

Drammer: Deterministic Rowhammer Attacks on Mobile Platforms

Victor van der Veen
Vrije Universiteit Amsterdam
vvdveen@cs.vu.nl

Yanick Fratantonio
UC Santa Barbara
yanick@cs.ucsb.edu

Martina Lindorfer
UC Santa Barbara
martina@iseclab.org

Daniel Gruss
Graz University of Technology
gruss@tugraz.at

Clémentine Maurice
Graz University of Technology
cmaurice@tugraz.at

Giovanni Vigna
UC Santa Barbara
vigna@cs.ucsb.edu

Herbert Bos
Vrije Universiteit Amsterdam
herbertb@cs.vu.nl

Kaveh Razavi
Vrije Universiteit Amsterdam
kaveh@cs.vu.nl

Cristiano Giuffrida
Vrije Universiteit Amsterdam
giuffrida@cs.vu.nl

ABSTRACT

Recent work shows that the Rowhammer hardware bug can be used to craft powerful attacks and completely subvert a system. However, existing efforts either describe probabilistic (and thus unreliable) attacks or rely on special (and often unavailable) memory management features to place victim objects in vulnerable physical memory locations. Moreover, prior work only targets x86 and researchers have openly wondered whether Rowhammer attacks on other architectures, such as ARM, are even possible.

We show that *deterministic* Rowhammer attacks are feasible on commodity *mobile platforms* and that they cannot be mitigated by current defenses. Rather than assuming special memory management features, our attack, DRAMMER, solely relies on the predictable memory reuse patterns of standard physical memory allocators. We implement DRAMMER on Android/ARM, demonstrating the practicability of our attack, but also discuss a generalization of our approach to other Linux-based platforms. Furthermore, we show that traditional x86-based Rowhammer exploitation techniques no longer work on mobile platforms and address the resulting challenges towards practical mobile Rowhammer attacks.

To support our claims, we present the first Rowhammer-based Android root exploit relying on *no software vulnerability*, and requiring *no user permissions*. In addition, we present an analysis of several popular smartphones and find that many of them are susceptible to our DRAMMER attack. We conclude by discussing potential mitigation strategies and urging our community to address the concrete threat of faulty DRAM chips in widespread commodity platforms.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CCS'16, October 24–28, 2016, Vienna, Austria.

© 2016 ACM. ISBN 978-1-4503-4139-4/16/10...\$15.00

DOI: <http://dx.doi.org/10.1145/2976749.2978406>

1. INTRODUCTION

The Rowhammer hardware bug allows an attacker to modify memory without accessing it, simply by repeatedly accessing, i.e., “hammering”, a given physical memory location until a bit in an adjacent location flips. Rowhammer has been used to craft powerful attacks that bypass all current defenses and completely subvert a system [16,32,35,47]. Until now, the proposed exploitation techniques are either probabilistic [16,35] or rely on special memory management features such as memory deduplication [32], MMU paravirtualization [47], or the `pagemap` interface [35]. Such features are often unavailable on commodity platforms (e.g., all are unavailable on the popular Amazon EC2 cloud, despite recent work explicitly targeting a cloud setting [32,47]) or disabled for security reasons [40,46]. Recent JavaScript-based attacks, in turn, have proven capable to reliably escape the JavaScript sandbox [11], but still need to resort to probabilistic exploitation to gain root privileges and to completely subvert a system [16].

Probabilistic Rowhammer attacks [16,35] offer weak reliability guarantees and have thus more limited impact in practice. First, they cannot reliably ensure the victim object, typically a page table in kernel exploits [16], is surgically placed in the target vulnerable physical memory location. This may cause the Rowhammer-induced bit flip to corrupt unintended data (rather than the victim page table) and crash the whole system. Second, even when the victim page table is corrupted as intended, they cannot reliably predict the outcome of such an operation. Rather than mapping an attacker-controlled page table page into the address space as intended, this may cause the Rowhammer-induced bit flip to map an unrelated page table, which, when modified by the attacker, may also corrupt unintended data and crash the whole system.

This paper makes two contributions. First, we present a *generic* technique for *deterministic* Rowhammer exploitation using *commodity* features offered by modern operating systems. In particular, we only rely on the predictable behavior of the default physical memory allocator and its memory reuse patterns. Using this technique (which we term

Phys Feng Shui), we can reliably control the layout of physical memory and deterministically place security-sensitive data (e.g., a page table) in an attacker-chosen, vulnerable physical memory location.

Second, we use the aforementioned technique to mount a deterministic Rowhammer attack (or DRAMMER) on mobile platforms, since they present different and unexplored hardware and software characteristics compared to previous efforts, which focus only on x86 architectures, mainly in desktop or server settings. Concerning the hardware, mobile platforms mostly use ARM processors. However, all known Rowhammer techniques target x86 and do not readily translate to ARM. Moreover, researchers have questioned whether memory chips on mobile devices are susceptible to Rowhammer at all or whether the ARM memory controller is fast enough to trigger bit flips [13, 35]. Concerning the software, mobile platforms such as Android run different and more limited operating systems that implement only a subset of the features available in desktop and server environments. For example, unless explicitly specified by a device vendor, the Android kernel does currently not support huge pages, memory deduplication, or MMU paravirtualization, making it challenging to exploit the Rowhammer bug and impossible to rely on state-of-the-art exploitation techniques.

DRAMMER is an instance of the *Flip Feng Shui (FFS)* exploitation technique (abusing the physical memory allocator to surgically induce hardware bit flips in attacker-chosen sensitive data) [32], which for the first time relies only on always-on commodity features. For any Rowhammer-based *Flip Feng Shui* attack to be successful, three primitives are important. First, attackers need to be able to “hammer sufficiently hard”—hitting the memory chips with high frequency. For instance, no bits will flip if the memory controller is too slow. Second, they need to find a way to massage physical memory so that the right, exploitable data is located in the vulnerable physical page. Third, they need to be able to target specific contiguous physical addresses to achieve (i) double-sided Rowhammer [9, 35], a technique that yields more flips in less time than other approaches, and (ii) more control when searching for vulnerable pages (important when mounting deterministic attacks). We show that, when attacking mobile platforms, none of these primitives can be implemented by simply porting existing techniques.

In this paper, we present techniques to implement aforementioned primitives when attacking mobile platforms. We detail the challenges towards reliable exploitation on Android/ARM and show how to overcome its limited feature set by relying on DMA buffer management APIs provided by the OS. To concretely demonstrate the effectiveness of our DRAMMER attack on mobile platforms, we present the first deterministic, Rowhammer-based Android *root* exploit. Our exploit can be launched by any Android app with no special permission and without relying on any software vulnerability.

Finally, we present an empirical study and assess how widespread the Rowhammer bug is on mobile devices. We investigate how fast we can exploit these bugs in popular smartphones and identify multiple phones that suffer from faulty DRAM: 17 out of 21 of our tested 32-bit ARMv7 devices—still the most dominant platform with a market share of over 97% [44]—and 1 out of our 6 tested 64-bit ARMv8 phones are susceptible to Rowhammer. We con-

clude by discussing how state-of-the-art Rowhammer defenses are ineffective against our DRAMMER attack and describe new mitigation techniques.

In summary, we make the following contributions:

- We present the first technique to perform *deterministic* Rowhammer exploitation using only *commodity* features implemented by modern operating systems.
- We demonstrate the effectiveness of our technique on mobile platforms, which present significant hardware and software differences with respect to prior efforts. We implement our DRAMMER attack on Android/ARM and present the first deterministic, Rowhammer-based Android root exploit. Our exploit cannot be mitigated by state-of-the-art Rowhammer defenses.
- We evaluate the effectiveness of DRAMMER and our Android root exploit and complement our evaluation with an empirical Rowhammer study on multiple Android devices. We identify multiple ARMv7/ARMv8 smartphones that suffer from faulty DRAM.
- To support future research on mobile Rowhammer, we release our codebase as an open source project and aim to build a public database of known vulnerable devices.¹

2. THREAT MODEL

We assume that an attacker has control over an unprivileged Android app on an ARM-based device and wants to perform a privilege escalation attack to acquire root privileges. We do not impose any constraints on the attacker-controlled app or the underlying environment. In particular, we assume the attacker-controlled app has no permissions and the device runs the latest stock version of the Android OS with all updates installed, all security measures activated, and no special features enabled.

3. ROWHAMMER EXPLOITATION

Rowhammer is a software-induced hardware fault that affects dynamic random-access memory (DRAM) chips. In practice, this has the net effect that a piece of software can flip some bits in physical memory by solely performing memory read operations. It is important to note that triggering the Rowhammer bug is different than using (i.e., exploiting) it in a security-relevant manner. In fact, an exploit usually needs to trick a victim component (e.g., another process, the OS, or another VM hosted on the same physical node) to use a vulnerable physical memory location to store security-sensitive content. In the general case, software exploitation of this kind proved to be challenging.

In this section, we first provide general background information on memory hardware and the Rowhammer bug. Then, we summarize existing exploitation techniques and describe the three distinct primitives that Rowhammer exploits need to implement.

¹<https://www.vusec.net/projects/drammer/>

3.1 Memory Hardware

In order to understand the root cause of the Rowhammer bug, it is important to understand the architecture and components of DRAM chips. DRAM works by storing charges in an array of *cells*, each of which consists of a capacitor and an access transistor. A cell represents a binary value depending on whether it is charged or not. Cells are further organized in *rows*, which are the basic unit for memory accesses. On each access, a row is “activated” by copying the content of its memory cells to a *row buffer* (thereby discharging them), and then copying the content back to the memory cells (thereby charging them). A group of rows that is serviced by one row buffer is called a *bank*. Finally, multiple banks further form a *rank*, which spans across multiple *DRAM chips*. A *page frame* is the smallest fixed-length contiguous block of physical memory into which the OS maps a *memory page* (a contiguous block of virtual memory). From a DRAM perspective, a page frame is merely a contiguous collection of memory cells, aligned on the page-size boundary (typically 4 KB).

