GuardION: Practical Mitigation of DMA-Based Rowhammer Attacks on ARM

Victor van der Veen[1(✉)], Martina Lindorfer[3], Yanick Fratantonio[4], Harikrishnan Padmanabha Pillai[2], Giovanni Vigna[3], Christopher Kruegel[3], Herbert Bos[1], and Kaveh Razavi[1]

[1] Vrije Universiteit, Amsterdam, The Netherlands
{vvdveen,herbertb,kaveh}@cs.vu.nl
[2] Amrita University, Coimbatore, India
hpadmanabhapillai@gmail.com
[3] University of California, Santa Barbara, USA
martina@iseclab.org, {vigna,chris}@cs.ucsb.edu
[4] EURECOM, Biot, France
yanick.fratantonio@eurecom.fr

Abstract. Over the last two years, the Rowhammer bug transformed from a hard-to-exploit DRAM disturbance error into a fully weaponized attack vector. Researchers demonstrated exploits not only against desktop computers, but also used single bit flips to compromise the cloud and mobile devices, all without relying on any software vulnerability.

Since hardware-level mitigations cannot be backported, a search for software defenses is pressing. Proposals made by both academia and industry, however, are either impractical to deploy, or insufficient in stopping all attacks: we present RAMPAGE, a set of DMA-based Rowhammer attacks against the latest Android OS, consisting of (1) a root exploit, and (2) a series of app-to-app exploit scenarios that bypass all defenses.

To mitigate Rowhammer exploitation on ARM, we propose GUARDION, a lightweight defense that prevents DMA-based attacks—the main attack vector on mobile devices—by isolating DMA buffers with guard rows. We evaluate GUARDION on 22 benchmark apps and show that it has a negligible memory overhead (2.2 MB on average). We further show that we can improve system performance by re-enabling higher order allocations after Google disabled these as a reaction to previous attacks.

1 Introduction

For decades, defensive research on memory corruption could brush aside the threat of exploitation via hardware bugs as "outside the threat model," if not science fiction entirely. The frightening list of devastating Rowhammer attacks, however, published at one security venue after another [5,12,16,24,28,30], suggests that we are in urgent need of practical defenses. Anything new? In this paper, we propose a practical, isolation-based protection that stops DMA-based Rowhammer attacks by carefully surrounding DMA buffers with DRAM-level

C. Giuffrida et al. (Eds.): DIMVA 2018, LNCS 10885, pp. 92–113, 2018.
https://doi.org/10.1007/978-3-319-93411-2_5

guard rows. We focus our work on mobile devices as here, the problem is even more worrisome: unlike desktop and server machines, it is impossible to perform hardware upgrades.

Rowhammer on Mobile Devices. The Rowhammer hardware bug at its core consists of the leakage of charge between adjacent memory cells on a densely packed DRAM chip [19]. Thus, whenever the CPU reads or writes one row of bits in the DRAM module, the neighboring rows are ever so slightly affected. Normally, this does not create problems as DRAM periodically refreshes the charge in its cells, well in time to preserve data integrity. However, an attacker who deliberately hits the same rows many times within a refresh interval may cause the charge leakage to accumulate to the point that a bit flips in an adjacent row and modify memory that she does not own. Initially considered a curiosity of relatively minor importance, researchers have shown that attackers can harness Rowhammer to completely subvert a system's security [5,8,16,24,26,28,30].

Clearly, the threat of Rowhammer attacks for smartphones and tablets is particularly serious, as replacing the memory chips of such devices is not an option. In addition, power consumption is a prime concern in the mobile world, and many of the hardware-level solutions (such as ECC memory or higher DRAM refresh rates) consume more power. Furthermore, even though newer standards such as LPDDR4 [18] discuss the adoption of Rowhammer mitigations, i.e., Target Row Refresh (TRR), they do so only as an *optional* protection mechanism, thus making LPDDR4 chips vulnerable as well [20,28].

Existing Software Defenses are Not Effective. Given the challenges of deploying hardware solutions, the development of effective software-based defenses is particularly important to protect mobile users against Rowhammer attacks. In our analysis, we systematically explore existing proposals, which fall into two categories: techniques that attempt to prevent attackers from triggering bit flips, and those that focus on making it impossible for a bit flip to bring physical memory into an exploitable state (Sect. 4). We argue that both directions have limitations, either in terms of practicality (for instance because they require specific hardware features), or worse, in terms of effectiveness (as they still allow for Rowhammer exploitation). We demonstrate this ineffectiveness by presenting novel attacks that circumvent all existing proposed and implemented defense techniques (Sect. 5).

The Need for *Practical* Solutions. Security solutions need to strike a balance between security and practicality—a defense against Rowhammer attacks should not incur unacceptable performance overhead, nor should it severely reduce the amount of available memory. Conversely, it should be effective and hard to bypass. In this work, we propose GUARDION, which effectively and efficiently blocks all known DMA-based Rowhammer attacks against mobile devices (Sect. 6).

GUARDION builds on the observation that triggering bit flips on ARM-based mobile platforms is facilitated by using uncached memory, accessible through DMA allocations [28]. Albeit other techniques exist, most are either impractical or easily addressable on ARM. For example, the `cacheflush()` system call that is exposed to userland by the Android kernel, only flushes up to the Level 2 cache,

and thus fails to force repetitive DRAM accesses for a single address. Additionally, ARMv8's unprivileged cache flush instruction can easily be disabled by the kernel and thus do not pose a security risk.

We thus explicitly limit our defense to the more generic class of DMA-based Rowhammer attacks that rely on uncached memory. Doing so has an important implication for our design: instead of attempting to isolate all sensitive information, which is impractical, we can instead isolate only DMA allocations. As we will show, DMA allocations constitute only a very small fraction of all allocations in the system, and we can hence afford to apply expensive fine-grained isolation for *each* DMA allocation using guard rows. In our design, we isolate DMA allocations from the rest of the system by using two guard rows, one at the top and another at the bottom. With this scheme, an attacker can no longer use DMA allocations to trigger bit flips in any memory page in the system except in the guard rows. In effect, this design defends against Rowhammer by eradicating the ability to inject bit flips in sensitive data.

Can GuardION Defend Against Any Rowhammer Exploit? No. GUARDION only enforces that DMA-based Rowhammer attacks can no longer flip bits in another process or kernel memory. Attacks that induce bit flips by means of cache eviction sets—another popular Rowhammer technique on x86—are still possible. The (1) lento, and (2) idiosyncratic nature of these attacks, however, make them harder to launch in practice. First, increased access times will result in less flipped bits at a slower rate. Second, a substantial amount of reverse engineering is required for such attacks, and this work must be repeated for each target architecture [12,28]. Thus, although not stopping all possible attacks, GUARDION reduces the attack surface significantly.

