Triggering Rowhammer Hardware Faults on ARM: A Revisit

Zhenkai Zhang
Vanderbilt University
Nashville, TN, USA
zhenkai.zhang@vanderbilt.edu

Zihao Zhan
Vanderbilt University
Nashville, TN, USA
zihao.zhan@vanderbilt.edu

Daniel Balasubramanian
Vanderbilt University
Nashville, TN, USA
daniel@isis.vanderbilt.edu

Xenofon Koutsoukos
Vanderbilt University
Nashville, TN, USA
xenofon.koutsoukos@vanderbilt.edu

Gabor Karsai
Vanderbilt University
Nashville, TN, USA
gabor.karsai@vanderbilt.edu

ABSTRACT

The rowhammer bug belongs to software-induced hardware faults, and has posed great security challenges to numerous systems. On x86, many approaches to triggering the rowhammer bug have been found; yet, due to several different reasons, the number of discovered approaches on ARM is limited. In this paper, we revisit the problem of how to trigger the rowhammer bug on ARM-based devices by carefully investigating whether it is possible to translate the original x86-oriented rowhammer approaches to ARM. We provide a thorough study of the unprivileged ARMv8-A cache maintenance instructions and give two previously overlooked reasons to support their use in rowhammer attacks. Moreover, we present a previously undiscovered instruction that can be exploited to trigger the rowhammer bug on many ARM-based devices. A potential approach to quickly evicting ARM CPU caches is also discussed, and experimental evaluations are carried out to show the effectiveness of our findings.

CCS CONCEPTS

• **Security and privacy → Embedded systems security**; **Hardware attacks and countermeasures**;

KEYWORDS

Hardware Faults, Rowhammer, Microarchitectural Attacks

ACM Reference Format:
Zhenkai Zhang, Zihao Zhan, Daniel Balasubramanian, Xenofon Koutsoukos, and Gabor Karsai. 2018. Triggering Rowhammer Hardware Faults on ARM: A Revisit. In *The Second Workshop on Attacks and Solutions in Hardware Security (ASHES'18), October 19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3266444.3266454

1 INTRODUCTION

As a basic security requirement, data in memory should be protected from unauthorized modifications; otherwise, the integrity of a system and its computations cannot be trusted. Through the years, there have been many efforts in both software and hardware towards meeting this requirement, resulting in mature techniques that are ubiquitously employed. However, a recently revealed hardware fault vulnerability in dynamic random-access memory (DRAM), the so-called rowhammer bug, introduces new attack surfaces for unauthorized data modifications [15] – when the activation of a DRAM row is toggled repeatedly at a high frequency, i.e., hammered, some bit(s) may flip in some physically adjacent row(s).

The existence of the rowhammer bug has been reported in various DRAM chips of different types [2, 15, 16, 27]. Since its discovery, this hardware bug has been continuously exploited to form a wide range of powerful rowhammer attacks. Examples of such attacks include privilege escalation [6, 10, 11, 24, 27, 29], cryptography breaking [5, 23], ASLR defeating [8], sandbox escaping [11, 22, 24], and denial-of-service [14, 19, 30] etc. Furthermore, instead of executing code locally to induce bit flips, new rowhammer attacks have been effectively demonstrated by only sending network packets [19, 26].

Many of the rowhammer attacks are performed on the x86 architecture. As ARM has become the *de facto* architecture used in mobile computing devices, there have been attempts to translate the x86-targeted rowhammer exploitation techniques to ARM. However, as discussed in [27], this adaption is non-trivial.