Memory cells naturally have a limited retention time and leak their charge over time. Therefore, they have to be refreshed regularly in order to keep their data. Thus, the DDR3 standard [19] specifies that the charge of each row has to be refreshed at least every 64 ms. This memory refresh interval is a trade-off between memory integrity on the one hand, and energy consumption and system performance on the other. Refreshing more often consumes more power and also competes with legitimate memory accesses, since a specific memory region is unavailable during the refresh [10].

3.2 The Rowhammer Bug

In a quest to meet increasing memory requirements, hardware manufacturers squeeze more and more cells into the same space. Unfortunately, Kim et al. [23] observed that the increasing density of current memory chips also makes them prone to disturbance errors due to charge leaking into adjacent cells on every memory access. They show that, by repeatedly accessing, i.e., “hammering,” the same memory row (the aggressor row) over and over again, an attacker can cause enough of a disturbance in a neighboring row (the victim row) to cause bits to flip. Thus, triggering bit flips through Rowhammer is essentially a race against the DRAM internal memory refresh in performing enough memory accesses to cause sufficient disturbance to adjacent rows. Relying on activations of just one aggressor row to attack an adjacent row is called *single-sided Rowhammer*, while the more efficient *double-sided Rowhammer* attack accesses the two rows that are directly above and below the victim row [35].

3.3 Exploitation Primitives

While it was originally considered mostly a reliability issue, Rowhammer becomes a serious security threat when an attacker coerces the OS into storing security-sensitive data in a vulnerable memory page (a virtual page that maps to a page frame consisting of at least one cell that is subject to the Rowhammer bug). Depending on the underlying hardware platform, OS, and already-deployed countermeasures, prior efforts developed different techniques to perform a successful end-to-end Rowhammer attack. This section summarizes prior techniques and describes the three required primitives to exploit the Rowhammer bug.

P1. Fast Uncached Memory Access. This primitive is the prerequisite to flip bits in memory and refers to the ability of activating rows in each bank *fast enough* to trigger the Rowhammer bug. In practice, this can be non-trivial for two reasons. First, the CPU memory controller might not be able to issue memory read commands to the memory chip fast enough. However, most often the challenge relates to the presence of several layers of caches, which effectively mask out all the CPU memory reads (after the first one). Thus, all known exploitation techniques need to implement a mechanism to bypass (or nullify) the cache.

P2. Physical Memory Massaging. This primitive consists of being able to trick the victim component to use—in a *predictable* or, in a weaker form, *probabilistic* way—a memory cell that is subject to the Rowhammer bug. More importantly, the attacker needs to be able to *massage* the memory precisely enough to push the victim to use the vulnerable cell to store security-sensitive data, such as a bit from a page table entry. This primitive is critical for an attacker to mount a privilege escalation attack, and it is also the most challenging one to implement in a fully *deterministic* way.

P3. Physical Memory Addressing. This last primitive relates to understanding how physical memory addresses are used in the virtual address space of an unprivileged process. While this primitive is not a hard requirement for Rowhammer exploitation in general, it is crucial to perform double-sided Rowhammer: to access memory from two aggressor rows, an attacker must know which virtual addresses map to the physical addresses of these rows.

4. THE FIRST FLIP

This section documents our efforts to perform Rowhammer on memory chips of mobile devices. In the first part, we focus on testing the hardware without attempting to mount any exploit. In the second part, we discuss how going from “flipping bits” to mounting a root privilege escalation attack is challenging, since there are several aspects that make successful exploitation on mobile devices fundamentally different compared to previous efforts on desktop and server settings. To support our claims, we compare all known techniques to implement the three primitives discussed in the previous section and we discuss why these techniques do not apply to commodity mobile platforms.

4.1 RowhARMer

Researchers have speculated that Rowhammer on ARM could be impossible, one of the main reasons being that the ARM memory controller might be *too slow* to trigger the Rowhammer bug [13, 35]. Not surprisingly, no existing work from academia or industry currently documents any success in reproducing the Rowhammer bug on mobile devices.

We set up an experiment to test whether memory chips used in mobile devices are subject to bit flips induced by Rowhammer and whether the ARM memory controller can issue *memory read operations* fast enough. Since this is a preliminary feasibility analysis, we perform the Rowhammer attack from a kernel module (i.e., with full privileges), which allows us to cultivate optimal conditions for finding bit flips: we disable CPU caching and perform double-sided Rowhammer by using the `pagemap` interface to find aggressor rows for each victim address.

We hammer rows by performing one million read operations on their two aggressors. To determine the minimum memory access time that still results in bit flips, we repeatedly hammer the same 5 MB of physical memory while artificially increasing the time between two read operations by inserting NOP instructions. We measure the time it takes to complete a single read operation and report the median over all hammered pages. We initiate all bytes in the victim row to `0xff` (all bits are set) and once the hammering phase finishes, we scan the victim row for bit flips—i.e., any byte that has a value different than `0xff`. Since we only perform *read* operations, any such modification to the memory content can be directly attributed to Rowhammer.

For this experiment, we used an LG Nexus 5 device running Android 6.0.1 (the latest version at the time of writing). The results of this experiment are encouraging: not only do bits flip, but it is also relatively simple to obtain them. In fact, we triggered flips in a matter of seconds, and observed up to 150 flips per minute. Figure 1 depicts the results of our experiment. It shows the dependency between the access time and the number of bit flips found when scanning a 5 MB memory chunk. Moreover, it shows that access times of 300 ns or higher are unlikely to trigger bit flips and that, surprisingly, the “sweet spot” for triggering the most flips on this particular DRAM chip is not reading at full speed (70 ns per read), but throttling to 100 ns per read. However, note that throttling does not necessarily result in a lower rate of actual accesses to DRAM cells: the memory controller may reorder accesses internally.

4.2 Exploitation on the x86 Architecture

Even when a memory chip is vulnerable to Rowhammer attacks, it is challenging to perform a successful end-to-end exploitation—we need to implement the three primitives described earlier. We now review how currently known Rowhammer exploitation techniques, which all target the x86 architecture, implement those primitives.

4.2.1 P1. Fast Uncached Memory Access

First and foremost an attacker needs the capability to activate alternating rows in each bank *fast enough* to trigger the Rowhammer bug. The main challenge here is to bypass the CPU cache. For this purpose, state-of-the-art attacks rely on one of the following techniques:

Explicit cache flush. This technique is based on using the `clflush` instruction, which flushes the cache entry associated to a given address. One can execute this instruction after accessing a particular memory address, so that subsequent *read* operations on that same address also trigger DRAM accesses. On x86 architectures, the `clflush` instruction is particularly useful because it can be executed even by a non-privileged process. This technique is used, for example, by Seaborn et al. [35,36] and is based on the findings from Kim et al. [23].

Cache eviction sets. This technique relies on repeatedly accessing memory addresses that belong to the same *cache eviction set* [9,11,16]. A cache eviction set is defined as a set of *congruent* addresses, where two addresses are congruent if and only if they *map* to the same cache line. Thus, accesses to a memory address belonging to the same congruent set will automatically flush the cache while reading (because the associated cache line contains the content of

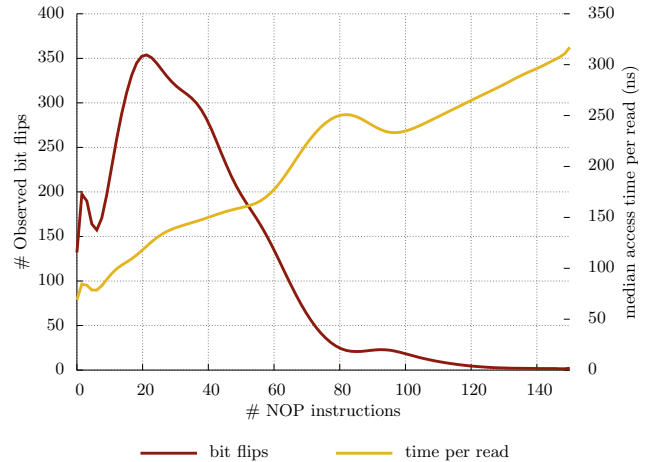


Figure 1: Dependency between observed bit flips (y_1) and memory access time (y_2) when repeatedly hammering the same 5 MB memory chunk, while increasing the number of NOP instructions (x) to simulate slower access times. The spike in access time around 60 to 80 NOP instructions may be caused by a background process during our analysis. The spike in observed bit flips around 20 NOP instructions indicates a “sweet spot” of memory access time.

the previously-read memory address). This observation is the basis for the several access patterns described by Gruss et al. [16] and is particularly useful when the `clflush` instruction is not available (e.g., when triggering Rowhammer from JavaScript).

Non-temporal access instructions. This technique relies on accessing memory using CPU instructions or APIs that, by design, do not use the cache. Previous efforts rely on non-temporal write instructions (e.g., `MOVNTI`, `MOVNTDQA`), which are also used in some `memset()` and `memcpy()` implementations [31]. In this context, *non-temporal* means that the data will not likely be reused soon and thus does not have to be cached. As a result, these operations cause the CPU to directly write the content to memory, thus bypassing the cache.

4.2.2 P2. Physical Memory Massaging

This primitive is essential to *trick* the victim component into storing security-sensitive data (e.g., a page table) in an attacker-chosen, vulnerable physical memory page. Existing efforts have mainly relied on the following techniques for this purpose:

Page-table spraying. Previous work exploits the Rowhammer bug to achieve root privilege escalation by flipping bits in page table entries (PTEs) [35,36]. This attack suggests a *probabilistic* exploitation strategy that *sprays* the memory with page tables, hoping that at least one of them lands on a physical memory page vulnerable to Rowhammer. The next step is then to flip a bit in the vulnerable physical memory page, so that the victim page table points to an arbitrary physical memory location. Given the sprayed physical memory layout, such location should *probabilistically* contain one of the attacker-controlled page table pages (PTPs) which allows attackers to map their own page tables in the

controlling address space. At that point, they can overwrite their own PTEs and access arbitrary (e.g., kernel) pages in physical memory to escalate privileges.