Contributions. In summary, we make the following contributions:

- We systematically explore the design of software defenses, and show that existing proposals are either not practical or not effective.
- To back our claims, we present RAMPAGE, a set of DMA-based Rowhammer attack variants on ARM. RAMPAGE consists of (1) a root exploit, and (2) a series of app-to-app attacks.
- We introduce GUARDION, a software-based defense that prevents DMA-based Rowhammer attacks. GUARDION is simple, efficient, and has low memory overhead.

In the spirit of open science, we provide our modifications to the Android source code for implementing GUARDION at https://github.com/vusec/guardion.

2 Threat Model

We consider an attacker with full control over a zero-permissions holding, unprivileged Android app that is running on the victim's device. She seeks to mount a DMA-based Rowhammer attack, similar to recent work [28], to either (1) escalate her privileges to root, or (2) compromise other apps present on the device. The victim device is hardened against other classes of Rowhammer attacks (e.g., GLitch [12]) and has the latest Android security updates installed.

3 Background

This section describes the relevant background information about the Rowhammer vulnerability and its exploitation. This is meant to provide only a brief introduction, for a more in-depth discussion, we point the interested reader to papers exclusively focusing on this topic [19,22,30].

3.1 The Rowhammer Vulnerability

Rowhammer is a hardware fault in dynamic random-access memory (DRAM) chips. DRAM chips work by storing charges in an array of *cells*. The charge state of a given cell encodes a binary value, a memory bit. Cells are organized in *rows*, which, at the hardware level, is the smallest unit for a memory access. When a memory row is accessed, the content of its cells is copied to a so-called *row buffer*. During this copy operation, the row's cells are discharged, and they are then recharged with their initial values.

Independently from the row access process, memory cells tend to leak their charged state (due to their nature), and their content thus needs to be refreshed regularly. Kim et al. [19] observed that the increasing density of memory chips makes them prone to disturbance errors due to charge leaking into adjacent cells on every memory access. In particular, they show that, by repeatedly accessing, i.e., "hammering," the same memory row (the aggressor row), an attacker can cause enough of a disturbance in a neighboring row (the victim row) to cause bits to flip. The Rowhammer vulnerability is thus a race against the DRAM memory refresh: if an attacker can cause sufficient disturbance, the refresh process may not be fast enough to recharge the cells with their initial values. Kim et al. show that it is possible to flip bits in memory by solely performing software-induced memory read operations, bypassing common memory isolation mechanisms.

3.2 Rowhammer Exploitation

Triggering the Rowhammer bug is different than exploiting it. Most bits in memory are irrelevant for an attacker, as flipping them would often just trigger a memory corruption, without obtaining any concrete security advantage. For successful exploitation, the attacker must first land a security-sensitive memory page (e.g., a page owned by the operating system or by another privileged process) into a vulnerable physical memory page. In the general case, software exploitation of this kind is challenging, and, as outlined by van der Veen et al. [28], requires the implementation of the following primitives:

Fast Uncached Memory Access. The attacker must access the DRAM chip "fast enough." One of the biggest challenges here is bypassing CPU caches, which, if not handled properly, would "block" any read attempt. The attacker thus needs to either flush them (to make sure that the next memory read access propagates to DRAM), or use uncached DMA memory to bypass CPU caches altogether.

Physical Memory Massaging. For successful exploitation, the attacker must land a security-sensitive page into a physical memory location that is vulnerable to Rowhammer. This entails that the attacker somehow massages the physical memory so that she can probabilistically [26] or deterministically [24] determine where security-sensitive memory pages would land in physical memory.

Physical Memory Addressing. To make Rowhammer exploitation more practical, the attacker can mount a so-called double-sided Rowhammer attack in which the victim row gets hammered by not one, but both adjacent rows. While this increases the chances of triggering bit flips, it is more challenging: the attacker must either be able to allocate physically contiguous memory, or determine how virtual addresses of an unprivileged process are mapped to physical addresses. In other words, the attacker must determine which virtual addresses map to the two physical rows adjacent to the victim row. We note that, while this primitive is not strictly necessary to implement Rowhammer attacks, its implementation is often required to make these attacks practical.

Security researchers demonstrated that a variety of different system environments are vulnerable to Rowhammer exploitation. Seaborn and Dullien [26] were the first to demonstrate two practical attacks: one to gain local privilege escalation, and another to escape native client sandboxes. Other researchers then used Rowhammer to bypass in-browser JavaScript sandboxes [5,16], and even to perform cross-VM attacks [24,30]. While most work focuses on the x86 platform, Van der Veen et al. show that also ARM-based mobile devices are vulnerable to the Rowhammer bug [28]. This last attack, Drammer, is the most problematic as it does not rely on any special hardware or software features. It shows that it is possible to mount a deterministic privilege escalation technique by relying only on basic memory management functions available in typical modern operating systems that cannot easily be turned off.

3.3 Android Memory Management

Android, as any other Linux platform, manages physical memory via the buddy allocator, whose goal is to minimize memory fragmentation [15]. In addition, starting from Android 4.0, Google introduced ION [31], a high-level interface that aims at replacing and unifying the several memory management interfaces exposed by each hardware manufacturer. One of the main features implemented by ION is a number of DMA Buffer Management APIs, which allows userland apps to obtain uncached memory. ION organizes its memory pools in several in-kernel heaps, such as the *kmalloc heap* (`SYSTEM_CONTIG`) and the *system heap* (`SYSTEM`). These heaps allocate memory at different memory locations and, in general, behave differently. For example, van der Veen et al. [28] observed how an app can use the *kmalloc heap* to obtain physically contiguous memory (now disabled by Google [14]), while this is not possible when using the *system heap*.

4 Overview of Software-Based Rowhammer Defenses

Proposed software-level Rowhammer mitigations try to (1) prevent Rowhammer from triggering bit flips, or (2) prevent massaging of physical memory into an exploitable state (i.e., bit flips in security-sensitive data structures). We now discuss these defenses in more detail and expose their limitations in terms of practicality—*What are the limitations for deploying this technique in practice?*—and security—*Does this technique stop all attacks?* Table 1 summarizes our discussion and shows that no previous solution is both practical *and* secure.

Table 1. Summary of existing defenses and their limitations when deployed to prevent DMA-based Rowhammer attacks on ARM

Class	Defense	Practical	Secure
¬flips	ANVIL [4]	✗	✓[a]
	B-CATT [6]	✗	✗
	Disabling the contiguous heap [14]	✓	✗
	Pool size reduction [14]	✓	✗
¬massage	CATT [7]	✗	✗
	Separation of lowmem/highmem [14]	✓	✗
	Our approach (GUARDION)	✓	✓

[a]Assuming a modified implementation that monitors DRAM accesses instead of cache misses.

4.1 Preventing Bit Flips (¬flips)

ANVIL [4] is a two-step mitigation technique that relies on the processor's performance monitoring unit (PMU) to (1) monitor last-level cache misses (LLC misses). If the number of LLC misses per time period exceeds a predefined value, it marks the offending load/store instructions as a potential Rowhammer attack. It then (2) instructs the PMU to also record virtual addresses accessed, and data sources used by those instructions. ANVIL analyzes the results of the latter, and, if it concludes that a Rowhammer attack is ongoing, it accesses neighboring rows to force an early refresh, effectively preventing any bit from flipping.