Among the many technical challenges, how to trigger the rowhammer bug on ARM-based devices needs to be addressed first and foremost; otherwise, even if the underlying DRAM were extremely vulnerable to hammering, one would still not be able to exploit the bug for a successful rowhammer attack. In order to trigger the rowhammer bug, a prerequisite is the ability to access the same location in DRAM rapidly. However, due to the presence of the CPU caches, most of the memory accesses to the same location can hardly reach the DRAM. Thus, how to circumvent the effect of the caches becomes the key to triggering the bug. On x86, three approaches to coping with the cache effect have been proposed: flushing the cache by using the `CLFLUSH` instruction [15], bypassing the cache by using certain non-temporal memory access instructions [22], and evicting the cache by using some good cache eviction strategies [4, 11]. However, it is either impractical or somehow not appreciated to adapt these approaches to ARM (more details are described in later sections). As a result, two approaches from other angles have been proposed to effectively address the cache effect on ARM: taking advantage of the user accessible DMA buffers on

ASHES'18, October 19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5996-2/18/10…$15.00
https://doi.org/10.1145/3266444.3266454

the Android system to bypass the CPU caches [27]; and, evicting the relatively simple caches of the integrated GPU to achieve fast accesses to the main memory [8].

In this paper, we revisit the problem of how to trigger the rowhammer bug on ARM-based devices, by carefully investigating whether it is possible as well as reasonable to translate the original x86-oriented approaches to ARM. We share our findings and point out some previously overlooked factors.

The main contributions of this paper are: (1) We provide a thorough study of the unprivileged ARMv8-A cache maintenance instructions, and give two previously overlooked reasons to support their use in rowhammer attacks; (2) We present a previously undiscovered instruction that can be exploited to trigger the rowhammer bug on many ARM-based devices; (3) We discuss a potential approach to quickly evicting the CPU caches; (4) We perform experimental evaluations to demonstrate the effectiveness of our findings.

The rest of this paper is organized as: Section 2 briefly sets the background; Section 3 formulates the problem considered in this paper; Section 4 presents our investigation as well as our findings; Section 5 evaluates different approaches of this paper; Section 6 describes possible mitigation techniques; Section 7 gives the related work; and Section 8 concludes this paper and states some future work.

2 BACKGROUND

In this section, we present some necessary background information on targeted ARM processors, DRAM, and the rowhammer bug as well as attacks. Additionally, we present the two existing approaches to triggering the rowhammer bug on ARM-based devices.

2.1 ARMv7-A v.s. ARMv8-A

ARM is a general term for a family of RISC architectures. Rowhammer attacks on ARM generally refer to such attacks on consumer devices or network infrastructures that are based on either ARMv7-A or ARMv8-A processors. The ARMv7-A architecture is only 32-bit, while the ARMv8-A architecture, released in 2011, supports both 32-bit and 64-bit computing. Although ARMv7-A processors are still in use, ARMv8-A processors have already become the mainstream processors currently used in the industry. In this paper, we focus on the rowhammer approaches targeting the ARMv8-A architecture.

2.2 DRAM Organization

Modern computing devices use DRAM as the main memory. For better memory bandwidth, DRAM is often partitioned into multiple channels. Each channel can be associated with multiple ranks, and each rank has multiple banks. As depicted in Fig. 1, each bank can be viewed as a two-dimensional array of cells, organized in rows and columns. Each cell consists of a capacitor and a transistor, where the capacitor is either charged or uncharged to represent a single binary value[1], and the transistor is used to access the capacitor. In each bank, there is also a row buffer, which can hold the contents of a single row. To access a cell, the corresponding row has to be activated first to put the contents of the row into the row buffer, and then the access is served from the row buffer. An activated

[1]Depending on the implementation, some cells use the charged state to represent '1', while other cells use the discharged state to represent '1'.