Memory deduplication. Razavi et al. [32] abuse memory deduplication to perform deterministic Rowhammer exploitation [11,32]. They show that an attacker can use memory deduplication to trick the OS into mapping two pages, an attacker-controlled virtual memory page and a victim-owned virtual memory page, to the same attacker-chosen vulnerable physical memory page. While such an exploitation strategy is powerful and has been successfully demonstrated in a cross-VM setting, it relies on memory deduplication, which is not an always-on feature, even in modern operating systems (e.g., off by default on Linux).

MMU paravirtualization. Xiao et al. [47] leverage Xen MMU paravirtualization to perform deterministic Rowhammer exploitation from a guest VM. This exploits the property that Xen allows a guest VM to specify the physical location of a (read-only) PTP, allowing a malicious VM to trick the VM monitor into mapping a page table into a vulnerable location to “hammer.” Similar to memory deduplication, this is not an always-on feature and only available inside Xen MMU paravirtualized VMs. In addition, MMU paravirtualization is no longer the common case in popular cloud settings, with MMU virtualization becoming much more practical and efficient.

4.2.3 P3. Physical Memory Addressing

Processes have direct access only to virtual memory which is then mapped to physical memory. While the virtual memory layout is known to processes in userland, the physical memory layout is not. As discussed in the previous section, to perform double-sided Rowhammer, an attacker needs to repeatedly access specific physical memory pages. For this purpose, previous efforts suggest the following techniques:

Pagemap interface. This technique relies on accessing the `/proc/self/pagemap` file which contains *complete* information about the mapping of virtual to physical addresses. Clearly, having access to this information is sufficient to repeatedly access specific rows in physical memory.

Huge pages. Another option is to use *huge (virtual) pages* that are backed by physically contiguous physical pages. In particular, a huge page covers 2MB of contiguous physical addresses. Although this is not as fine-grained as knowing absolute physical addresses, one can use relative offsets to access specific physical memory pages for double-sided Rowhammer. In fact, it guarantees that two rows that are contiguous in virtual memory are also contiguous in physical memory.

4.3 Challenges on Mobile Devices

When assessing whether one can exploit Rowhammer bugs on mobile devices, we attempted, as a first step, to reuse known exploitation techniques described above. We found, however, that none of the primitives are applicable to mobile devices. Table 1 presents an overview of our analysis.

Explicit cache flush (P1). On ARMv7, the cache flush instruction is privileged and thus only executable by the kernel. Since our threat model assumes an unprivileged app, we cannot use this instruction to implement P1. Although the Android kernel exposes a `cacheflush()` system call to

Table 1: Techniques previously used for x86-based Rowhammer attacks and their availability on ARM-based mobile devices in unprivileged mode (●), privileged mode (○), or not at all (–). Some techniques are available in unprivileged mode, but are not practical enough to use in our setting (◐). Note how none of these techniques can be generally applied on all modern versions of mobile devices.

Primitive	x86 Platforms	Mobile Devices
<i>Fast Uncached Memory Access</i>		
Explicit cache flush	●	○/◐
Cache eviction sets	●	–/–
Non-temporal access instructions	●	–/◐
<i>Physical Memory Massaging</i>		
Page-table spraying	●	◐
Memory deduplication	●	–
MMU paravirtualization	●	–
<i>Physical Memory Addressing</i>		
Pagemap interface	●	○
Huge pages	●	–

userland, this system call flushes only up to the Level 2 cache and thus fails to force repetitive DRAM accesses for a single address. Interestingly, ARMv8 does provide unprivileged cache flush instructions, but they may be disabled by the kernel.

Cache eviction sets (P1). In principle, it is possible to use cache eviction sets to flush addresses from the cache. Unfortunately, this technique proved to be too slow in practice to trigger bit flips on both ARMv7 and ARMv8.

Non-temporal access instructions (P1). ARMv8 offers non-temporal load and store instructions, but they only serve as a hint to the CPU that caching is not useful [8]. In practice, we found that memory still remains cached, making these instructions not usable for our exploitation goals.

Page-table spraying (P2). As documented by Seaborn et al. [36], the page-table spraying mechanism is probabilistic and may crash the OS. We aim to implement a deterministic Rowhammer exploit and thus cannot rely on this technique.

Special memory management features (P2). Although device vendors may enable memory deduplication for *Low RAM* configurations [1], it is not enabled by default on stock Android. Moreover, MMU paravirtualization is not available and we can thus not rely on existing special memory management features.

Pagemap interface (P3). The Linux Kernel no longer allows unprivileged access to `/proc/self/pagemap` since version 4.0 [40]. This change was (back)ported to the Android Linux kernel in November 2015 [34], making it impossible for us to use this interface for double-side Rowhammer.

Huge pages (P3). Although some vendors ship their mobile devices with huge page support (Motorola Moto G, 2013 model, for example), stock Android has this feature disabled by default. We thus cannot rely on huge pages to perform double-sided Rowhammer in our setting.

Additional challenges. In addition, there are further characteristics that are specific to mobile devices and that affect mobile Rowhammer attacks. First, the ARM specifications [7, 8] do not provide memory details and, for example, it is not clear what the size of a row is. Second, mobile devices do not have any swap space. Consequently, the OS—the Low Memory Killer in particular on Android—starts killing processes if the memory pressure is too high.

5. THE DRAMMER ATTACK

We now describe how we overcome the limited availability of known techniques on mobile devices and how we mount our DRAMMER attack in a deterministic fashion. In contrast to most primitives discussed in the previous section, DRAMMER relies on general memory management behavior of the OS to perform deterministic Rowhammer attacks. For simplicity, we first describe our attack for Android/ARM (focusing on the more widespread ARMv7 platform) and later discuss its applicability to other platforms in Section 7.

5.1 Mobile Device Memory

One prerequisite to implement useful exploitation primitives is to understand the memory model of the chip we are targeting. One of the key properties to determine is the *row size*. Previous x86-based efforts ascertain the row size either by consulting the appropriate documentation or by running the `decode-dimms` program. Unfortunately, ARM does not document row sizes, nor does its platform provide instructions for fingerprinting DRAM modules. As such, we propose a *timing-based side channel* to determine a DRAM chip’s row size.

Our technique is generic and can be applied independently from the chosen target architecture. It relies on the observation that accessing two memory pages from the same bank is slower than reading from different banks: for same-bank accesses, the controller has to refill the bank’s row buffer for each read operation. In particular, by accessing physical pages n and $n + i$ while increasing i from 0 to x , our timing side channel shows a slower access time when page $n + i$ lands in the same bank as page n . Such increase in access time indicates that we walked over all the pages in a row and that $n + i$ now points to the first page in the second row, falling in the same bank. By setting x large enough (e.g., to 64 pages, which would indicate a row size of 256 KB), we ensure that we always observe the side channel, as it is not expected that the row size is 256 KB or larger. We evaluate our side channel in more detail in Section 8.

5.2 DMA Buffer Management

Modern (mobile) computing platforms consist of several different hardware components: besides the CPU or System-on-Chip (SoC) itself, devices include a GPU, display controller, camera, encoders, and sensors. To support efficient memory sharing between these devices as well as between devices and userland services, an OS needs to provide direct memory access (DMA) memory management mechanisms. Since processing pipelines that involve DMA buffers bypass the CPU and its caches, the OS must facilitate explicit cache management to ensure that subparts of the pipeline have a coherent view of the underlying memory. Moreover, since most devices perform DMA operations to physically contiguous memory pages only, the OS must also provide allocators that support this type of memory.

We refer to the OS interface that provides all these mechanisms as a *DMA Buffer Management API* which essentially exports “DMA-able” memory to userland. By construction, userland-accessible DMA buffers implement two of our attack primitives: (P1) providing uncached memory access and (P3) (relative) physical memory addressing.

5.3 Physical Memory Massaging

For the remaining and most crucial primitive (P2), we need to arrange the physical memory in such a way that we can control the content of a vulnerable physical memory page and *deterministically* land security-sensitive data therein. For this purpose, we propose *Phys Feng Shui*, a novel technique to operate physical memory massaging that is solely based on the predictable *memory reuse patterns* of standard physical memory allocators. In addition to being deterministic, this strategy does not incur the risk of accidentally crashing the system by causing bit flips in unintended parts of physical memory.

On a high level, our technique works by exhausting available memory chunks of different sizes to drive the physical memory allocator into a state in which it has to start serving memory from regions that we can reliably predict. We then force the allocator to place the target security-sensitive data, i.e., a page table, at a position in physical memory which is vulnerable to bit flips and which we can hammer from adjacent parts of memory under our control.

5.3.1 Memory Templating

Since our attack requires knowledge about which exact memory locations are susceptible to Rowhammer, we first need to probe physical memory for flippable bits—although the number and location of vulnerable memory regions naturally differs per DRAM chip, once found, the large majority of flips is reproducible [23]. We refer to this process as *memory templating* [32]. A successful templating session results in a list of *templates* that contain the location of vulnerable bits, as well as the direction of the flip, i.e., whether it is a 0-to-1 or 1-to-0 flip.

5.3.2 Physical Memory Allocator

Linux platforms manage physical memory via the buddy allocator [15]. Its goal is to minimize external fragmentation by efficiently splitting and merging available memory in power-of-2 sized blocks. On each allocation request, it iteratively splits larger blocks in half as necessary until it finds a block matching the requested size. It is important to note that the buddy allocator always prioritizes the smallest fitting block when splitting—e.g., it will not attempt to split a block of 16 KB if a fitting one of 8 KB is also available. As a result, the largest contiguous chunks remain unused for as long as possible.