ANVIL could prevent DMA-based Rowhammer attacks by monitoring DRAM accesses instead of LLC misses. Such a defense would be secure, as it would successfully prevent bits from flipping. We were unsuccessful, however, in our search for PMU features on ARM that allow an efficient implementation of ANVIL's second stage: we were unable to locate any feature that allows us to keep track of which virtual or physical addresses are read from or written to. As such, we conclude that ANVIL is impractical as a mitigation against DMA-based Rowhammer attacks.

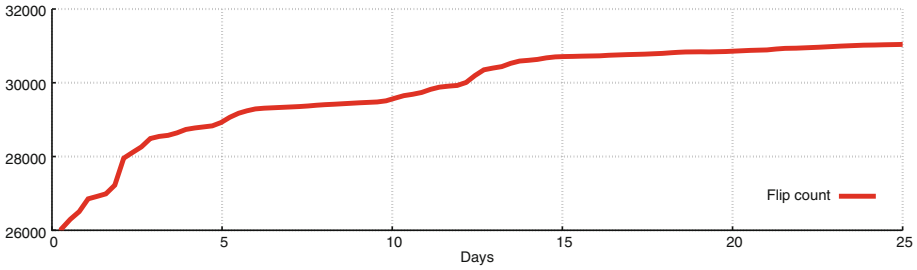

Fig. 1. Number of unique bit flips found while repeatedly hammering the same 4 MB chunk using double-sided Rowhammer on a Nexus 5 over a time-period of 25 days. Results were obtained by using Drammer's source code [3].

B-CATT [6] instructs the bootloader to run a Rowhammer test over the entire physical memory to identify memory pages with vulnerable cells. It then instructs the operating system to mark these pages as unavailable, forcing the system to never use them. This effectively removes the ability of an attacker to induce bit flips.

To evaluate B-CATT's security guarantees, we ran an experiment in which we search for bit flips on a Nexus 5 device by repeatedly performing double-sided Rowhammer on the same 4 MB chunk of contiguous memory. We ran our test for little less than a month while keeping track of each bit flip. Figure 1 shows our results: it depicts the number of unique bit flips over time, where a bit flip is unique if the physical address at which it was reported has not been flipped during an earlier round. Our results show that unique flips indeed do increase over time, and proves that mitigations based on blacklisting vulnerable memory, such as B-CATT, do not scale and are inherently insecure. Furthermore, it shows that any technique that relies on observations and thresholds derived during a testing period, can be subverted as the attacker can trigger different bit flips during runtime. This is on par with related work that discusses the importance of different bit patterns unique to every device when hammering [20,25]), which makes generic hammering techniques not always effective in finding all vulnerable memory regions.

In parallel, there are many issues that make B-CATT impractical. First, since devices may average close to one bit flip per page [28,30], B-CATT would have to disable *all* of physical memory for those. Second, blacklisted pages contribute to physical memory fragmentation, making it harder—or impossible—for apps that require physically contiguous memory to run properly. Third, doing a single sweep of a devices's physical memory may take over a day to complete—as we experienced when scanning the entire 4 GB of LPDDR4 memory of a Google Pixel, which is in line with observations of related work [30].

Disabling the Contiguous Heap was Google's first reaction to Drammer [28] and is a third defense that tries to prevent the attacker from flipping bits. In the November 2016 security update for Android, Google disabled the *kmalloc heap*, removing an attacker's primitive to allocate contiguous memory [14]. Without access to the `pagemap` interface—a special file in `procfs` for retrieving physical addresses—this update effectively disables an attacker's ability of performing double-sided Rowhammer, greatly reducing the number of bits she can flip [5].

As this was Google's first attempt at mitigating Drammer, disabling the contiguous heap is proven to work in practice on a variety of devices. We will show in Sect. 5, however, that it is not secure: it possible to implement primitives for obtaining contiguous memory allocations even when using the regular *system heap* (which does not guarantee the allocation of contiguous memory). In concurrent work, Frigo et al. [12] present another side channel for detecting contiguous memory.

Pool Size Reduction was part of a second round of Drammer mitigations by Google which reduced the number of internal *system heap* pools to two. Before, ION could allocate and pool memory using many different pool sizes (4 KB, 8 KB, 16 KB, ..., 4 MB). If one requested a large chunk of memory, say 4 MB, from an empty pool, ION would request these large chunks from the underlying allocator directly, increasing the likelihood for an attacker to obtain physically contiguous memory. By reducing the maximum pool size to 64 KB, the attacker is more likely to obtain fragmented memory pieces that are not physically contiguous.

Although this is a proven practical solution, it does not eradicate the problem at its root. We show in Sect. 5 how a determined attacker can still force the system to allocate contiguous memory to launch double-sided Rowhammer. Moreover, we show how limiting system allocations to low orders (up to 64 KB) is not effective when memory is not heavily fragmented. In fact, a request for 200 MB would get split up in many 64 KB allocations, some of which will very likely be allocated right next to each other in physical memory.

4.2 Preventing Physical Memory Massaging (¬massage)

CATT proposes a static partitioning of physical memory between different security domains [6]. In principle, its design allows for an arbitrary *finite* number of security domains. However, only a prototype for the special case of two security domains was implemented and evaluated: *lowmem* (kernel memory) and *highmem* (user memory). By design, this system guarantees that, under any circumstance, the kernel never touches userland memory, and vice versa.

Modern operating system kernels are designed to make all possible resources available to an app or to the kernel itself. For example, in Linux, the memory management code moves physical memory between zones (e.g., highmem and lowmem) to alleviate memory pressure in them, as a function of the current workload. Due to its *static* partition, CATT severely limits this capability, making it unlikely to be used in practice. Moreover, as acknowledged by the authors,

the generalization of CATT to more than two security domains presents a number of significant practicality and complexity challenges. For example, to prevent app-to-app attacks that we will discuss in Sect. 5, CATT must enable as many domains as there are apps installed. To support this, the prototype must be able to pass additional arguments to the kernel's memory allocator to specify the security domain of the process requesting the allocation. This would result in memory fragmentation, which would in turn lead, among other problems, to performance issues: the memory allocator must scan the memory to find a "suitable" memory region for each memory allocation.

Not only is CATT impractical, recent work also demonstrates that so-called *double-ownership* kernel buffers (e.g., video buffers that are shared between user and kernel) allow an attacker to bypass CATT's security guarantees [9].

Separation of Highmem/Lowmem was part of Google's mitigations against Drammer. Android now enforces that the *system heap*—which is exposed to userland apps—only returns memory pages from highmem, separating attacker-controlled memory for critical data structures in lowmem.

The highmem/lowmem separation suffers from the same issues as described before. Additionally, we show in the next section that an attacker can allocate many ION chunks to deplete the highmem pool and force the kernel to serve new requests from lowmem. Thus, despite Android's latest security updates, an unprivileged app can still force the system to allocate userland pages in lowmem.

