform checks for pages that *do not* belong to the current process, we cannot simply infer ownership from the currently active mapping (`CR3` control register), but have to tag physical pages in their metadata `page struct` objects. We bootstrap and reset this information in `__init_single_page`, which initializes all page descriptors during boot and upon freeing of pages during run time. We further modify the allocator's main entry point, `__alloc_pages_nodemask`, which first determines the memory zone for the allocation (usually *normal zone* for user-space allocations), and then calls a number of helper functions. Internally, the page allocator separates the allocation by kind and size. The number of pages to be allocated is expressed as a power of two, the exponent of which is called the *order* of the allocation. The allocator maintains separate freelists per order for every memory zone.

5.3 Spatial Isolation through guard pages

To ensure the spatial physical isolation between allocations for different processes we reserve *guard pages* in the upper and lower rows of an allocated page. In case this is not possible, the buddy allocator continues with the next page in the freelist. If the page fulfills the criteria it is returned to the process. If the current freelist contains no pages that fulfill the criteria the allocation algorithm switches to the next higher order and iterates over the corresponding freelist. While this may result in a slight over-allocation the allocator splits the chunk of pages in two halves (i.e., *buddies*), returning one buddy to the process and appending the remaining buddy to the freelist. Upon freeing, the allocator checks if the pages can be merged with existing buddies, and if so, creates a larger block that is inserted into the higher order freelist again. This also ensures that the amount of over-allocation is minimized and fragmentation is kept to a minimum.

It is noteworthy to mention that the guard pages are not actually allocated, but kept as part of the free memory page pool instead. This means that guard pages for a page frame could themselves be allocated as physical memory by other processes. However, upon checking of the upper and lower rows, this allocation will be prevented, as the guard pages are tagged with the guarded process identity (i.e., the PID). For subsequent allocations of the guarded process, these pages remain allocatable. This means that when two allocated DRAM regions are growing close they can be merged into one guarded region (if they belong to the same process) by allocating the guard pages in between the regions and only keeping the pages on top and below the upper-most and lowest DRAM row. We verified that RIP-RH can handle also potentially large number of processes and provide a detailed evaluation of this case in Section. The calculation of the rank, bank, and row indexes was implemented in the form of C macros, which will be inlined to reduce the performance impact of those checks. For this, we leverage the physical-to-DRAM mapping, which has been reverse engineered by the related work. While this information depends on the architecture and type of RAM (DDR3 or DDR4), we configure this information during kernel compilation. This also enables us to translate the corresponding page struct object for a specific physical address to its specific physical location on the DRAM module. However, it is also noteworthy that the row index by itself can actually be calculated without requiring micro-architecture specific mappings, as the number of physical pages per row is determined by the DRAM standard. This means that the row index for a physical page can be calculated from its physical address given the number of pages per row just from the DRAM module type.

6 SECURITY ANALYSIS

To exploit rowhammer, an adversary needs to achieve two conditions: (1) the system must be susceptible to bit flips, i.e., if the DRAM module is not vulnerable to the rowhammer hardware bug it cannot be exploited; and (2) the attacker must force *physical co-location* to victim memory. For our work we assume the first condition to be naturally fulfilled, as many DRAM modules are vulnerable to bit flips in practice (cf., section 3).

To test this, we re-implemented the physical co-location strategy for an attacker program **A** and a simple test program **V**. The attacking program tries to force physical co-location in a loop (for both *anonymous* and *file-backed* mappings, i.e., **A** can ultimately exhaust the rest of the available physical memory on the system (depending on the co-location strategy). As the system becomes unresponsive if *all* physical memory is allocated by a process, we reserve 2% of the system's memory pages and **A** will not allocate those pages in any case. We then let our test program **V** allocate 1000 individual memory pages (i.e., 1000 separate allocations of order 0). For every page we logged the allocated address and determined its physical address through the `/proc/self/pagemap` interface. We then executed the two programs in two different configurations.

6.1 Non-critical vs. Critical

In our first test series we setup two processes, a critical process (**V**) and a non-critical process (**A**). We then iterate through all successfully allocated pages to scan for co-location with one of the pages owned by **V**. We further added several debugging output statements to the page allocator, which was then written to the kernel log. Via this log we confirmed that allocation requests included the physical address of a reserved page, and that those pages were blocked correctly by RIP-RH upon allocation request from **V**. We let the experiment run for more than 48 hours and also repeated each test after rebooting the system between trials, however, process **A** never obtained any page that was co-located to a page belonging to **V**.