On each deallocation request, the buddy allocator examines neighboring blocks of the same size to merge them again if they are free. To minimize the internal fragmentation produced by the buddy allocator for small objects, Linux implements a slab allocator abstraction on top of it. The default *SLUB* allocator implementation organizes small objects in a number of pools (or *slabs*) of commonly used sizes to quickly serve allocation and deallocation requests. Each slab is expanded on demand as necessary using physically contiguous chunks of a predetermined per-slab size allocated through the buddy allocator.

5.3.3 Phys Feng Shui

Phys Feng Shui lures the buddy allocator into reusing and partitioning memory in a predictable way. For this purpose, we use three different types of physically contiguous chunks: large chunks (L), medium-sized chunks (M), and small chunks (S). The size of small chunks is fixed at 4 KB (the page size). Although other values are possible (see also Section 6), for simplicity, we set the size of M to the row size and use the size of the largest possible contiguous chunk of memory that the allocator provides for L. As illustrated in Figure 2, our attack then includes the following steps:

Preparation and templating. We first exhaust (i.e., allocate until no longer possible) all available physically contiguous chunks of size L (step 1) and probe them for vulnerable templates which we later can exploit. We then exhaust all chunks of size M (step 2), leaving the allocator in a state where blocks of size M and larger are no longer available (until existing ones are released).

Selective memory reuse. Next, we select one of the templates generated in the previous step as the target for our exploit and refer to its corresponding L block as L^* . We then release L^* (step 3), and immediately exhaust all M chunks again (step 4). Since we depleted all the free chunks of size M or larger in the previous step, this forces the allocator to place them in the region we just released (i.e., predictably reuse the physical memory region of the released L^* chunk). We refer to the M chunk that now holds the exploitable template as M^* .

Finally, in preparation of landing the page table (PT) in the vulnerable page of M^* , in the next step we release M^* (step 5). Note that we restrict our choice of M^* to chunks that are not at the edge of L^* , since we need access to its surrounding memory in order to perform double-sided Rowhammer later.

Our technique naturally increases memory pressure. In practice, the OS handles low memory or out of memory (OOM) conditions by freeing up unused memory when the available memory falls under a certain threshold. This is especially critical on mobile devices, which do not utilize swap space. In preparation of the next steps, which need to allocate several S chunks and would risk bumping the amount of available memory below the threshold, we now free the remaining L chunks to avoid triggering memory cleanup by the OS (or worse: a system crash).

Landing the first page table in the vulnerable region. We now steer the memory allocator to place a S chunk in the vulnerable chunk M^* that was released. For this purpose, we deplete the allocator of available blocks of size $S \dots M/2$ by repeatedly allocating S chunks. This guarantees that subsequent S allocations land in M^* (step 6). We allocate S chunks by forcing (4 KB) page table allocations: we repeatedly map memory at fixed virtual addresses that mark page table boundaries (i.e., every 2 MB of the virtual address space on ARMv7). Since the maximum number of page tables per process is 1024, we spawn a number of worker processes to allocate as many as we need. Once all smaller chunk sizes are depleted, our next S allocation predictably lands in the vulnerable region (no other smaller block is available).

Determining when allocations reach the vulnerable region is trivial on Linux: the proc filesystem provides `/proc/zoneinfo` and `/proc/pagetypeinfo`. These special files are world-readable and detail information on the number of available

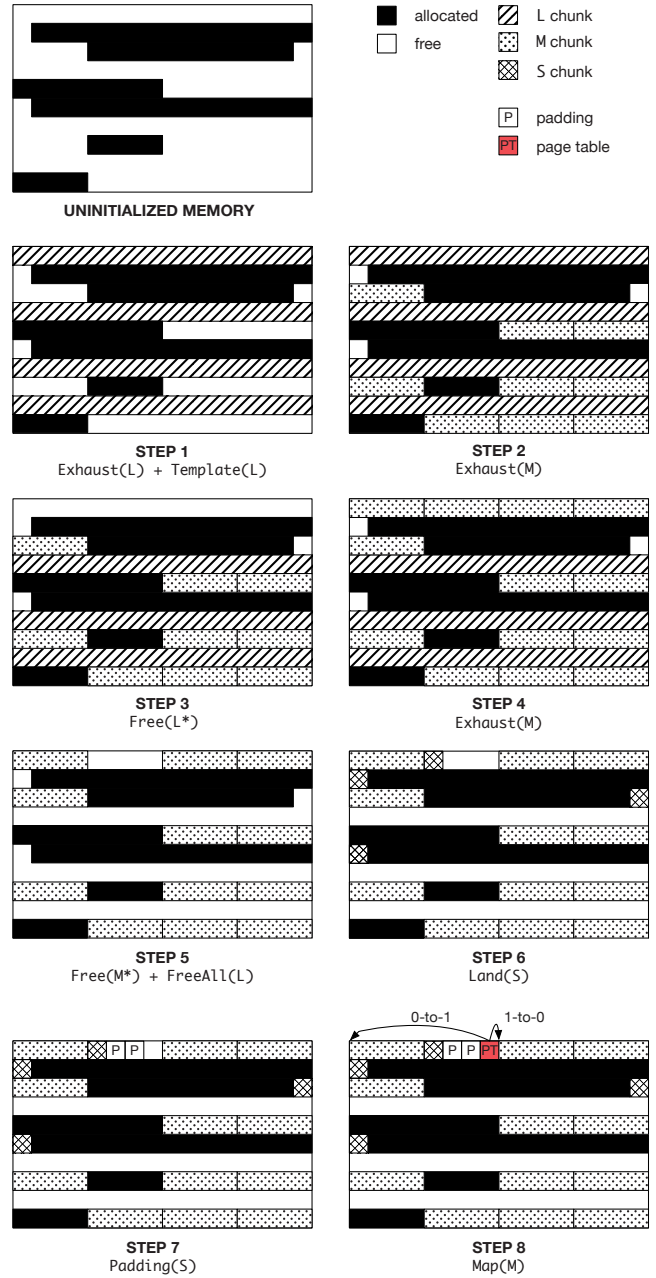


Figure 2: Physical memory layout before and after each step of *Phys Feng Shui*. Depending on the direction of the targeted bit flip, we map either the chunk before or after the vulnerable one in the last step.

and allocated memory pages and zones. In case these files are not available, we can exploit a timing or instruction-count (via the Performance Monitoring Unit) side-channel to detect when S lands in M^* : depending on whether the allocator can serve a request from a pool of available chunks of a matching size, or whether it has to start splitting larger blocks, the operation takes more time and instructions. We can use this observation by measuring the time between an allocation and deallocation of an M chunk every time we force the allocation of a new S chunk. Once this time falls below

a certain (adaptively computed) threshold, we know that we are filling the vulnerable region with S , since the allocator could no longer place a new M chunk there and had to start breaking down blocks previously occupied by one of the former L chunks.

Aligning the victim page table. Finally, we map a page p in the former L^* chunk that neighbors M^* on the left (in case of a 0-to-1 flip), or on the right (in case of a 1-to-0 flip), at a fixed location in the virtual memory address space to force a new PTP allocation (step 8). Depending on the virtual address we pick, the page table entry (PTE) that points to p is located at a different offset within the PTP—essentially allowing us to align the victim PTE according to the vulnerable template.

We can similarly align the victim PTP according to the vulnerable page to make sure that we can flip selected bits in the victim PTE. For this purpose, we force the allocation of a number of padding PTPs as needed before or after allocating the victim PTP (step 7).

We further need to ensure that the vulnerable PTP allocated in M^* and the location of p are 2^n pages apart: flipping the n lowest bit of the physical page address in the victim PTE deterministically changes the PTE to point to the vulnerable PTP itself, mapping the latter into our address space. To achieve this, we select any page p in the M chunk adjacent to M^* to map in the victim PTP, based on whether it satisfies this property.

Exploitation. Once we selected and aligned the victim PTP, PTE, and n according to the vulnerable template, we perform double-sided Rowhammer and replicate the bit flip found in the templating phase. Once we trigger the desired flip, we gain write access to the page table as it is now mapped into our address space. We can then modify one of our own PTPs and gain access to any page in physical memory, including kernel memory.

Note that the exploit is fully reliable and may only fail if the flip discovered in the templating phase is not reproducible (e.g., if a 0-to-1 flip is now applied to a 1-bit content). Since the buddy allocator provides chunk alignment by design, however, we can address this issue. By exploiting knowledge about relative offsets inside the L^* chunk, we can predict the lower bits of physical addresses in the vulnerable PTE. For example, if L^* is of size 4 MB, meaning that it must start at a physical address that is a multiple of 2^{22} , we can predict the lower $2^{22}/4096 = 2^{10} = 10$ bits of all 1024 page frames that fall in L^* . Thus, if our templating phase on ARMv7 reports a 0-to-1 flip in page 426, at bit offset 13 of a 32-bit word—a potential PTE, where offsets 1–12 are part of its properties field—we can immediately conclude that this flip is not exploitable: if, after Rowhammer, bit 13 is 1, the PTE may never point to its own page 426 (in fact, it could only point to uneven pages). In contrast, a 1-to-0 flip in page 389, at bit offset 16 of a 32-bit word is exploitable if we ensure that a PTE at this location points to page 397: `-----_01 1000 1101|ppropt iess` flips to `-----_01 1000 0101|ppropt iess` ($= 389$).

5.4 Exploitable Templates

The number of templates that an attacker can use for exploitation is determined by a combination of (i) the number of flips found in potential PTEs, and (ii) the relative location of each flip in L^* .