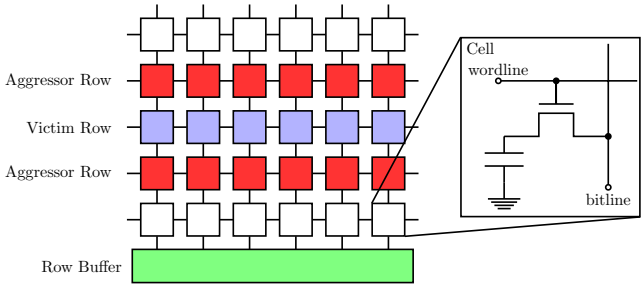

Figure 1: DRAM organization and an illustration of double-sided rowhammer, where the red aggressor rows sandwich the blue victim row.

row remains in the row buffer until being closed by the memory controller, and before then, consecutive accesses to that row will be served directly from the row buffer. Depending on what memory controller policy is being used, an active row can be closed due to different reasons: if the memory controller uses an "open-page" policy, the active row will not be closed until a different row in the same bank is accessed; on the other hand, if a "closed-page" policy is employed, the memory controller can proactively close the row to optimize performance [10, 19].

Without intervention, a cell can only keep its charged state for a short period of time, because its capacitor leaks the charge over time. In order to prevent any data loss, the cells must be refreshed regularly. DDR3 and DDR4 specifications require that the refresh interval must not be longer than 64ms. Normally, the refresh interval is between 32ms to 64ms.

2.3 Rowhammer Bug and Attacks

As the cells in DRAM are getting increasingly smaller and denser, the overall DRAM reliability is significantly impacted in a negative way. First thoroughly studied in [15], the rowhammer bug has become the most well-known DRAM reliability issue: When a DRAM row is repeatedly activated and closed (namely, hammered) enough times within a refresh interval, one or more bits in its physically adjacent rows can be flipped to the opposite value[2]. Usually, a row that is hammered is called an aggressor row, and a row that has flipped bits is called a victim row.

Since many of the memory controllers use "open-page" policies, to trigger the rowhammer bug on such systems, two aggressor rows in the same bank need to be alternately activated. If the two aggressor rows are not intentionally chosen to "sandwich" a row, it is termed as single-sided rowhammer. On the other hand, if the two aggressor rows lie on both sides of another row, it is called double-sided rowhammer (Fig. 1 shows an illustration of double-sided rowhammer). As demonstrated in practice, the double-sided rowhammer is much more effective and efficient than the single-sided rowhammer [24]. Some new memory controllers may use "closed-page" policies, and in such cases even one aggressor row is

[2]The large current coupled with toggling the activation of a row repeatedly and rapidly accelerates the discharging rate of cells in the physically adjacent rows. Before the next refresh, if too much charge in a cell has been leaked, the stored bit information will be lost, namely the bit is flipped.

sufficient to induce bit flips around the row, which is called one-location rowhammer [10].

Because the rowhammer bug allows one to modify the contents of a row without accessing it, severe security risks are posed. Since the discovery of this devastating hardware vulnerability, many rowhammer attack vectors have been developed by exploiting the rowhammer bug to compromise the security defenses of a system. Usually, to successfully perform a rowhammer attack, an adversary not only needs the ability to trigger the rowhammer bug on the targeted system, but also needs to be capable of steering the targeted security-critical data to vulnerable rows for exploitation. Several methods for helping exploit the rowhammer bug have been developed, such as memory spraying [24], memory grooming [27], memory waylaying [10], and memory ambushing [7]. Nevertheless, having approaches to triggering the rowhammer bug is always the prerequisite for a successful rowhammer attack.

2.4 Existing Rowhammer Approaches on ARM

As mentioned before, it is very challenging to adapt the x86-oriented rowhammer approaches to ARM. For example, an undisclosed proprietary pseudo-random cache replacement policy is used in ARM processors, so the most efficient and effective cache eviction strategy has to be empirically searched [9, 20]. However, it has been shown that even the best eviction strategy that can be found is still too slow to trigger the rowhammer bug [27].