6.2 Critical vs. Critical

To further ensure that RIP-RH correctly isolates critical processes physically we subsequently started two critical processes **C-A** and **C-V**. However, in this scenario the attacker is assumed to have control over one of the critical processes **C-A**. We now perform the same test as before, only that the attacking process should now be isolated as well. To this end we again logged and verified the addresses of allocated pages in the kernel log for **C-V**. Again, we let the experiment run for more than 48 hours and restarting the test several times while rebooting the machine in between the experiments.

We were able to confirm that also in this scenario the attacker is actively blocked from allocating any pages that are co-located to **C-V**, by cross-checking the physical addresses of the allocated benign pages with the denied allocation requests in the kernel logs.

7 PERFORMANCE EVALUATION

We thoroughly evaluated RIP-RH against several popular benchmarking suites and also on real-world applications such as Firefox. Since RIP-RH isolates processes upon request, we wrote a small wrapper script that invokes any command line program by calling the `fork_critical` system call, which will setup the isolation and mark the started process as *critical*.

These standard benchmarks are designed to explicitly stress the system, e.g., by creating memory pressure situations and evaluate system behavior in a number of different memory allocation situations. Additionally, we conducted micro-benchmarks to assess potential initialization and memory overhead.

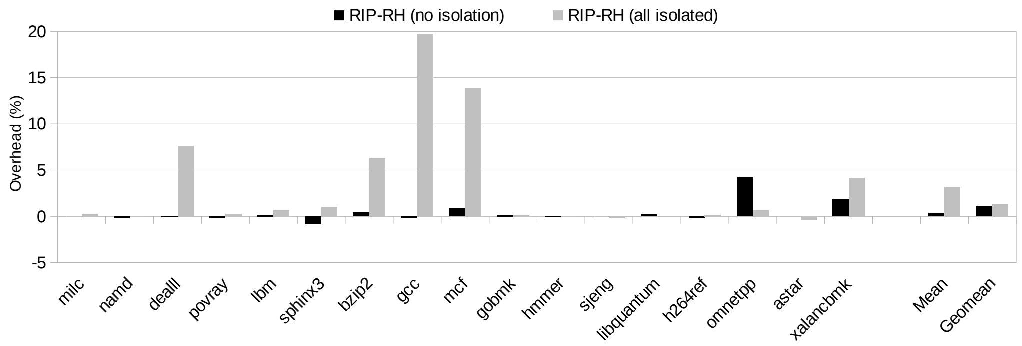

Figure 4: SPEC2006 results for RIP-RH with no tests marked as *critical* and all tests running as *critical* processes respectively.

The machine used for our tests was equipped with an Intel i3 quad-core processor clocked at 2.93GHz and 2 DRAM modules with a capacity of 2GB each. We first describe and explain the results for Spec, then Phoronix and LMBench, and finally our real-world experiments.

7.1 SPEC CPU2006 Benchmarks

The Spec CPU2006 benchmarking suite contains two sets of tests, all of which mainly perform CPU-bound computation. The first range of tests work with floating point numbers, the second focuses on integer computation. Spec performs three runs and then selects one of those as a representative. We ran Spec on three different kernels: first, a plain Linux 4.10, second, with RIP-RH but none of the tests marked as *critical* processes, and third, with RIP-RH and all tests marked as *critical*. The results of those runs are plotted in Figure 4. Overall we can see that the average performance overhead induced by RIP-RH is less than 4% (3.17%), demonstrating that a page-based physical isolation of dynamic process memory in DRAM from software is practical. In particular, there is barely any effect visible through the OS-level changes in RIP-RH for non-isolated processes. For isolated processes, there are two notable tests, `gcc` and `mcf`. We analyzed the behavior of those tests and found that for `gcc` the reason is that the test compiles several input files. For every compilation unit, the compiler is restarted as a process, meaning that the load-time overhead of setting up the isolation is repeated for every compilation. Since the actual run-time of the test is rather short, but the time it takes to initiate an isolated process is fixed, this means that the relative overhead is higher compared to all other tests.