As discussed earlier, a bit flip in a PTE is exploitable if it flips one of the lower bits of the address part. For ARMv7, this means that flips found in the lowest 12 bits of each 32-bit aligned word are not exploitable as these fall into the properties field of a PTE. Moreover, a flip in one of the higher bits of a PTE is also not exploitable in a deterministic matter: a 0-to-1 flip in the highest bit would require the PTE to point to a page that is, physically, 2 GB to the right of its PTP. Without access to absolute physical addresses, we can only support bit flips that trigger a page offset shift of at most the size of $L - 1$. For example, if L is 4 MB (512 page frames), a 0-to-1 flip in bit 9 of a possible 32-bit word in the first page of L , is exploitable: the exploit requires a PTE that points to a page that is $2^9 = 256$ pages away from the vulnerable page. The same flip in page 300 of L , however, is not exploitable, as it would require an entry pointing to a page outside of L .

In addition, ARMv7’s page tables are, unlike x86’s ones, of size 1 KB. Linux, however, expects page tables to fill an entire page of 4 KB and solves this by storing two 1 KB ARM hardware page tables, followed by two shadow pages (used to hold extra properties that are not available in the hardware page tables), in a single 4 KB page. This design further reduces the number of exploitable flips by a factor two: only flips that fall in the first half of a page may enclose a hardware PTE.

To conclude, for ARMv7, with a maximum L size of 4 MB, a template is not exploitable if (i) it falls in the second half of a page (a shadow page) (ii) it falls in the lowest 12 bits of a 32-bit word (the properties field of a PTE), or (iii) it falls in the highest 11 bits of a 32-bit word. Consequently, for each word, at most 9 bits are exploitable, and since there are only 256 hardware PTEs per page, this means that, *at most*, 2,304 bits out of all possible 32,768 bits of a single page are exploitable (around 7.0%).

5.5 Root Privilege Escalation

Once we have control over one of our own PTPs, we can perform the final part of the attack (i.e., escalate privileges to obtain *root* access). For this purpose, we repeatedly map different physical pages to scan kernel memory for the security context of our own process (`struct cred`), which we identify by using a unique 24-byte signature based on our unique (per-app) UID. We discuss more details and evaluate the performance of our Android root exploit in Section 8.

6. IMPLEMENTATION

To demonstrate that deterministic Rowhammer attacks are feasible on commodity mobile platforms, we implemented our end-to-end DRAMMER attack on Android. Android provides DMA Buffer Management APIs through its main memory manager called ION, which allows userland apps to access uncached, physically contiguous memory. Note, however, that DRAMMER extends beyond ION and we discuss how to generalize our attack on other platforms in Section 7.

6.1 Android Memory Management

With the release of Android 4.0 (Ice Cream Sandwich), Google introduced ION [50] to unify and replace the fragmented memory management interfaces previously provided by each hardware manufacturer. ION organizes its memory pools in at least four different in-kernel heaps, including the `SYSTEM_CONTIG` heap, which allocates physically contiguous

memory allocated via `kmalloc()` (slab allocator). Furthermore, ION supports buffer allocations with explicit cache management, i.e., cache synchronization is left up to the client and memory access is essentially direct, uncached. Userland apps can interact with ION through `/dev/ion`—allowing uncached, physically contiguous memory to be allocated by any unprivileged app without any permissions.

Our implementation uses ION to allocate `L` and `M` chunks—and maps such chunks to allocate `S` page table pages (4 KB on Android/ARM). Given that SLUB’s `kmalloc()` resorts directly to the buddy allocator for chunks larger than 8 KB, we can use ION to reliably allocate 16 KB and larger chunks. We set `L` to 4 MB, the largest size `kmalloc()` supports. This gives us the most flexibility when templating and isolating vulnerable pages. Although more complex configurations are possible, for simplicity we set `M` to the row size (always larger than 16 KB). Intuitively, this allows us to release a single vulnerable row for page table allocations, while still controlling the aggressor rows to perform double-sided Rowhammer. Supporting other `M` values yields more exploitable templates, at the cost of additional complexity.

6.2 Noise Elimination

To ensure reliability, an attacker needs to eliminate interferences from other activity in the system (e.g., other running apps) during the *Phys Feng Shui* phase. The risk of interferences is, however, minimal. First, our *Phys Feng Shui* phase is designed to be extremely short-lived and naturally rule out interferences. Second, interferences are only possible when the kernel independently allocates memory via the buddy allocator in the low memory zone. Since most kernel allocations are served directly from slabs, interferences are hard to find in practice.

Nonetheless, the attacker can further minimize the risk of noise by scheduling the attack during low system activity, e.g., when no user is interacting with the device or when the system enters low power mode with essentially no background activity. Android provides notifications for both scenarios through the intents `ACTION_SCREEN_OFF` and `ACTION_BATTERY_LOW`.

7. GENERALIZATION

ION facilitates a Rowhammer attack on Android/ARM by readily providing DMA buffer management APIs to userland, but it is not yet available on every Linux platform (although there are plans to upstream it [39, 42]). Nonetheless, we describe how one can generalize our deterministic attack to other (e.g., x86) platforms by replacing ION with other standard capabilities found in server and desktop Linux environments. More specifically, the use of ION in DRAMMER can be replaced with the following strategies:

(1) Uncached memory. Rather than having ION map uncached memory in userland, one can employ `clflush` or any of the other cache eviction techniques that have previously been used for Rowhammer [11, 16, 23, 25, 26, 31, 32, 35, 47].

(2) L chunks. Transparent hugepages (THP) [2] supported by Linux (enabled by default on recent distributions for better performance [6, 33] and available on some Android devices) can replace the physically contiguous `L` chunks allocated by ION. By selectively unmapping part of each `L` chunk backed by a THP, one can also directly create a `M`-sized hole without filling it with `M` chunks first. To learn

when a THP allocation fails (i.e., when `L` chunks have been exhausted), one can read statistics from `/proc/vmstat` [24]. To force the kernel to allocate THPs (normally allocated in high memory) in low memory (normally reserved to kernel pages, e.g., PTPs), one can deplete the `ZONE_HIGHMEM` zone before starting the attack, as detailed in prior work [22]. Note that THPs contribute to the ability to mount deterministic attacks, not just to operate double-sided Rowhammer [32, 35, 47], in our setting.

(3) M chunks. While using THPs to exhaust `M` chunk allocations is infeasible (they are larger), one can abuse the SLUB allocator and force it to allocate chunks of a specific size through the buddy allocator. This can be done by depleting a slab cache of a carefully selected size (e.g., depleting `kmalloc-256` would force one 4 KB allocation) by triggering multiple allocations via particular system calls, e.g., `sendmmsg()`, as described by Xu et al. [48, 49].

(4) Predictable PTE content. To ensure that our victim PTE points to an attacker-controlled physical page at a predictable offset, we rely on mapping neighboring chunks (the first and the last fragment of a THP in our generalized attack) multiple times in userland (the first time at allocation time, the second time when creating the victim page table). To implement this strategy with THPs, one can create a shared memory segment associated to the vulnerable THP (shared memory support for THPs is being merged mainline [41]) and attach it multiple times at fixed addresses to force page table allocations with PTEs pointing to it. As an alternative (until THP shared memory support is available mainline), one can simply map a new anonymous (4 KB) user page when forcing the allocation of the page table. As the `ZONE_HIGHMEM` zone is depleted, such user page will also end up in low memory right next to the PTP.

8. EVALUATION

In this section, we evaluate various aspects of DRAMMER. We (i) evaluate our proposed side channel to detect the row size on a given device. Using its results, we (ii) perform an empirical study on a large set of Android smartphones to investigate to what extent consumer devices are affected by susceptible DRAM chips. Finally, we (iii) combine results from our study with the final exploitation step (i.e., from page table write access to root privilege escalation) and compute how fast we can perform our end-to-end attack.

8.1 Mobile Row Sizes

We evaluate the row size detection side channel described in Section 5 by constructing a heatmap for page-pair access times. We set our upper limit in both directions to 64 pages, and thus read from page pairs $(1, 1), (1, 2), \dots, (1, 64), (2, 1), (2, 2), \dots, (64, 64)$.

Figure 3 shows such a heatmap for a LG Nexus 5 phone. We determined that the row size is 16 pages, which is $16 \times 4K = 64K$. This is somewhat surprising, given that most previous x86 efforts report a row size of 128K.

8.2 Empirical Study

For our empirical study, we acquired the following ARMv7-based devices: 15 LG Nexus 5 phones, a Samsung Galaxy S5, two One Plus Ones, two Motorola Moto G devices (the 2013 and 2014 model), and a LG Nexus 4. We further analyzed a number of ARMv8-based devices: a LG Nexus 5X,

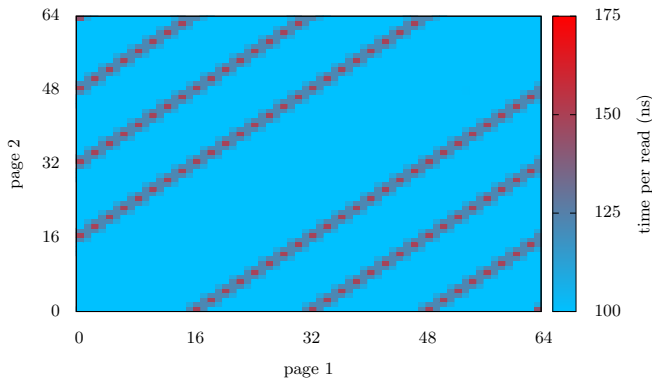


Figure 3: Heatmap representing the time required to access a given pair of pages on a LG Nexus 5. The diagonal pattern clearly indicates that the row size is 16 pages = 64K.

a Samsung Galaxy S6, a Lenovo K3 Note, a Xiaomi Mi 4i, a HTC Desire 510, and a LG G4.