To address the problem of quickly accessing the DRAM to trigger the rowhammer bug on ARM, two approaches have been proposed so far. Since memory regions allocated for I/O devices are often marked as non-cacheable to avoid any use of stale data, the first approach takes advantage of the DMA buffers exposed to the userspace by the Android ION memory management interface [27]. (However, a recently proposed mitigation has made the rowhammer bug triggered in this way non-exploitable [28].) The second approach relies on the observation that the integrated GPU in an ARM SoC (system-on-chip) often has relatively simple and small caches with a deterministic replacement policy; thus, by cleverly evicting the integrated GPU cache, the main memory can be quickly accessed [8].

3 PROBLEM STATEMENT

In this paper, we only focus on finding more approaches to triggering the rowhammer bug on platforms that use ARMv8-A processors. In other words, we do not consider how to identify addresses for the most efficient double-sided rowhammer, nor do we investigate how to exploit the rowhammer bug to compromise a system. Specifically, we will try to answer the following two questions:

- Is it possible as well as reasonable to adapt the original x86-oriented rowhammer approaches to ARMv8-A?
- Is there any "good" special ARMv8-A instruction that can be used to effectively hammer the underlying DRAM? If so, how can it be used?

4 NEW EXPLORATION

In this section, we first investigate the rowhammer methods based on some unprivileged cache maintenance instructions, and provide some arguments to support their practicability and "tenaciousness". Then, we present how to utilize a commonly-found instruction to bypass the cache in most of the ARMv8-A processors, which enables us to hammer the DRAM easily. In addition, we discuss a possible method for triggering the rowhammer bug by evicting the CPU caches.

4.1 Flushing/Cleaning the Cache

The original rowhammer approach targeting the x86 architecture uses the unprivileged `CLFLUSH` instruction to flush the data associated with an address from the cache hierarchy [15]. The availability of such an instruction greatly facilitates an adversary to repeatedly access the same DRAM locations at a high frequency. While the ARM instructions managing caches (termed as cache maintenance instructions) are privileged in all of the ARMv7-A processors, they have been exposed to the userspace since ARMv8-A. Therefore, it becomes feasible to use rowhammer approaches similar to the original `CLFLUSH`-based one on devices using ARMv8-A processors.

```
put addr1 into X9
put addr2 into X10
for i := 0 to N − 1 do
    LDR X0, [X9]
    LDR X0, [X10]
    DC CIVAC, X9
    DC CIVAC, X10
    DSB 0xB
```

(a) using `DC CIVAC`

```
put addr1 into X9
put addr2 into X10
for i := 0 to N − 1 do
    STR X0, [X9]
    STR X0, [X10]
    DC CVAC, X9
    DC CVAC, X10
    DSB 0xB
```

(b) using `DC CVAC`

Figure 2: Using cache maintenance instructions to perform rowhammer, where N is the number of iterations of the hammering loop.

The unprivileged ARMv8-A cache maintenance instructions can be classified into two categories[3]:

- Instructions related to the Point of Unification (PoU): the `DC CVAU` instruction cleans data cache by virtual address to PoU, and the `IC IVAU` instruction invalidates instruction cache by virtual address to PoU.
- Instructions related to the Point of Coherency (PoC): the `DC CIVAC` instruction cleans as well as invalidates data cache by virtual address to PoC, and the `DC CVAC` instruction only cleans data cache by virtual address to PoC.

PoU is the point at which the instruction and data caches are guaranteed to see the same copy of a memory location, while PoC is the point at which all processing elements that can access memory are guaranteed to see the same copy of a memory location [18]. Although PoU and PoC are not explicitly specified in the reference manual, typically PoU is defined as the unified L2 cache and PoC is specified as the main memory. Therefore, we can only use the PoC-related cache maintenance instructions as the counterpart of the x86 `CLFLUSH` instruction. Fig. 2 shows two examples that demonstrate the original x86-oriented rowhammer approach on ARMv8-A,

[3]There is actually another unprivileged cache maintenance instruction `DC CVAP` that is related to the Point of Persistence (PoP), but we omit its discussion, since PoP is either equal to PoC or associated with non-volatile storage.

where two virtual addresses *addr1* and *addr2* are physically mapped to two different DRAM rows in the same bank.