7.2 Phoronix Test Suite

The Phoronix Test Suite contains a series of different benchmarks, such as filesystem access, network performance, computation, memory benchmarks, and also a couple of real-world applications such as Apache, OpenSSL, and 7-zip. In summary, the performance impact of RIP-RH on the filesystem and network benchmarks is negligible.[6] This makes sense, since most of the memory is handled by drivers in the OS, which does not affect user memory directly. Similarly, the cachebench and computation-heavy application benchmarks like PHPBench, PyBench, and OpenSSL are practically unaffected by RIP-RH's process isolation as well. One exception to this is 7-zip, which was unexpected since compression processes are usually CPU-bound as well. However, during our investigation we found that 7-zip uses separate forks instead of separate threads for parallel processing, and hence, the nominal overhead for the isolation setup is higher (similar to the gcc test in Spec).

To confirm that RIP-RH does not otherwise affect the performance of 7-zip we tested the program separately with multi-processing mode disabled and repeated the compression test. Without any isolation compressing a 10G file sequentially with 7-zip took on average over three repetitions 9 minutes and 39 seconds; when we ran 7-zip as a *critical* process sequentially compressing the same file took 9 minutes and 40 seconds on average. In summary, RIP-RH shows no adverse effects in terms of memory or performance for standard applications.

7.3 LMBench Microbenchmarks

To further assess the initialization and potential memory overhead observed in the case of `gcc` and `mcf`, we conducted micro benchmarks, using the widely used LMBench test suite. In particular, we started the LMBench tests with and without using our fork wrapper script, showing visible overhead in application startup, but no slowdowns in terms of write or read performance.[7]

7.4 Real-world Applications

To better assess the performance of RIP-RH we conducted additional experiments using the three popular applications OpenSSH, Firefox, and Chrome. We started both applications as *critical* processes. Further we created isolated variants of all setuid binaries to demonstrate feasibility.

7.4.1 OpenSSH. For our real-world use case of OpenSSH we started the SSH daemon itself as a critical process. We then connected over the local network to copy files of different sizes and recorded the file transfer times. To eliminate any significant random noise events, we repeated the whole process three times and rebooted the machine in between our experiments.

Overall, these results are in line with our benchmark measurements. In particular, the overhead in all cases is below 4%, also for

[6]In Appendix A we provide the detailed scores and overhead numbers for all of the experiments we conducted (Table 4).

[7]Detailed evaluation results are provided in Appendix A (Table 2).

smaller file transfers that take just a couple of seconds over the local network.[8] The experiment demonstrates that a physical isolation for processes is practical in the network service scenario.

7.4.2 Firefox. In our second real-world experiment we tested the performance of RIP-RH while running the popular browser Firefox as a physically isolated user process. We used the browser benchmarks Kraken and Octane to see if our prototype would impact the browsing performance and thereby affect user experience.

We found the performance impact for both benchmarks to be less than 3% for an isolated Firefox process.[9] We conclude that RIP-RH does not adversely affect user experience while running desktop applications such as Firefox.

7.4.3 Chrome Tabs. In addition, to provoke high memory pressure in order to get an understanding of the effects of memory fragmentation we tested how many isolated tabs a user could open on the popular Chrome browser (in contrast to Firefox, opening a tab will start a new process on Chrome). In our experiment, we were able to open and load 100 tabs without problems from a Chrome process that was marked as critical. Hence, we conclude that RIP-RH represents a feasible defense in practice—also for a larger number of processes.

7.4.4 `setuid` binaries. We further verified that isolating all setuid binaries is feasible in practice by marking all of them as critical. We were able to successfully boot the system without any problems and also restarted the phoronix test suite (we omit these results for brevity as there are no noticeable deviations).

8 DISCUSSION

In this paper we focused on isolation policies between individual user-level programs. However, in this section we discuss several areas that might be of interest for future work.

8.1 In-process isolation

One possible extension of our physical DRAM isolation mechanism could be realized on the allocation level. For instance, instrumenting the system-wide heap allocator would allow for an isolation policy between individual application components or threads. This would be particularly useful for large single-process applications, that are structured into components.