5 RAMpage: Breaking the State-of-the-Art

This section elaborates on the security limitations of existing defenses as discussed in the previous section. We document new attack strategies, showing that the defense mechanisms that appear to be practical are not effective for preventing Rowhammer attacks. We first show how it is possible to mount Rowhammer-based attacks even when ION memory allocations are not contiguous and served from highmem. Next, we discuss several app-to-app attack scenarios that show kernel-owned data is not the only target memory to protect.

5.1 Exploiting Non-contiguous Memory

Before Drammer, the ION subsystem allowed userland apps to allocate a large number of contiguous chunks. As described previously, to mitigate Drammer, Google disabled the ION *kmalloc heap*. The ION *system heap*, however, is still available. This heap has two features that make the Drammer attack more challenging: (1) ION allocations from this heap are no longer guaranteed to be physically contiguous, preventing attackers from performing double-sided Rowhammer; (2) the system heap allocates memory from a different zone (highmem, as opposed to lowmem for the kmalloc heap).

We now detail our first RAMPAGE variant, R0: a reliable Drammer implementation that shows how disabling contiguous memory allocations does not prevent Rowhammer-based privilege escalation attacks.

Exhausting the System Heap. We observe that once ION's internal pools are drained, subsequent allocations are handled directly by the buddy allocator. In this state, we rely on the predictable behavior of the buddy allocator to get contiguous pages [28]. With access to contiguous chunks of memory, we then perform double-sided Rowhammer to find exploitable bit flips. However, as mentioned, the system heap initially allocates memory from highmem while the interesting data structures reside in the lowmem zone. To force lowmem allocations, we simply continue allocating memory until no highmem is left (which we detect by monitoring `procfs`). Once this is the case, the kernel serves subsequent requests from lowmem, allowing us to find bit flips in physical memory that may later hold a page table.

Shrinking the Cache Pool. Armed with an exploitable bit flip in lowmem, we perform *Phys Feng Shui* to trick the kernel in storing a page table in the vulnerable page. For this, we need to free the vulnerable row so that the buddy allocator may use it as a page table later. Simply releasing the chunk, however, is not sufficient: after freeing, it ends up in the ION memory pool. We thus require a primitive to shrink system heap pools.

On Android, the low-memory killer (LMK) [29] handles low-memory conditions that arise in the system before the more severe Linux Out-of-Memory (OOM) killer is triggered. The LMK works similarly to the OOM killer, but keeps track of additional information, such as various shrinkers that are available. Shrinkers are registered by memory subsystems or drivers that reserve an amount of memory from the system RAM [1]. When the system is close to running out of memory, the LMK calls the registered shrinkers to release and regain cached memory.

We now construct a primitive to release physical memory of the system heap pools back to the kernel: (1) we read from `/proc/meminfo` to learn how much free memory is available and use this to (2) trigger a `mmap` allocation from userland which is large enough to trigger the LMK. This indirectly forces the ION subsystem to release its preallocated cached memory, including the row with the vulnerable page.

Rooting a mobile device. By combining our primitives with the *Phys Feng Shui* methodology of Drammer [28], we implement the remaining steps of the attack (i.e., finding exploitable chunks and landing page tables in vulnerable locations) and develop a root exploit. We were successful in mounting our proof of concept against a LG G4 running the latest version of Android (7.1.1. at the time of our experiments).

The implementation of these steps involves solving a number of engineering challenges that, from a conceptual point of view, are similar to what was presented in Drammer. We report the details of our attack in Appendix A.

5.2 Exploiting System-Wide Isolation

In this section, we detail how defense solutions that only protect specific parts of system memory (e.g., the CATT prototype) do not provide a comprehensive protection mechanism. We present three more RAMPAGE variants that illustrate how one can bypass these defenses.

ION-to-ION (R1). In this scenario, we use ION allocations to corrupt ION buffers that belong to another app or process. We start with allocating ION memory to search for exploitable bit flips. Next, we release the vulnerable page to which this bit belongs so that our victim may reuse it. Depending on our victim process, we must then either wait for it to allocate ION memory, or we can trigger allocations by sending an app-specific *intent*.

To investigate the feasibility of this attack, we developed a proof-of-concept in which we trigger bit flips in ION memory that is in use by a victim process. During a real attack, an attacker will likely target a privileged app, such as the media server, in which case she must investigate what bits are sensitive to flip. We acknowledge that it may not be trivial to perform such an attack, and we believe this to be an interesting direction for future research. However, we argue that this scenario and our proof of concept provide a concrete example showing how current defense mechanisms are not comprehensive enough.

CMA-to-CMA Attack (R2). The Contiguous Memory Allocator (CMA) is another kernel mechanism to implement DMA-like primitives [10,11,21] and thus provides another venue for attackers. Mounting CMA attacks is technically more challenging since it uses a bit map for deciding allocations: depending on the internal state of the bit map, the victim may not get the same chunk of memory that the attacker releases after the templating, i.e., the probing for vulnerable memory locations. However, the attacker can exhaust the CMA bit map before releasing the vulnerable range to make sure that the victim will reuse the attacker's vulnerable target chunk.

CMA-to-System Attack (R3). Although challenging, it is also possible to corrupt system memory from CMA-allocated memory, leading to our last RAMPAGE variant. In fact, the buddy allocator is designed to migrate pages from the CMA heap when the system is close to out-of-memory situations. These pages can be claimed back at any time by the CMA heap (in other words, they are moveable). We note that this attack cannot directly target page tables (because they are unmovable); however, the attacker might be able to target other sensitive system-owned data structures (e.g., `struct cred`).

6 GuardION: Fine-Grained Memory Isolation

As discussed, the main reason for which defenses fail in practice is because they aim to protect *all* sensitive information by making sure that they are not affected by Rowhammer bit flips. Hence, they are either impractical or they miss

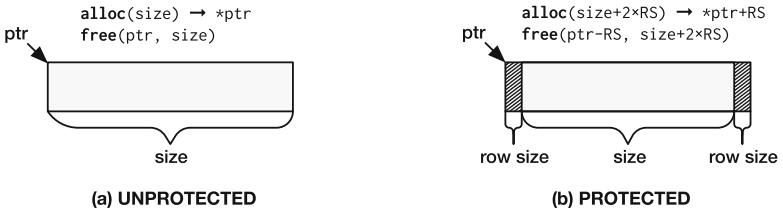

Fig. 2. Allocations on ION's contiguous heap (a) without and (b) with GUARDION (RS = row size).

cases (e.g., variants R1–R3). As RAMPAGE shows, ARM devices are still widely exposed, and providing an adequate software protection is particularly pressing.