Note that, in Fig. 2 (b), it is each of the `DC CVAC` instructions that accesses the underlying DRAM in each loop iteration. The first two store instructions (i.e. `STR`) always hit in the L1 data cache after the first iteration, and make the corresponding cache lines dirty. In other words, if the `DC CVAC` instruction is used, it has to be paired up with a store instruction; otherwise, the cache lines will not be dirty, and by cleaning them, nothing will happen to the memory system. On the contrary, if the `DC CIVAC` instruction is used, it can be paired up with either load or store instructions. (For instance, in Fig. 2 (a), two load instructions `LDR` are used.) This is because the execution of the `DC CIVAC` instruction not only cleans the cache lines, but also invalidates these cache lines; as a result, the memory blocks will always be fetched from the memory in the next iteration. In addition, the last `DSB` instructions in both Fig. 2 (a) and (b) are memory barrier instructions, which force all pending load, store, and cache maintenance instructions to be completed before the program execution continues.

Although these exploitable cache maintenance instructions are unprivileged by default, there is a system control register which can be set by the kernel to disable the use of these instructions in the userspace [18]. This is why the use of such instructions is not considered as a good rowhammer (or side-channel) exploitation primitive [27, 31]. However, here we argue that it is in fact a simple and good facility due to two main reasons.

(1) *Disabling the use of the exploitable cache maintenance instructions will make certain applications non-runnable.* Since the ARM architecture does not maintain coherency between the instruction and data caches, flushing the cache is a necessary step towards supporting self-modifying code (e.g., Just-in-Time compilation). On Linux systems (including Android), the GCC builtin function __builtin___clear_cache() is called to flush the cache. Under the hood, the function makes a system call, whose system call number is `0xF0002`, on ARMv7-A. However, on ARMv8-A, the compiler will generate a sequence of instructions using the PoU-related `DC CVAU` and `IC IVAU` instructions to mange the cache directly in the userspace. The differences are illustrated in Fig. 3. There is only one bit in the system control register that enables/disables the use of all unprivileged cache maintenance instructions in the userspace. Therefore, disabling the exploitable PoC-related cache maintenance instructions will unfortunately also make the needed PoU-related ones unusable in the userspace, which will cause problems in many existing applications.

(2) *It may be very difficult to disable these instructions on certain platforms.* We have noticed that there is a cache maintenance exception handler in the Linux kernel introduced since version 4.8 for 64-bit ARMv8-A processors[4]. (Linux kernel 4.10 has been used in the latest Android.) In this case, even if these exploitable cache maintenance instructions were disabled by a kernel module, the exceptions raised by executing such instructions would trap into the kernel and the handler would use the corresponding instructions to take care of the desired cache operations[5]. Note that, although such handlers are designed to run as fast as possible, compared to single instructions, their execution time may be still too long to trigger the rowhammer bug on some platforms. A parallelization technique can be used to address this problem, which is described in Section 5.2.

Based on the two above-mentioned (yet overlooked) reasons, we believe that the PoC-related cache maintenance instructions can be relied on to hammer the DRAM rows on ARMv8-A processors.