8.2 Shared memory

Memory areas that are accessible to multiple processes are problematic for memory-isolation schemes for several reasons: first, shared memory might be accessed by a malicious third-party process. Second, even if all processes with access to the shared memory region are benign, an adversary might trick a benign process into maliciously accessing the shared region (i.e., a *confused-deputy attack*). Third, while individual third-party processes by themselves might be benign, they could act maliciously in combination by working together (i.e., a *collusion attack*). For these reasons, we argue that shared memory—in any memory-isolation scheme—must be considered untrusted and potentially compromised. Consequently, RIP-RH does not support isolation of shared memory regions. In practice utilizing shared memory can be avoided, e.g., through techniques such as static linking or decoupling shared components from the main application (called *broker architecture*).

9 RELATED WORK

In this section, we briefly summarize existing rowhammer attacks and compare RIP-RH against other proposed defenses.

9.1 Rowhammer-based Attacks

As described in section 2 induced bit flips were first systematically study and analyze Kim et al. [14], however, they did not provide any practical attack.

The first practical rowhammer-based privilege-escalation attacks were presented by Seaborn and Dullien [27], by exploiting the `clflush` instruction on x86. First, they exploited rowhammer-induced bit flips to escape the Chrome browser sandbox (which is called Native Client, or NaCl [36]). This attacks shows, that malicious developers can escape the sandbox by exploiting rowhammer, and achieve code execution on the victim machine. Second, the authors presented an attack to compromise the kernel from an unprivileged user-mode application by resorting to rowhammer. By manipulating page-table entries via rowhammer the attack can gain the possibility to modify arbitrary kernel memory, and subsequently compromise the entire system

Combining the first and second attack in a two-stage attack it is easy to imagine that an adversary can completely compromise machine remotely, i.e., by delivering malicious java script first escaping the sandbox and afterwards the kernel.

Thereafter, Qiao and Seaborn [24] demonstrated a new rowhammer attack using the class of non-temporal instructions on x86 (e.g., `movnti`). Since many library functions, such as `memcpy`, utilize these non-temporal reads and writes for optimization purposes, an adversary can exploit rowhammer by resorting to code-reuse attack techniques [28]. Hence, it is not required to inject a dedicated code to induce bit flips, instead existing code can be re-used to conduct an attack.

Further, Aweke et al. [1] demonstrated that executing a rowhammer attack is possible even without leveraging any dedicated instruction to circumvent the caches and force a memory access. Instead, the authors crafted a memory-access pattern that will force the processor to evict a particular element from the cache reliably and at a high rate. Therefore they concluded that higher refresh rates for DRAM would not be enough to successfully prevent rowhammer attacks.

Gruss et al. [11] showed that rowhammer attacks can also be launched from JavaScript. More specifically, the authors were able to exploit bit flips in the page tables only by running a JavaScript program in a recent version of Firefox.

Bosman et al. [2] followed up on this work by leveraging memory deduplication in Windows 10 to create fake JavaScript objects. They corrupted these objects by generating bit flips and subsequently gained arbitrary read/write access within the browser process.

Later, Razavi et al. [25] applied a very similar technique to attack and fully compromise cryptographic keys in co-located virtual machines running on the same host.

[8]We provide detailed evaluation results in Appendix A (Table 1).
[9]We provide detailed evaluation results in Appendix A (Table 3).

Xiao et al. [35] presented a second cross-VM attack at the same time: they exploited rowhammer to manipulate page-table entries of the Xen hypervisor to gain hypervisor privileges and compromise the host. They were also the first to present a methodology for automatically reverse engineering the physical-to-DRAM mapping, i.e., the exact relationship between physical addresses and DRAM structures such as rows and banks.

Pessl et al. [22] independently presented a similar methodology to reverse engineer this mapping. Based on the uncovered DRAM mappings, they demonstrated that cross-CPU rowhammer attacks are possible, and for the first time presented a practical attack on DDR4 modules. Moreover, Van der Veen et al. [33] demonstrated that the original rowhammer exploit can be used to escalate privileges on smartphones.

More recently, Gruss et al. [10] demonstrated that row-hammer-based privilege-escalation exploits are a realistic threat, they showed that real-world setuid binaries such as `sudo` can be exploited through a technique called *opcode flipping*.

Further, Frigo et al. [8] showed that rowhammer attacks can be accelerated by using the graphical-processing unit (GPU) of the system. As they were able to demonstrate, this opens up a new attack vector for exploiting bit flips as GPU memory can be controlled completely independent from the CPU, e.g., from a browser.