We propose GUARDION, a mitigation against DMA-based Rowhammer exploits on mobile devices. Instead of trying to protect all physical memory, we focus on limiting the capabilities of an attacker's uncached allocations. As we will show in Sect. 7, these only constitute a small fraction of all allocations in the system. We can hence afford to apply expensive fine-grained isolation for *each* DMA allocation. GUARDION isolates such buffers with two *guard rows*, one at the 'top' (the first n bytes before an allocation), and another at the 'bottom' (n additional bytes starting at the last address of the allocation). This enforces a strict containment policy in which bit flips that are triggered by reading from uncached memory cannot occur outside the boundaries of that DMA buffer. In effect, this design defends against Rowhammer by eradicating the ability of the attacker to inject bit flips in sensitive data.

Note that GUARDION works under the assumption that bit flips never occur in memory pages that are physically more than one row 'away' from the aggressor rows. This is in the same spirit as other defenses and we believe a sane assumption: such flips have never been reported before, and the electrical properties of Rowhammer make this unlikely to ever occur. Additionally, our current prototype assumes that physical addresses are linearly mapped to DRAM addresses. While this is true for most ARM-based chipsets [22,28], a next version of GUARDION should use a kernel allocator that is aware of DRAM geometry.

We now describe our implementation of this fine-grained isolation for the Android kernel. Specifically, we modify three allocators that potentially hand contiguous uncached memory to userland apps: the ION contiguous heap, the ION system heap, and the contiguous memory allocation heap (i.e., the CMA heap). In all cases, we need modifications in the allocation and deallocation routines, which we now discuss in more detail.

6.1 Isolating ION's Contiguous Heap

Google disabled ION's contiguous heap, the *kmalloc heap* (`SYSTEM_CONTIG`), as part of their efforts to thwart the Drammer attack. This was possible since most devices do not require physically contiguous memory allocations to be available for regular userland apps. Device configurations that do require this, however,

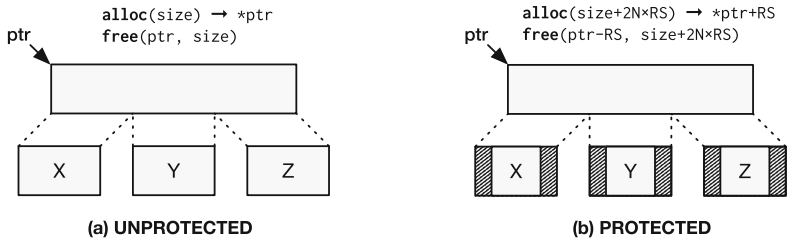

Fig. 3. Allocations on ION's system heap (a) without and (b) with GUARDION (N = number of pool members, size = X + Y + Z, RS = row size).

remain exposed. Since isolating these allocations is simple, we explore them first before describing our more elaborate efforts for isolating ION's system and CMA heaps.

For each request, the contiguous heap allocator takes the requested size and computes the smallest buddy order that satisfies this. The allocator then requests the required number of pages from that buddy order. To free a previously allocated buffer, the allocator simply returns the pages back to the buddy allocator. To isolate bit flips in these buffers, we allocate two guard rows that sandwich the allocation as shown in Fig. 2. At allocation time for a given size s, we request two extra rows, i.e., $s + 2 \times RS$ (RS being the row size), and return the buffer starting after the guard row to the user. Note that the user process will not have access to these guard rows, as they are never mapped to virtual memory.

For this to work, we need to round up the allocation size to at least the row size. Hence, to protect a 4 KB buffer, we need to allocate 3×64 KB (assuming a row size of 64 KB). Fortunately, at runtime, many requested buffers have a larger size, amortizing the overhead of guard rows. Further, given that DMA buffers constitute a small fraction of an entire app's memory, this overhead becomes negligible as we will show in Sect. 7.

6.2 Isolating ION's System Heap

There are two main limitations with ION's contiguous heap: (1) it is not possible to satisfy requests if physically contiguous memory is not available due to fragmentation, and (2) the interaction with the buddy allocator is expensive. To address these limitations, the *system heap* (`SYSTEM`) provides its users with virtually contiguous memory backed by memory pools of various sizes.

Figure 3(a) shows how the system heap satisfies an allocation of a given size by stitching multiple smaller physically contiguous allocations together. These smaller allocations are satisfied from pools with pre-defined sizes. The system heap makes an attempt to use pools of the largest suitable size before resorting to pools with smaller sizes to reduce management overhead for each allocation. These pools act as a cache of the buddy allocator in order to improve the allocation performance. Whenever the system is under memory pressure, free memory from these pools is reclaimed and given back to the buddy allocator.

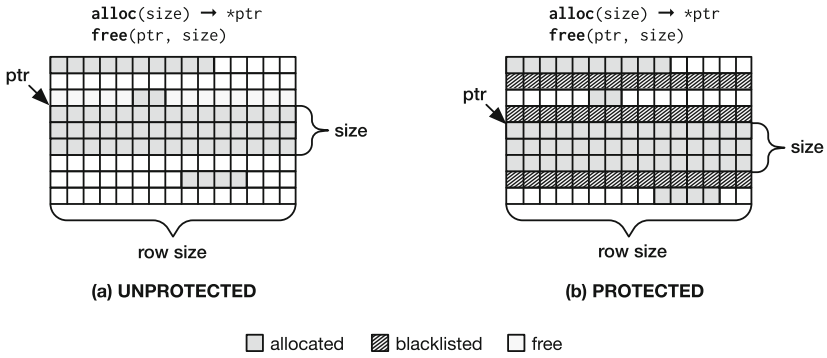

Fig. 4. Allocations on ION's CMA heap (a) without and (b) with GUARDION.

Currently, in order to thwart Drammer, Android only enables pools of size 4 KB and 64 KB, instead of previously-supported larger sizes. Since the size of a memory row is usually 64 KB on ARM, it was expected that attackers cannot allocate large-enough physically contiguous buffers to perform the templating step reliably. We showed that this is not the case in Sect. 4 and we now discuss how we can protect the system heap with GUARDION.

We extend our design for the contiguous heap to protect each physically contiguous allocation from each pool, as depicted in Fig. 3(b). We extended the pool allocation and deallocation routines to isolate every pool member, similar to how we isolated each allocation from the contiguous heap. Our modifications are mostly straightforward, given that we do not introduce an additional state in the pool allocator. During `free` operations, we return the extra guard rows back to the system.

The overhead of isolating uncached allocations in the system heap depends on the number of allocations from the pools for each request. Given that we are isolating each sub-allocation, we now safely re-enable pools with larger sizes. On top of reducing per-allocation management overhead in the system heap, enabling larger pools reduces the overhead of GUARDION given that it reduces the number of sub-allocations for each request.

6.3 Isolating ION's CMA Heap

While it was possible to disable ION's contiguous heap for newer mobile devices, there are still drivers that may require physically contiguous memory allocations. These allocations, however, mostly happen in the kernel and are handled by the *CMA heap* (`CMA`). The CMA heap has a statically-defined size which is reserved in physical memory at boot time. These pages may be used by other users when necessary but can always be claimed back by the kernel. While the CMA heap is currently only used by the kernel, we found that recent Android versions still expose it to unprivileged apps. Although we did not find any userland app on a Google Pixel that requires it, we still implement isolation for this heap to provide complete protection.