We subject each device to a Rowhammer test, which (i) exhausts all the large ION chunks to allocate a maximum amount of hammerable memory (starting at 4MB chunks, down to chunks that are 4 times the row size); (ii) performs double-sided Rowhammer on each victim *page* for which aggressor pages are available twice and checks the entire victim *row* for flips (i.e., we hammer once with all victim bits set to 1—searching for 1-to-0 flips—and once with all bits set to 0); and (iii) for each induced flip, dumps its virtual (and physical, if `/proc/self/pagemap` is available) address, the time it took to find it (i.e., after n seconds), its direction (1-to-0 or 0-to-1), and whether it is exploitable by our specific DRAMMER attack, as discussed in Section 5.

To hammer a page, we perform 2x 1M read operations on its aggressor pages. Although prior work shows that 2.5M reads yields the optimal amount of flips, lowering the *read count* allows us to finish our experiments faster, while still inducing plenty of bit flips.

Our results (presented in Table 2) show that many tested ARMv7 devices are susceptible to the Rowhammer bug, while our ARMv8 devices seem somewhat more reliable (although still vulnerable). However, due to our small current ARMv8 sample size, we cannot conclude that ARMv8 devices are more resilient by design. Moreover, it should be noted that our Nexus 5 devices have been used extensively in the past for a variety of research projects. This may indicate a correlation between (heavy) use and the ability to induce (more) bit flips. Future research is required, however, to confirm whether *DRAM wearing* actually contributes to more observable bit flips.

A second key finding presented in Table 2 is that a device from 2013 (Moto G, 1st generation) is vulnerable to the Rowhammer bug. This shows that ARM memory controllers have been capable in performing Rowhammer attacks for at least three years. Moreover, the MSM8226 SoC (Qualcomm Snapdragon 400), which both Moto Gs are shipped with, is still a popular SoC for smartwatches (e.g., Motorola Moto 360 2nd generation or Huawei Watch), suggesting that smartwatches may be susceptible to Rowhammer-based ex-

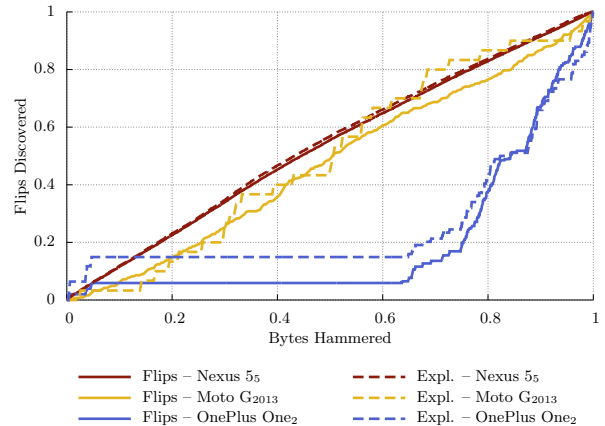


Figure 4: Template distribution over memory for three cases—one with many flips (Nexus 5₅), one with few flips (Moto G₂₀₁₃), one with a large memory region without flips (OnePlus One₂).

ploits as well. Additionally, while the majority of flips are induced on LPDDR3-devices, the reported flips on the Nexus 4 show that even LPDDR2 is vulnerable.

Third, when looking at the exploitability of observed flips (*exploitable* and *1st* columns of Table 2), we conclude that *once we see flips, we will eventually find exploitable flips*. Although only around 6% of all observed flips are exploitable, we always find enough flips to also find those that we can use in our end-to-end Android root exploit.

To get a better understanding of the template distribution over memory, we display a CDF for three interesting devices in Figure 4: one with many flips (Nexus 5₅), one with few flips (Moto G₂₀₁₃), and the OnePlus One₂. For most other devices, the distribution trend is similar as that of the Nexus 5 and Moto G. For the OnePlus One, however, it is interesting to notice that a large region of the DRAM exposes no flips at all (from 5% to 60%).

8.3 Root Privilege Escalation

Armed with bit flips from the memory templating step, we rely on *Phys Feng Shui* to place the victim page table in a vulnerable template-matching location and proceed to reproduce an exploitable bit flip. This step allows us to control one of our own page tables, enabling root privilege escalation. To complete our Android root exploit, we overwrite the controlled page table to probe kernel memory for our own `struct cred` structure.

The `struct cred` structure represents a process’ security context and holds, among others, its real, effective, and saved user and group IDs (6 UIDs). Since Android provides each app—and thus each running process—a unique UID, we can fingerprint a security context by comparing $6 \times 4 = 24$ bytes. In our experiments on the latest kernel, the physical page that stores a specific `struct cred` has 20 bits of entropy, placed between `0x30000000` and `0x38000000`. Moreover, the structure is always aligned to a 128 byte boundary, that is there are $\frac{4096}{128} = 32$ possible locations within a page on which a `struct cred` can be found. To successfully find our own, we thus have to map and scan 2^{20} different physi-

Table 2: Empirical analysis results. For each device, the table shows *Hardware Details*, which include the SoC (System on a Chip), available DRAM (LPDDR3, unless stated otherwise), and detected row size (RS), and *Analysis Results*. For the latter, the *MB* column depicts the amount of hammered DRAM, *ns* shows the median access time for a single read operation, *flips* holds the number of unique flips found, *KB* depicts the average amount of KB that contain a single flip, *1-to-0* and *0-to-1* show the directions of the found flips, *exploitable* shows the number of exploitable templates according our attack, and *1st* shows after how many seconds of hammering we found the first exploitable flip. For same model devices, we use a subscript number to identify them individually. The top half of the table shows ARMv7-based (32-bit) smartphones, while the lower rows are ARMv8 (64-bit).

Device	Hardware Details			Analysis Results								
	SoC	DRAM	RS	MB	ns	# flips	KB	# 1-to-0	# 0-to-1	# exploitable	1 st	
ARMv7	Nexus 5 ₁	MSM8974 [†]	2 GB	64	441	70	1,058	426	1,011	47	62 (5.86%)	116s
	Nexus 5 ₂	MSM8974 [†]	2 GB	64	472	69	284,428	2	261,232	23,196	14,852 (5.22%)	1s
	Nexus 5 ₃	MSM8974 [†]	2 GB	64	461	69	547,949	1	534,695	13,254	32,715 (5.97%)	1s
	Nexus 5 ₄	MSM8974 [†]	2 GB	64	616	71	0	–	–	–	–	–
	Nexus 5 ₅	MSM8974 [†]	2 GB	64	630	69	747,013	1	704,824	42,189	46,609 (6.24%)	1s
	Nexus 5 ₆	MSM8974 [†]	2 GB	64	512	69	215,233	3	207,856	7,377	13,365 (6.21%)	3s
	Nexus 5 ₈	MSM8974 [†]	2 GB	64	485	70	32,328	15	28,500	3,828	1,894 (5.86%)	4s
	Nexus 5 ₉	MSM8974 [†]	2 GB	64	569	69	476,170	2	434,086	42,084	30,190 (6.34%)	0s
	Nexus 5 ₁₀	MSM8974 [†]	2 GB	64	406	69	160,245	3	150,485	9,760	8,701 (5.43%)	1s
	Nexus 5 ₁₁	MSM8974 [†]	2 GB	64	613	70	0	–	–	–	–	–
	Nexus 5 ₁₂	MSM8974 [†]	2 GB	64	600	70	17,384	35	16,767	617	1,241 (7.14%)	16s
	Nexus 5 ₁₃	MSM8974 [†]	2 GB	64	575	69	161,514	4	160,473	1,041	10,378 (6.43%)	355s
	Nexus 5 ₁₄	MSM8974 [†]	2 GB	64	576	69	295,537	2	277,708	17,829	18,900 (6.40%)	1s
	Nexus 5 ₁₅	MSM8974 [†]	2 GB	64	573	69	38,969	15	35,515	3,454	2,775 (7.12%)	11s
	Nexus 5 ₁₇	MSM8974 [†]	2 GB	64	621	70	0	–	–	–	–	–
	Galaxy S5	MSM8974 [‡]	2 GB	64	207	82	0	–	–	–	–	–
	OnePlus One ₁	MSM8974 [‡]	3 GB	64	292	71	3,981	75	2,924	1,057	242 (6.08%)	942s
OnePlus One ₂	MSM8974 [‡]	3 GB	64	1189	69	1,992	611	942	1,050	94 (4.72%)	326s	
Moto G ₂₀₁₃	MSM8226	1 GB	32	134	127	429	275	419	10	30 (6.99%)	441s	
Moto G ₂₀₁₄	MSM8226	1 GB	32	151	127	1,577	98	1,523	54	71 (4.66%)	92s	
Nexus 4	APQ8064	2 GB*	64	82	18	1,328	64	1,061	267	104 (7.83%)	7s	
ARMv8	Nexus 5x	MSM8992	2 GB	64	271	63	0	–	–	–	–	–
	Galaxy S6	Exynos7420	3 GB ^o	128	234	82	0	–	–	–	–	–
	K3 Note	MT6752	2 GB	64	423	218	0	–	–	–	–	–
	Mi 4i	MSM8939	2 GB	64	327	159	0	–	–	–	–	–
	Desire 510	MSM8916	1 GB	32	186	122	0	–	–	–	–	–
	G4	MSM8992	3 GB	64	833	64	117,496	8	117,260	236	6,560 (5.58%)	5s

[†]MSM8974AA [‡]MSM8974AC *LPDDR2 ^oLPDDR4

cal pages in the worst-case scenario, and for each page perform 32 different compare operations, resulting in, at most, $2^{20} * 32 = 33,554,432$ calls to `memcmp`. Since we control only a single page table—on ARMv7 capable of storing PTEs to 512 physical pages—we also need to flush the TLB every 512 tries. Thus, in order to read all possible pages that may contain our `struct cred`, we must perform $\frac{2^{20}}{512} = 2,048$ TLB flushes.

We flush the TLB by reading from 8,196 different pages in a 32 MB memory region, which takes approximately 900 μ s. Comparing 24 bytes using `memcmp()` takes at most 600 ns, limiting the upper bound time of the final exploitation step to $2,048 * 900 \mu$ s + $33,554,432 * 600$ ns \sim 22 seconds (measured on a Nexus 5). Note that having to break 20 bits of entropy does not make our attack less deterministic: we will always be able to find our own `struct cred`.