To summarize all the attacks described in the literature require memory belonging to a different security domain (e.g., kernel or setuid process) to be physically co-located to memory that is under the attacker's control. Our defense prevents such direct co-location between user-level processes.

9.2 Defenses against Rowhammer

Kim et al. [14] presented a list of defense strategies, however, most of their solutions require modifications to the hardware, such as improved manufacturing, higher refresh rates, and error-correcting codes. Consequently, these solutions are not practical, as the production of improved hardware is costly, and deployment of such hardware typically takes many years, if implemented at all. Higher refresh rates for DRAM (e.g., every 16ms instead of 64ms) were also proposed by manufacturers as rowhammer mitigation as the attacker needs to access rows many times between two refresh events, in order to destabilize any adjacent memory cells and produce a bit flip. Refreshing rows more frequently could serve as a stabilizing factor to prevent attacks. However, Aweke et al. [1] successfully conducted rowhammer attacks within 16ms. Therefore, higher refresh rates do not represent an effective countermeasure against rowhammer-based attacks. Moreover, error-correcting codes (ECC) are able to detect and correct single-bit errors. However, Kim et al. [14] already demonstrated in their original study that rowhammer can induce multiple bit errors, and hence, ECC memory does not prevent rowhammer-based attacks. Finally, Kim et al. [14] suggested probabilistic adjacent row activation (PARA) to mitigate rowhammer in the memory controller. The idea is that any access to a row can trigger an activation of adjacent rows with some configurable probability. Since malicious rows are activated many times during an attack, the victim row will eventually receive a refresh, and hence, will be stabilized during the attack. The advantage of this approach is that is has a low performance overhead, however, it also requires changes to the memory controller and is therefore not suited to protect legacy systems, where the hardware cannot be updated easily. So far, only few software-based defenses against rowhammer were proposed: first, Aweke et al. [1] proposed ANVIL, which uses performance counters to detect high cache-eviction rates, which is a typical effect of an ongoing rowhammer-based attack. Nonetheless, their defense has several disadvantages, e.g., it requires CPU performance counters, and also relies on heuristics, which means that ANVIL suffers from false positives. Second, CATT [3] provide a deterministic approach that stops rowhammer-based kernel-privilege escalation attacks. It devise a partitioning scheme for isolating the kernel memory domain from that of any user processes. Consequently, an adversary can still induce bit flips, but no longer force physical co-location to kernel-owned memory. However, they only demonstrate their isolation for the special case of two domains, i.e, kernel and user, and partitioning DRAM physically among a larger number of entities comes with significant challenges. As we demonstrate in this paper, RIP-RH tackles all of these challenges and enables physical isolation for user processes. ZebRAM [15] uses a static approach to isolate memory rows using guard rows. RIP-RH, in contrast, dynamically manages protection domains, which allows for better memory utilization.

Other defenses apply the general concept of physical memory isolation, as introduced by CATT [3], to provide protection for special scenarios. For instance, "GuardION only enforces that DMA-based Rowhammer attacks can no longer flip bits in another process or kernel memory" [34]. ALIS [32] proposes physical memory isolation for network buffers.

10 CONCLUSION

In this paper we present RIP-RH, the first rowhammer defense that allows physical isolation of processes. Through its DRAM-aware physical page allocator, it enables the creation of *critical* processes on a system, which will be isolated physically in memory from all other processes. As we demonstrate in our evaluation, RIP-RH offers a high performance with an overhead of only 3.17% for standard benchmarks. This shows that software-based defenses for rowhammer are practical also for a larger number of security domains.

ACKNOWLEDGMENT

This work has been supported by the German Research Foundation (DFG) as part of projects HWSec, P3 and S2 within the CRC 1119 CROSSING, by the German Federal Ministry of Education and Research (BMBF) and the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within CRISP, by BMBF within the projects iBlockchain and CloudProtect, and by the Intel Collaborative Research Institute for Collaborative Autonomous & Resilient Systems (ICRI-CARS).

REFERENCES

[1] 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. In *International Conference on Architectural Support for Programming Languages and Operating Systems.*

[2] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2016. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *IEEE Symposium on Security and Privacy*.

[3] 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*.