Figure 4(a) shows how the CMA allocator handles requests using a bit map that tracks free memory in the CMA region. The CMA allocator scans this bit map to find the first fit for a requested allocation size. This means that over time, this bit map gets fragmented. To provide isolation in this heap, we follow the following strategy: we blacklist all odd rows in the bit map during initialization. This provides isolation for all allocations that are smaller than the size of the row. To support allocations larger than the row size, we scan the bit map to find a first fit assuming we can allocate blacklisted rows. We use a secondary bit map for the rows to keep track of odd rows that are allocated as part of a large allocation and maintain it during free operations of these large allocations. Figure 4(b) shows an isolated allocation from the CMA heap with GUARDION in place.

7 Evaluation

We now evaluate GUARDION under several aspects: security, performance, and ease of adoption.

7.1 Security Evaluation

GUARDION provides an isolation primitive that makes it impossible for attackers to use uncached DMA allocations to flip bits in memory that is in use by the kernel or any userland app. Within our threat model, where attacks are only possible by attacking uncached memory, GUARDION protects all known Rowhammer attack vectors, and, to the best of our knowledge, no existing technique can bypass it. We verify this by mounting the exploits detailed in Sect. 5 which all failed: we were unable to flip bits in the memory of another process.

7.2 Performance and Memory Footprint

We now evaluate the overhead of GUARDION, focusing on both performance and memory overhead.

Dataset. To evaluate GUARDION, we execute 22 Android benchmark apps that we selected as follows: (1) we built a dataset of 135 benchmark apps, obtained by searching for the *benchmark* keyword on Google Play; (2) by profiling the ION subsystem, we found that only 28 of them use uncached DMA memory; (3) we discarded two of them as they perform the same tests, one because it does not produce a score, and three because they do not produce reproducible numbers for our baseline.

We run each benchmark app thrice on a Google Pixel running Android 7.1.1 without GUARDION (baseline) and reboot the device after each execution. We then enable GUARDION and repeat this experiment. We compute the median over the three runs and use this as the benchmark score. Since some benchmarks report higher scores for better performance, while for others a lower score indicates better performance, we normalize the scores across benchmarks. Finally, we calculate the geometric mean (geomean) over all benchmark results.

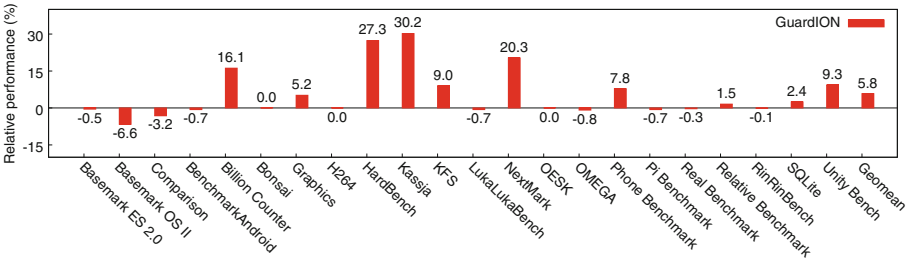

Fig. 5. Performance results of applying GUARDION. The numbers show the relative improvement (positive number) or degradation (negative number) of the performance according to each benchmark app. The last column reports the geometric mean of these results (5.8%), which shows that the performance generally increases.

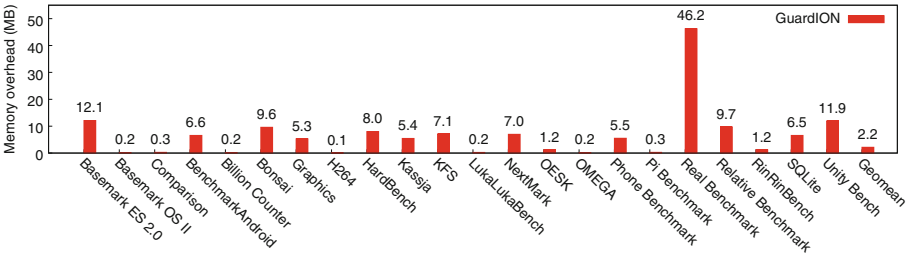

Fig. 6. Memory overhead of DMA isolation with GUARDION. The numbers show the overhead in MB. The last column reports the geometric mean of these results (2.2 MB), which indicates that the memory overhead is, in the general case, negligible.

Performance Overhead. Figure 5 shows the overall performance impact of GUARDION. In particular, the figure shows the relative performance compared to the baseline, where a positive value indicates an improvement, while a negative value indicates a performance degradation.

In the worst case, GUARDION results in a performance degradation of 6.6%, which we believe is still acceptable. The geomean, however, indicates a performance improvement of 5.8%, likely caused by the fact that GUARDION allows us to revert Google's second series of patches, which reduced pool size as outlined in Sect. 4. This allows the ION subsystem to use higher order allocations again in case a process files a request for DMA memory larger than 64 KB: for example, the Kassja benchmark triggers many 2 MB uncached allocations. With large order allocations enabled, each such request triggers the ION subsystem to call the underlying page allocator only once, at most (if the request cannot be processed by the pool). Without GUARDION, however, the request would get split up in $\frac{2\,MB}{64\,KB} = 32$ allocations, each one introducing additional overhead.

Memory Overhead. Figure 6 shows the memory overhead of GUARDION. We determine the memory footprint of an app by modifying the ION subsystem

to log every allocation and free operation, including the kernel affected virtual addresses. This way, we can map each allocation to its associated free operation.

The geomean of the memory overhead is 2.2 MB, which is negligible, especially when considering that modern devices usually have at least 2 GB of RAM. Interestingly, the RealBench app shows a significant overhead of 46.2 MB, which is much higher than the average. Upon investigation, we determine that this is because the app pressures the uncached DMA, allocating about 190.2 MB during the test.

Impact on UI Performance and Everyday Usage. Google measures UI performance of apps mainly in terms of frames per second (fps) and number of "janky," i.e., delayed or dropped, frames [13]. A consistent rate of 60 fps is considered ideal for smooth UI animations. Android provides measurements of these values for a specific app through `adb shell dumpsys gfxinfo <packagename>`. We evaluate the performance of web browsing with Google Chrome on the Google Pixel with and without GUARDION and aggregated the results [2]. Averaged over five runs each, we did not notice any significant differences in UI performance: 22.4 out of 266.8 (8.4%) and 22.0 out of 279.0 (7.9%) frames were janky without and with GUARDION, respectively, with the frame rate remaining constant at 60 fps.

Finally, we evaluated the impact of GUARDION on day-to-day device use. For this test, we performed several "everyday" operations on a Google Pixel with and without GUARDION. These operations include taking a photo, shooting and watching a video, watching a video through the YouTube app, making a phone call, making a video call through the Skype app, and browsing the web with Chrome. In all scenarios, we did not notice any difference. Moreover, we did not notice any sign of slowdown or instability.