Based on the results from our empirical study, we find that, for the most vulnerable phone, an end-to-end attack (including the final exploitation step) takes less than 30 sec-

onds, while in the worst-case scenario, it takes a little over 15 minutes, where templating is obviously the most time-consuming phase of the attack. To confirm that our exploit is working, we successfully exploited our Nexus 5s in less than 20 seconds.²

Finally, to support future research on mobile Rowhammer and to expand our empirical study to a broader range of devices, we release our codebase as an open source project and aim to build a public database of known vulnerable devices on our project website.

9. MITIGATION AND DISCUSSION

In this section, we investigate the effectiveness of current Rowhammer defenses and discuss potential design improvements of the memory management process that could mitigate our attack.

²See <https://vusec.net/projects/drammer/> for a demo.

9.1 Existing Rowhammer Defenses

Countermeasures against Rowhammer have already been proposed, both in software and hardware, but very few are applicable to the mobile domain or effective against a generic attack such as the one we proposed.

Software-based. *Instruction “blacklisting”*, i.e., disallowing or rewriting instructions such as `CLFLUSH` [35, 36] and non-temporal instructions [31] has been proposed as a countermeasure and is now deployed in Google Native Client (NaCl). Similarly, access to the Linux `pagemap` interface is now prohibited from userland [34, 40]. However, these countermeasures have already proven insufficient, since Rowhammer has been demonstrated in JavaScript [11, 16], where neither these special instructions nor the `pagemap` interface are present. As a more generic countermeasure, ANVIL [9] tries to detect Rowhammer attacks by monitoring the last-level cache miss rate and row accesses with high temporal locality. Similarly, Herath et al. [17] propose to monitor the number of last-level cache misses during a given refresh interval. Both approaches rely on CPU performance counters specific to Intel/AMD. Furthermore, our attack bypasses the cache completely, thus producing no cache misses that could raise red flags.

Hardware-based. Memory with *Error Correcting Codes (ECC)* corrects single bit flip errors, and reports other errors. However, Lanteigne [26] studied Rowhammer on server settings with ECC and reported surprising results, as some server vendors implement ECC to report bit flip errors only upon reaching a certain threshold—and one vendor even failed to report any error. Likewise, ECC often does not detect multiple flips in a single row. *Doubling DRAM refresh rates* has been the response of most hardware vendors via EFI or BIOS updates [5, 18, 27]. It severely limits most attacks. However, Kim et al. [23] show that the refresh rate would need to be increased up to eight times to completely mitigate the issue. Aweke et al. [9] mention that both doubling the DRAM refresh rate and prohibiting the `CLFLUSH` instruction defeat Rowhammer attacks, but no system currently implements it. As increased refresh rates have severe consequences for both power consumption and performance [10], this countermeasure does not seem well-suited for mobile devices. In addition, it aligns poorly with the direction taken by the LPDDR4 standard [20], which requires the refresh rate to drop at low temperatures to conserve battery life.

Further mitigations rely on the *Detection of Activation Patterns* to refresh targeted rows and need support from the DRAM chip or the memory controller. The LPDDR4 standard proposes Target Row Refresh (TRR) [20], which seems to be an effective countermeasure, but we need to expand our study to more devices shipped with this type of memory. Probabilistic Adjacent Row Activation (PARA) [23] refreshes neighboring rows on each activation with a low probability (and thus very likely during repeated activations in a Rowhammer attack), but requires modifications of the memory controller to do so. ARMOR [14] introduces an extra cache buffer for rows with frequent activations, i.e., hammered rows, but again needs to be implemented in the memory controller.

9.2 Countermeasures Against Drammer

We now elaborate on countermeasures that are more specific to our DRAMMER attack on mobile platforms.

Restriction of userland interface. Since DMA plays an important part in the deterministic Rowhammer attack on mobile devices, the question arises whether userland apps should be allowed unrestricted access to DMA-able memory. On Android, the motivation for doing so via ION is device fragmentation: vendors need to define custom heaps depending on the specific hardware requirements of each product, and provide a mapping of use cases to heaps in their custom implementation of `gralloc()`. It is Google’s policy to keep the vendors’ product-specific code in user rather than in kernel mode [43].³ Linux implements similar DMA-support with the `dma-buf` buffer sharing API [37], but with a more restricted interface. However, ION seems to fill a gap in this regard [3] and efforts are underway to upstream it [39, 42].

Concurrently to our work Google has adopted several defenses from the Linux kernel in Android [21] concerning memory protection and attack surface reduction. While the majority of defenses do not affect our attack, Android now provides mechanisms to enforce access restrictions to `ioctl` commands and added `seccomp` to enable system call filters. These mechanisms could be used to restrict the userland interface of ION.

However, we note that simply disabling ION is not the final solution to Rowhammer-based attacks: (i) as discussed in Section 7, it is possible to generalize our attack to other Linux-based platforms without ION; (ii) since a large number of DRAM chips are vulnerable to bit flips and an attacker might still be able to exploit them through other means. Nevertheless, improvements to the interface of ION and memory management in general could significantly raise the bar for an attacker. For example, a possible improvement is to adopt constraint-based allocation [38], where the (ION) allocator picks the type of memory for an allocation based on constraints on the devices sharing a buffer and defers the actual allocation of buffers until the first device attaches one (rather than upon request from a userland client).

Memory isolation and integrity. In the face of an OS interface that provides user applications access to DMA-able memory, stricter enforcement of memory isolation may be useful. Specifically, it may be possible to completely isolate DMA-able memory from other regions. Currently, ION readily breaks the isolation of memory zones by allowing userland to allocate physically contiguous memory in low memory regions usually reserved for the kernel and page tables. One option is to isolate ION regions controlled by userland from kernel memory. In principle, ION can already support isolated heaps (e.g., ION carveout), but such heaps are statically preallocated and do not yet provide a general buffer management primitive. Furthermore, even in the absence of ION, an attacker can force the buddy allocator to allocate memory (e.g., huge or regular pages) in kernel memory zones by depleting all the memory available to userland [22]. Thus, the design of isolation and integrity measures for security-critical data such as page tables also needs improvements.

³Full discussion at the Linux Plumbers Conference 2013: <https://www.youtube.com/watch?v=8okc75j5cKk>

For instance, the characteristics of the underlying DRAM cells could be taken into account when allocating memory regions for security-critical data. Flikker [29] proposes to allocate critical data in memory regions with higher refresh rates than non-critical data. RAPID [45] suggests that the OS should prefer pages with longer retention times, i.e., that are less vulnerable to bit flips. Even without a DRAM-aware allocator, isolating security-critical data (e.g., page tables) in zones that the system never uses for data that can be directly (e.g., ION buffers) or indirectly (e.g., slab buffers) controlled would force attackers to resort to a probabilistic attack with low chances of success (no deterministic or probabilistic memory reuse). However, enforcing strict isolation policies is challenging as, when faced with high memory pressure, the physical page allocator naturally encourages cross-zone reuse to eliminate unnecessary OOM events—opening up again opportunities for attacks [22]. In addition, even strict isolation policies may prove insufficient to completely shield security-sensitive data. For example, ION is also used by the media server, which is running at a higher privilege than normal apps. Hence, an attacker controlling a hypothetically isolated ION region could still potentially corrupt security-sensitive data, i.e., the media server’s state, rather than, say, page tables.

Prevention of memory exhaustion. Per-process memory limits could make it harder for an attacker (i) to find exploitable templates and (ii) exhaust all available memory chunks of different sizes during *Phys Feng Shui*. Android already enforces memory limits for each app, but only at the Dalvik heap level. As a countermeasure, we could enforce this limit at the OS level (accounting for *both* user and kernel memory), and per-user ID (to prohibit collusion).

10. RELATED WORK

The Rowhammer bug has gathered the attention of the scientific community for two years, beginning with the work of Kim et al. [23], who studied the possibility and the prevalence of bit flips on DDR3 for x86 processors. Aichinger [4] later analyzed the prevalence of the Rowhammer bug on server systems with ECC memory and Lanteigne performed an analysis on DDR4 memory [25]. In contrast to these efforts, we are the first to study the prevalence of the Rowhammer bug on ARM-based devices. Several efforts focused on finding new attack techniques [9, 11, 16, 23, 31, 32]. However, all these techniques only work on x86 architectures and, as discussed in Section 4.3, are not applicable to our setting.

DRAMMER is an instance of the *Flip Feng Shui* (*FFS*) exploitation technique [32]. Rather than using memory deduplication, DRAMMER relies on *Phys Feng Shui* for physical memory massaging on Linux. This shows that *FFS* can be implemented with always-on commodity features.

Lipp et al. [28] demonstrated cache eviction on ARM-based mobile devices, but did not evaluate the possibility of Rowhammer attacks based on cache eviction. Other attack techniques focus on the DRAM itself. Lanteigne [25, 26] examined the influence of data and access patterns on bit flip probabilities on DDR3 and DDR4 memory on Intel and AMD CPUs. Pessl et al. [30] demonstrated that reverse engineering the bank DRAM addressing can reduce the search time for Rowhammer bit flips. These techniques are complementary to our work.

Another line of related work uses the predictable memory allocation behavior of Linux for the exploitation of use-after-free vulnerabilities. Kemerlis et al. [22] showed in their *ret2dir* attack how kernel allocators can be forced to allocate user memory in kernel zones. Xu et al. [48, 49] used the recycling of slab caches by the SLUB allocator to craft the *PingPongRoot* root exploit for Android. Finally, Lee Campbell [12] relied on kernel object reuse to break out of the Chrome sandbox on Android.