[4] Nathan Burow, Scott A. Carr, Stefan Brunthaler, Mathias Payer, Joseph Nash, Per Larsen, and Michael Franz. 2016. Control-Flow Integrity: Precision, Security, and Performance. (2016). http://arxiv.org/abs/1602.04056

[5] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. 2015. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *IEEE Symposium on Security and Privacy*.

[6] J. Criswell, N. Dautenhahn, and V. Adve. 2014. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *IEEE Symposium on Security and Privacy*.

[7] Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad-Reza Sadeghi. 2012. MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones. In *Annual Network and Distributed System Security Symposium*.

[8] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2018. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In *IEEE Symposium on Security and Privacy*.

[9] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. 2016. Fine-Grained Control-Flow Integrity for Kernel Software. In *IEEE European Symposium on Security and Privacy*.

[10] 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 *IEEE Symposium on Security and Privacy*.

[11] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2016. Rowhammer.js: A Cache Attack to Induce Hardware Faults from a Website. In *Conference on Detection of Intrusions and Malware and Vulnerability Assessment*.

[12] IC Insights. 2017. DDR4 Set to Account for Largest Share of DRAM Market by Architecture. http://icinsights.com/data/articles/documents/969.pdf.

[13] Michael Kerrisk. 2010. *The Linux programming interface: a Linux and UNIX system programming handbook*. No Starch Press.

[14] 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 *Annual International Symposium on Computer Architecture*.

[15] Radhesh Krishnan Konoth, Marco Oliverio, Andrei Tatar, Dennis Andriesse, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. 2018. ZebRAM: Comprehensive and Compatible Software Protection Against Rowhammer Attacks. In *USENIX Symposium on Operating Systems Design and Implementation*.

[16] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. 2015. Preventing Use-after-free with Dangling Pointers Nullification. In *Annual Network and Distributed System Security Symposium*.

[17] Zhiqiang Lin, RyanD. Riley, and Dongyan Xu. 2009. Polymorphing Software by Randomizing Data Structure Layout. In *Conference on Detection of Intrusions and Malware and Vulnerability Assessment*.

[18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: reading kernel memory from user space. In *USENIX Security Symposium*.

[19] Nick Nikiforakis, Steven Van Acker, Wannes Meert, Lieven Desmet, Frank Piessens, and Wouter Joosen. 2013. Bitsquatting: Exploiting bit-flips for fun, or profit?. In *Proceedings of the 22nd international conference on World Wide Web*.

[20] Gene Novark and Emery D Berger. 2010. DieHarder: securing the heap. In *ACM Conference on Computer and Communications Security*.

[21] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. 2010. G-Free: Defeating Return-Oriented Programming through Gadget-less Binaries. In *Annual Computer Security Applications Conference*.

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

[23] Jannik Pewny and Thorsten Holz. 2013. Control-flow Restrictor: Compiler-based CFI for iOS. In *Annual Computer Security Applications Conference*.

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

[25] 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*.

[26] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *IEEE Symposium on Security and Privacy*.

[27] Mark Seaborn and Thomas Dullien. 2016. Exploiting the DRAM rowhammer bug to gain kernel privileges. https://googleprojectzero.blogspot.de/2015/03/exploiting-dram-rowhammer-bug-to-gain.html.

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

[29] Sam Silvestro, Hongyu Liu, Corey Crosser, Zhiqiang Lin, and Tongping Liu. 2017. FreeGuard: A Faster Secure Heap Allocator. In *ACM Conference on Computer and Communications Security*.

[30] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William R. Harris, Taesoo Kim, and Wenke Lee. 2016. Enforcing Kernel Security Invariants with Data Flow Integrity. In *Annual Network and Distributed System Security Symposium*.

[31] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *IEEE Symposium on Security and Privacy*.

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

[33] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. 2016. Drammer: Deterministic Rowhammer Attacks on Commodity Mobile Platforms. In *ACM Conference on Computer and Communications Security*.

[34] Victor van der Veen, Martina Lindorfer, Yanick Fratantonio, Harikrishnan Padmanabha Pillai, Giovanni Vigna, Christopher Kruegel, Herbert Bos, and Kaveh Razavi. 2018. GuardION: Practical Mitigation of DMA-Based Rowhammer Attacks on ARM. In *Detection of Intrusions and Malware, and Vulnerability Assessment*.

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

[36] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. 2009. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *IEEE Symposium on Security and Privacy*.