7.3 Patch Complexity and Adoption

We believe that it is easy to integrate GUARDION with the current Android code base. In particular, our prototype implementation for Android 7.1.1 consists of only 844 lines of code. The patch is mostly contained in the ION subsystem, and adds functionality to the bit map data structure of the kernel. Touching only 9 files in the Android source code, it is thus well contained. We also note that a significant part of our patch (422 lines, 5 files) relates to augmenting the bit map data structure to protect the CMA heap, which we believe should not be exposed to userland apps in the first place (thus possibly reducing the size of our patch even more).

We are currently in the process of submitting our patch to Google, and we hope that Google adopts our proposal either as a security patch for existing versions or in newer versions of Android.

8 Related Work

This section provides an overview of related work in the field of Rowhammer exploitation and prevention that were not yet discussed in Sects. 3.2 and 4.

8.1 Rowhammer Attacks

After Kim et al. [19] performed the first systematic study on the Rowhammer hardware fault, and Seaborn and Dullien [26] demonstrated the first practical attacks, Qiao and Seaborn [23] showed how to use non-temporal access instructions such as `movnti`, to bypass the cache (instead of relying on `clflush`, which was disabled in the Google Native Client browser sandbox following the first Rowhammer attacks). Aweke et al. [4] showed that it is possible to trigger bit flips without using special instructions: they show how an attacker can force the cache to invalidate its content by accessing memory addresses belonging to the same cache eviction set. Another recent technique abuses Intel's Cache Allocation Technology (CAT) to reduce the number of active ways in the last-level cache, which in turn significantly decreases the number of memory accesses required to trigger a bit flip [27]. As discussed in Sect. 3.2, related work has also shown how the Rowhammer vulnerability can be exploited in a number of different scenarios [5,16,24,28,30].

8.2 Rowhammer Defenses

The aforementioned attacks have demonstrated the severity of the Rowhammer vulnerability and prompted the research community to propose a number of defense mechanisms, both in hardware and in software.

Hardware-Level Defenses. One of the most obvious defense mechanism is the production of memory chips that do not suffer from the Rowhammer vulnerability. Kim et al. [19] discuss the various aspects that could be improved: for example, one could increase the row refresh rate. The DDR3 standard [17] specifies that rows should be refreshed at least every 64 ms, while Kim et al. suggest to refresh the rows at least every 32 ms. Other proposals are Error-Correcting Code (ECC) memory and Target Row Refresh (TRR). One last protection mechanism is PARA [19], which probabilistically activates rows adjacent to a potential victim row.

Unfortunately, all these techniques have significant limitations. First, many of these techniques rely on hardware modifications: ECC and TRR require the production of new memory chips, while PARA requires a change in the memory controller. This makes their deployment less practical, mainly because these chips would be more expensive, would require new development and production pipelines, and they cannot be easily adopted by existing systems, especially mobile devices. Moreover, newer standards such as LPDDR4 [18] already discuss the adoption of TRR, but only as an *optional* protection mechanism, thus leaving LPDDR4 chips still vulnerable to Rowhammer. Protecting mobile devices through hardware protection mechanisms is even more challenging due to the energy consuming nature of these mechanisms and the importance minimizing the device's battery consumption.

Second, these mechanisms are not always effective, even when deployed. For example, Aweke et al. [4] show that they could perform Rowhammer exploitation under 32 ms, making the faster refresh rate ineffective. Instead, mechanisms

like ECC memory have the limitations of protecting only from one-bit memory corruption, which is not enough since Kim et al. could induce multiple bit flips. These limitations provided strong incentives to develop software-level defenses.

Software-Level Defenses. The research community has started to propose software-based solutions only recently. The first concrete solution is ANVIL [4], which we discussed in Sect. 4. Unfortunately, ANVIL is not applicable to mobile devices. Furthermore, we discussed B-CATT and CATT [6,7], as well as Google's patches in reaction to Drammer, in Sect. 4, and demonstrated why they are not effective in Sect. 5.

9 Conclusion

In recent years, the Rowhammer vulnerability gathered a lot of attention from both the academic and industrial community. While researchers have demonstrated exploits for a range of devices in a variety of settings [5,16,19,24,26,28,30], the Drammer attack on mobile devices [28] is particularly worrying, since it allows for a deterministic attack on very popular systems, by just relying on basic memory management features. Given that it is impossible to perform hardware upgrades on these devices, there is a clear need for effective and efficient software-based defenses.

In this paper, we showed that existing software mitigations do not solve the problem: they are either impractical to deploy, or do not provide adequate protection. To back our claims, we presented RAMPAGE, a set of DMA-based Rowhammer attacks against the latest Android OS. As a mitigation, we proposed GUARDION, a lightweight, software-only defense to prevent Rowhammer exploitation on mobile devices. Our evaluation shows that GUARDION introduces negligible memory overhead, improves performance compared to Google's mitigation in reaction to previous attacks, and prevents all known DMA-based Rowhammer attacks, even when considering app-to-app attacks. We release our modifications as open source, and are in the process of sharing our patches with Google, hoping they will adopt our proposal in newer versions of Android.

Acknowledgments. We thank the anonymous reviewers for their valuable comments and input to improve the paper, as well as Pietro Frigo for his help on understanding GLitch.

This work was supported by the Netherlands Organisation for Scientific Research through grants NWO CSI-DHS 628.001.021, by the European Commission through project H2020 ICT-32-2014 "SHARCS" under Grant Agreement No. 644571, the NSF under Award No. CNS-1408632, the ONR under Award No. N00014-17-1-2897, DARPA under agreement number FA8750-15-2-0084, and a Security, Privacy and Anti-Abuse award from Google. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views, position, official policies, or endorsements, either expressed or implied, of the U.S. Government, DARPA, ONR, NSF, or Google.

A RAMpage Attack Details

This section provides additional details on RAMPAGE variant R0, our end-to-end exploit that bypasses Google's deployed defenses against Drammer. We rely on primitives and steps discussed in Sect. 5: how to exhaust the *system heap*, how to shrink the cache pool, and how to trigger the Low Memory Killer (LMK). We now discuss the remaining steps: how to find exploitable chunks, how to land a page table in the vulnerable page, and how to obtain root access.

With the help of the primitives defined in Sect. 5, we can allocate memory chunks directly from the buddy allocator. We exploit the deterministic behavior of buddy to obtain large contiguous chunks of memory that we then template for exploitable bit flips. Typically, Android ARMv8 devices are configured to use 3 levels of translation tables with 4 KB pages, resulting in 39 bits that are available for virtual addressing. Page table entries are 64 bits wide, but since most devices are shipped with only 4 GB of RAM or less, half of those bits are never used. We thus ignore those during the templating phase.