In concurrent work, Zhang et al. [51] perform a systematic analysis of the design and implementation of ION, although without studying the topic of cache coherency. They found similar security flaws as the ones we exploit for our attack, but described different attack scenarios: (i) due to the unlimited access to ION heaps, both concerning access restrictions and memory quotas, an attacker can perform a denial-of-service attack by exhaustively allocating all device memory; (ii) the recycling of kernel memory for userland apps—and in this case missing buffer zeroing logic in between—makes ION vulnerable to sensitive information leakage. Consequently, their proposed redesign of ION contains some of the countermeasures we discussed in the previous section, e.g., enforcing memory quotas and restricting the userland interface. However, they observe that implementing these changes is challenging, as they incur performance penalties, break backward-compatibility, and add complexity by introducing new security mechanisms specifically for the access to ION heaps.

11. CONCLUSION

In this paper, we demonstrated that powerful *deterministic* Rowhammer attacks that grant an attacker root privileges on a given system are possible, even by only relying on always-on features provided by commodity operating systems. To concretely substantiate our claims, we presented an implementation of our DRAMMER attack on the Android/ARM platform. Not only does our attack show that practical, deterministic Rowhammer attacks are a real threat for billions of mobile users, but it is also the first effort to show that Rowhammer is even possible at all (and reliably exploitable) on any platform other than x86 and with a much more limited software feature set than existing solutions. Moreover, we demonstrated that several devices from different vendors are vulnerable to Rowhammer. To conclude, our research shows that practical large-scale Rowhammer attacks are a serious threat and while the response to the Rowhammer bug has been relatively slow from vendors, we hope our work will accelerate mitigation efforts both in industry and academia.

Disclosure

We have reported our attack (and possible countermeasures) to Google and cooperated with the National Cyber Security Centre in the Netherlands to coordinate disclosure of the vulnerabilities to relevant parties.

Acknowledgements

We would like to thank the anonymous reviewers for their valuable comments and input to improve the paper, as well as Ben Gras, Erik Bosman, Kevin Borgolte, Andrea Continella, Michael Schwarz, and Moritz Lipp for help with some

experiments. This work was supported by the Netherlands Organisation for Scientific Research through grants NWO 639.023.309 VICI “Dowsing” and NWO CSI-DHS 628.001.021, and by the European Commission through project H2020 ICT-32-2014 “SHARCS” under Grant Agreement No. 644571, programme GA No. 644052 (HECTOR) and EU FP7 programme GA No. 610436 (MATTHEW).

This material is also based upon work supported by the ONR under Award No. N00014-15-1-2948, and by the NSF under Award No. CNS-1408632. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the ONR and the NSF.

12. REFERENCES

- [1] Low RAM Configuration. <https://source.android.com/devices/tech/config/low-ram.html>.
- [2] Transparent Hugepage Support. <https://www.kernel.org/doc/Documentation/vm/transhuge.txt>.
- [3] L. Abbott. Lessons from Ion. Embedded Linux Conference (ELC), April 2016.
- [4] B. Aichinger. DDR Memory Errors caused by Row Hammer. In *Proceedings of the 19th IEEE High Performance Extreme Computing Conference (HPEC)*, 2015.
- [5] Apple Inc. Mac EFI Security Update 2015-001. <https://support.apple.com/en-us/HT204934>, June 2015.
- [6] A. Arcangeli. Transparent Hugepage Support. <http://www.linux-kvm.org/images/9/9e/2010-forum-thp.pdf>, August 2010.
- [7] ARM Limited. *ARM Architecture Reference Manual. ARMv7-A and ARMv7-R edition*, 2012.
- [8] ARM Limited. *ARM Architecture Reference Manual. ARMv8, for ARMv8-A architecture profile*, 2013.
- [9] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin. ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.
- [10] I. Bhati, M.-T. Chang, Z. Chishti, S.-L. Lu, and B. Jacob. DRAM Refresh Mechanisms, Penalties, and Trade-Offs. *IEEE Transactions on Computers*, 65(1), 2016.
- [11] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P)*, 2016.
- [12] L. Campbell. Exploiting NVMAP to escape the Chrome sandbox - CVE-2014-5332. <http://googleprojectzero.blogspot.com/2015/01/exploiting-nvmap-to-escape-chrome.html>, January 2015.
- [13] H. Flake. Three Things that Rowhammer Taught Me. Null Singapore, March 2016.
- [14] M. Ghasempour, M. Lujan, and J. Garside. ARMOR: A Run-Time Memory Hot-Row Detector. <http://apt.cs.manchester.ac.uk/projects/ARMOR/RowHammer>, 2015.
- [15] M. Gorman. *Understanding the Linux Virtual Memory Manager*. Prentice Hall PTR, 2007.
- [16] D. Gruss, C. Maurice, and S. Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *Proceedings of the 13th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2016.
- [17] N. Herath and A. Fogh. These are Not Your Grand Daddy’s CPU Performance Counters - CPU Hardware Performance Counters for Security. In *Black Hat USA (BH-US)*, 2015.
- [18] Hewlett Packard. Moonshot Component Pack Version 2015.05.0 Release Notes. <http://h10032.www1.hp.com/ctg/Manual/c04676483>, May 2015.
- [19] JEDEC Solid State Technology Association. DDR3 SDRAM Specification. JESD79-3F, 2012.
- [20] JEDEC Solid State Technology Association. Low Power Double Data 4 (LPDDR4). JESD209-4A, 2015.
- [21] Jeff Vander Stoep. Protecting Android with more Linux kernel defenses. <http://android-developers.blogspot.com/2016/07/protecting-android-with-more-linux.html>, July 2016.
- [22] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis. ret2dir: Rethinking Kernel Isolation. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.
- [23] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)*, 2014.
- [24] C. Lameter. Light weight event counters V4. <https://lwn.net/Articles/188327>, June 2006.
- [25] M. Lanteigne. A Tale of Two Hammers: A Brief Rowhammer Analysis of AMD vs. Intel. <http://www.thirdio.com/rowhammer1.pdf>, May 2016.
- [26] M. Lanteigne. How Rowhammer Could Be Used to Exploit Weaknesses in Computer Hardware. <http://www.thirdio.com/rowhammer.pdf>, March 2016.
- [27] Lenovo. Row Hammer Privilege Escalation. https://support.lenovo.com/us/en/product_security/row_hammer, March 2015.
- [28] M. Lipp, D. Gruss, R. Spreitzer, and S. Mangard. ARMageddon: Cache Attacks on Mobile Devices. In *Proceedings of the 25th USENIX Security Symposium*, 2016.
- [29] S. Liu, K. Pattabiraman, T. Moscibroda, and B. Zorn. Flicker: Saving DRAM Refresh-power through Critical Data Partitioning. In *Proceedings of the 16th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.
- [30] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *Proceedings of the 25th USENIX Security Symposium*, 2016.

- [31] R. Qiao and M. Seaborn. A New Approach for Rowhammer Attacks. In *Proceedings of the 9th IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2016.
- [32] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos. Flip Feng Shui: Hammering a Needle in the Software Stack. In *Proceedings of the 25th USENIX Security Symposium*, 2016.
- [33] Red Hat. How to use, monitor, and disable transparent hugepages in Red Hat Enterprise Linux 6? <https://access.redhat.com/solutions/46111>, September 2015.
- [34] M. Salyzyn. AOSP Commit 0549ddb9: "UPSTREAM: pagemap: do not leak physical addresses to non-privileged userspace". <http://goo.gl/Qye2MN>, November 2015.
- [35] M. Seaborn and T. Dullien. Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges. In *Black Hat USA (BH-US)*, 2015.
- [36] M. Seaborn and T. Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges. <http://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html>, March 2015.
- [37] S. Semwal. DMA Buffer Sharing API Guide. <https://www.kernel.org/doc/Documentation/dma-buf-sharing.txt>.
- [38] S. Semwal. dma-buf Constraints-Enabled Allocation helpers. <https://lwn.net/Articles/615892/>, October 2014.
- [39] S. Semwal. Upstreaming ION Features: Issues that remain. Linux Plumbers Conference, August 2015.
- [40] K. A. Shutemov. Linux commit ab676b7d: "pagemap: do not leak physical addresses to non-privileged userspace". <http://goo.gl/Zvd0qf>, March 2015.
- [41] K. A. Shutemov. THP-enabled tmpfs/shmem using compound pages. <http://lwn.net/Articles/687352>, May 2016.
- [42] J. Stultz. Integrating the ION memory allocator. <https://lwn.net/Articles/565469/>, September 2013.
- [43] J. Stultz. Summary of the Android Graphics microconference. <https://lwn.net/Articles/569704/>, October 2013.
- [44] Unity. Mobile (Android) Hardware Stats. <http://hwstats.unity3d.com/mobile/cpu-android.html>, June 2016.
- [45] R. K. Venkatesan, S. Herr, and E. Rotenberg. Retention-aware placement in DRAM (RAPID): Software methods for quasi-non-volatile DRAM. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA)*, 2006.
- [46] VMware. Security considerations and disallowing inter-Virtual Machine Transparent Page Sharing. <https://kb.vmware.com/kb/2080735>, October 2014.
- [47] Y. Xiao, X. Zhang, Y. Zhang, and M.-R. Teodorescu. One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. In *Proceedings of the 25th USENIX Security Symposium*, 2016.
- [48] W. Xu and Y. Fu. Own Your Android! Yet Another Universal Root. In *Proceedings of the 9th USENIX Workshop on Offensive Technologies (WOOT)*, 2015.
- [49] W. Xu, J. Li, J. Shu, W. Yang, T. Xie, Y. Zhang, and D. Gu. From Collision To Exploitation: Unleashing Use-After-Free Vulnerabilities in Linux Kernel. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, 2015.
- [50] T. M. Zeng. The Android ION memory allocator. <https://lwn.net/Articles/480055>, February 2012.
- [51] H. Zhang, D. She, and Z. Qian. Android ION Hazard: the Curse of Customizable Memory Management System. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, 2016.