Test	Plain 4.10	RIP-RH	OH
Kraken (all LIB)			
astar	119.7ms	118.7ms	-0.84%
beat	171.0ms	182.7ms	6.40%
dft	232.6ms	203.3ms	-12.59%
fft	86.3ms	96.5ms	11.82%
oscillator	93.1ms	102.0ms	9.56%
gaussian	124.6ms	124.4ms	-0.16%
darkroom	163.3ms	161.1ms	-1.35%
desaturate	107.3ms	106.6ms	0.65%
parse	71.7ms	70.1ms	-2.23%
stringify	91.4ms	96.1ms	5.14%
aes	94.3ms	95.4ms	1.17%
ccm	160.8ms	168.7ms	4.91%
pbkdf2	196.1ms	192.4ms	-1.89%
sha256	72.2ms	73.5ms	1.80%
Total	1784.4ms	1791.5ms	0.4%
Octane (all HIB)			
Richards	19932	20013	-0.41%
Deltablue	30018	29713	1.02%
Crypto	19722	19488	1.19%
Raytrace	64157	65793	-2.55%
EarleyBoyer	21723	21962	-1.1%
Regexp	2150	2089	2.84%
Splay	11124	10245	7.90%
SplayLat	14789	13678	7.51%
NavierStokes	27992	27992	0%
pdf.js	7556	8036	-6.35%
Mandreel	12608	12800	-1.52%
MandreelLat	14491	13848	4.44%
GB Emu	38697	36548	5.55%
CodeLoad	10566	8082	23.51%
Box2DWeb	24158	24302	-0.6%
zlib	48055	50136	-4.33%
Typescript	14200	13484	5.04%
Total	17616	17137	2.72%

Table 3: In our second real-world test we start Firefox as a physically isolated user process. We measure the performance impact within Firefox using the widely used browser benchmarking tools Kraken and Octane. These test show a high variance in the results, hence, the negative performance overhead has to be accounted to measurement inaccuracy.

A DETAILED EVALUATION RESULTS

File Size	Plain 4.10	RIP-RH (isolated)	OH
512MB	3.59s	3.71s	3.34%
1GB	10.33s	10.24s	-0.87%
10GB	143.89s	148.83s	3.44%

Table 1: Our real-world test runs OpenSSH as an physically isolated server process while transfering different file sizes over the local network.

LMBench3	RIP-RH
File & VM Latency:	
1K File Create	-3.39%
1K File Delete	4.54%
10K File Create	1.49%
10K File Delete	3.00%
Mmap Latency	0.80%
Mean	1.29%
Local Bandwidth:	
Bcopy (libc)	1.12%
Bcopy (hand)	1.62%
Mem read	-3.88%
Mem write	2.18%
Mean	0.26%
Memory Latency:	
Stride 16	0.37%
Stride 32	-0.08%
Stride 64	0.77%
Stride 128	-2.53%
Rand mem	4.85%
Mean	0.68%

Table 2: Our microbenchmarks using LMBench show that apart from a small setup overhead during process initialization RIP-RH incurs no measurable memory or run-time overhead.

Phoronix	Plain	RIP-RH (none)	RIP-RH (all isolated)	OH (none)	OH (all isolated)
IOZone (HIB)	25011.10	25817.16	24813.43	-3.2%	0.79%
PostMark (HIB)	3363	3318	3055	1.34%	0.88%
7-Zip (HIB)	7768	7756	6717	0.15%	13.39%
OpenSSL (HIB)	137.8	137.7	137.97	0.07%	-0.12%
PyBench (LIB)	3227	3215	3220	-0.37%	-0.22%
Apache (HIB)	14868.43	15097.01	15190.69	-1.53%	-2.17%
PHPBench (HIB)	237096	234941	233913	0.91%	1.34%
stream (HIB)	41492.82	41477.74	41550.9	0.04%	-0.14%
ramspeed (HIB)	88084.72	87790.59	87269.41	0.29%	0.89%
cachebench (HIB)	34407.55	34338.04	34351.52	0.20%	0.16%
Mean				-0.22%	1.48%
Geomean				-1.08%	1.31%

Table 4: The benchmarking results for Phoronix indicate that RIP-RH induces no significant performance overhead with all tests started as *critical* processes. In some cases, we observed a negative performance overhead, i.e., an improvement. We attribute such results to measurement inaccuracies.