The exploitation steps for RAMPAGE R0 now involve the following sequence:

1. Exhaust ION memory page pools.
2. Monitor `/proc/pagetypeinfo` and allocate chunks using ION's `SYSTEM` heap with large orders that span at least 3 or more physical rows, e.g., chunks of at least 256 KB if the row size is 64 KB. As soon as ION's internal pools are drained, we will see each of these large order allocations immediately affecting `pagetypeinfo`. From this point, each subsequent request for (large) ION chunks is likely to be contiguous as they are being served by buddy directly.
3. Optionally, to confirm that allocated chunks are contiguous, we could either (1) perform double-sided Rowhammer to check if there are any flips, or (2) use the bank-conflict side-channel [12].
4. Template the memory using double-sided Rowhammer to find an exploitable page.
5. Perform *Phys Feng Shui* so that the large chunk that contains the vulnerable page is split in multiple smaller chunks (of the row size) that we can release individually [28].
6. Confirm that the aggressor rows are still accessible by performing a second templating round.
7. Release the vulnerable row. This will give it back to the ION cache.
8. Perform a *cache pool shrink* operation. This will release memory from all registered shrinkers—including the ION cache—back to buddy.
9. Perform page table spraying while monitoring `/proc/pagetypeinfo` until the first chunk of the row size is touched.
10. Allocate page tables until all chunks of the row size are used.

Once page tables using row size chunks are allocated, we could set those page tables with values that point it back to itself when hammered. Once we are able to get access to the page table, we scan the kernel memory for `struct cred` bytes and overwrite the UID's to that of root.

References

1. Low-Memory Shrinker API, October 2013. http://www.phonesdevelopers.info/1815288. Accessed 5 May 2017
2. cookie-butter: Python Script for Making Graphics Performance Charts for an Android App (2016). https://github.com/Turnsole/cookie-butter
3. Drammer: Native Binary for Testing Android Phones for the Rowhammer Bug (2016). https://github.com/vusec/drammer
4. Aweke, Z.B., Yitbarek, S.F., Qiao, R., Das, R., Hicks, M., Oren, Y., Austin, T.: ANVIL: software-based protection against next-generation Rowhammer attacks. In: Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (2016)
5. Bosman, E., Razavi, K., Bos, H., Giuffrida, C.: Dedup Est Machina: memory deduplication as an advanced exploitation vector. In: Proceedings of IEEE Symposium on Security and Privacy (S&P) (2016)
6. Brasser, F., Davi, L., Gens, D., Liebchen, C., Sadeghi, A.R.: CAn't touch this: practical and generic software-only defenses against Rowhammer attacks, November 2016. arXiv:1611.08396 [cs.CR]
7. Brasser, F., Davi, L., Gens, D., Liebchen, C., Sadeghi, A.R.: Can't touch this: practical and generic software-only defenses against Rowhammer attacks. In: Proceedings of USENIX Security Symposium (2017)
8. Cai, Y., Ghose, S., Luo, Y., Mai, K., Mutlu, O., Haratsch, E.F.: Vulnerabilities in MLC NAND flash memory programming: experimental analysis, exploits, and mitigation techniques. In: Proceedings of International Symposium on High-Performance Computer Architecture (HPCA) (2017)
9. Cheng, Y., Zhang, Z., Nepal, S.: Still hammerable and exploitable: on the effectiveness of software-only physical kernel isolation, February 2018. arXiv:1802.07060 [cs.CR]
10. Corbet, J.: Contiguous Memory Allocation for Drivers, July 2010. https://lwn.net/Articles/396702/
11. Corbet, J.: A Reworked Contiguous Memory Allocator, June 2011. https://lwn.net/Articles/447405/
12. Frigo, P., Giuffrida, C., Bos, H., Razavi, K.: Grand Pwning unit: accelerating microarchitectural attacks with the GPU. In: Proceedings of IEEE Symposium on Security and Privacy (S&P) (2018)
13. Google: Testing UI Performance. https://developer.android.com/training/testing/performance.html
14. Google: ion: Disable ION_HEAP_TYPE_SYSTEM_CONTIG, November 2016. https://android.googlesource.com/device/google/marlin-kernel/
15. Gorman, M.: Understanding the Linux Virtual Memory Manager. Prentice Hall PTR, Upper Saddle River (2007)
16. Gruss, D., Maurice, C., Mangard, S.: Rowhammer.js: a remote software-induced fault attack in JavaScript. In: Proceedings of Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA) (2016)
17. JEDEC Solid State Technology Association: DDR3 SDRAM Specification. JESD79-3F (2012)
18. JEDEC Solid State Technology Association: Low Power Double Data 4 (LPDDR4). JESD209-4A (2015)

19. Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J.H., Lee, D., Wilkerson, C., Lai, K., Mutlu, O.: Flipping bits in memory without accessing them: an experimental study of DRAM disturbance errors. In: Proceedings of International Symposium on Computer Architecture (ISCA) (2014)
20. Lanteigne, M.: How Rowhammer Could Be Used to Exploit Weaknesses in Computer Hardware, March 2016. http://www.thirdio.com/rowhammer.pdf
21. Nazarewicz, M.: A Deep Dive into CMA, March 2012. https://lwn.net/Articles/486301/
22. Pessl, P., Gruss, D., Maurice, C., Schwarz, M., Mangard, S.: DRAMA: exploiting DRAM addressing for cross-CPU attacks. In: Proceedings of USENIX Security Symposium (2016)
23. Qiao, R., Seaborn, M.: A new approach for Rowhammer attacks. In: Proceedings of IEEE International Symposium on Hardware Oriented Security and Trust (HOST) (2016)
24. Razavi, K., Gras, B., Bosman, E., Preneel, B., Giuffrida, C., Bos, H.: Flip Feng Shui: hammering a needle in the software stack. In: Proceedings of USENIX Security Symposium (2016)
25. Schaller, A., Xiong, W., Salee, M.U., Anagnostopoulos, N.A., Katzenbeisser, S., Szefer, J.: Intrinsic rowhammer PUFs: leveraging the Rowhammer effect for improved security. In: Proceedings of IEEE International Symposium on Hardware Oriented Security and Trust (HOST) (2017)
26. Seaborn, M., Dullien, T.: Exploiting the DRAM Rowhammer bug to gain kernel privileges. In: Black Hat USA (BH-US) (2015)
27. Aga, M.T., Aweke, Z.B., Austin, T.: When good protections go bad: exploiting anti-DoS measures to accelerate Rowhammer attacks. In: Proceedings of IEEE International Symposium on Hardware Oriented Security and Trust (HOST) (2017)
28. van der Veen, V., Fratantonio, Y., Lindorfer, M., Gruss, D., Maurice, C., Vigna, G., Bos, H., Razavi, K., Giuffrida, C.: Drammer: deterministic Rowhammer attacks on mobile platforms. In: Proceedings of ACM Conference on Computer and Communications Security (CCS) (2016)
29. Vorontsov, A.: Android Low Memory Killer vs. Memory Pressure Notifications, December 2011. https://lkml.org/lkml/2011/12/18/173
30. Xiao, Y., Zhang, X., Zhang, Y., Teodorescu, M.R.: One bit flips, one cloud flops: cross-VM Rowhammer attacks and privilege escalation. In: Proceedings of USENIX Security Symposium (2016)
31. Zeng, T.M.: The Android ION Memory Allocator, February 2012. https://lwn.net/Articles/480055