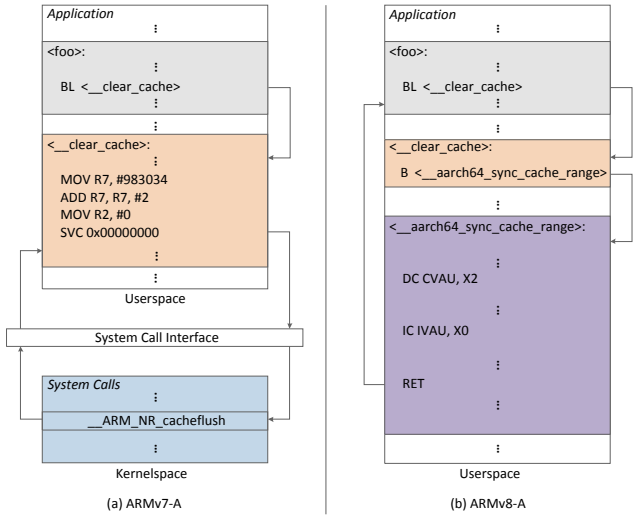

Figure 3: __builtin___clear_cache() used in a userspace function foo() will invoke a library function __clear_cache() that will (a) make a system call to *__ARM_NR_cacheflush* (i.e., `0xF0002`) on ARMv7-A; yet will (b) call another library function __aarch64_sync_cache_range() to clear the cache all in user mode on ARMv8-A.

Also, note that, although the `DC CIVAC` instruction is the one that is normally mentioned and compared in the literature, we find that the `DC CVAC` instruction is actually a much faster and thus better option for being used in rowhammer attacks, as evaluated in Section 5. On some platforms, the `DC CIVAC` instruction may be too slow to trigger the rowhammer bug. In that case, we can take advantage of parallelization to make the `DC CIVAC`-based approach work.

4.2 Bypassing the Cache

Data referenced by a program can be **temporal** (namely, the data will be used again in the near future), or **non-temporal** (namely, the data are used only once but not again in the near future). For example, streaming data are usually non-temporal and should not be cached in order to avoid cache "pollution". Therefore, some architectures introduce non-temporal memory access instructions, which can be used to directly access the memory and prevent non-temporal data from "polluting" the cache. This bypassing-the-cache

[4]The trap handler is implemented in arch/arm64/kernel/traps.c.

[5]The `DC CVAC` instruction is treated as the `DC CIVAC` instruction in the handler.

feature can greatly facilitate rowhammer attacks, as shown in [22] on the x86 architecture.

Although the ARMv8-A instruction set contains non-temporal load and store instructions (which are the LDNP and STNP instructions respectively), it is believed that they are not useful for triggering the rowhammer bug. As stated in [27], these ARMv8-A non-temporal memory access instructions only provide a hint to the memory system that data caching is not required, and in practice, the data are still found cached.

However, we discover that there is another instruction, the DC ZVA instruction, which can be exploited to bypass the cache in some ARMv8-A processors. The DC ZVA instruction is introduced to efficiently zero out a block of memory[6]. The ARMv8-A architecture reference manual makes no statements about whether or not the DC ZVA instruction causes allocation to any particular level of the cache [18]; yet, we notice that the DC ZVA instruction in our tested Cortex-A53 processors does not cause an L1 or L2 allocation, *if it misses in the cache hierarchy*, namely it writes zero directly to the main memory. Indeed, the Cortex-A53 technical reference manual confirms this behavior [17].

The DC ZVA instruction is unprivileged, so it can be executed in the userspace. In order to trigger the rowhammer bug, we need to use the DC ZVA instruction to alternately activate two different rows in the same bank fast enough. Given two virtual addresses *addr1* and *addr2*, which are physically mapped to two different DRAM rows of the same bank, the pseudo code shown in Fig. 4 can be used to achieve the goal. Both code in Fig. 4 (a) and (b) can work, and the only difference lies in that code in Fig. 4 (b) does not use the memory barrier instruction DSB. In terms of CLFLUSH-based rowhammer attacks on x86, Xiao *et al.* have witnessed that it is more effective to induce bit flips without memory barrier instructions [29]. Interestingly, we also observe similar results when conducting the experiments on ARM, which will be detailed in Section 5.

When using the DC ZVA instruction to trigger the rowhammer bug, one requirement is that we have to make sure the blocks being zeroed are not cached in the hierarchy before the hammering loop; otherwise, the underlying DRAM will not be accessed, and only the cached copies are zeroed and marked as dirty. This is totally different from using the non-temporal store instructions for rowhammer attacks on x86, where cached memory accesses to the same addresses written by the non-temporal store instructions need to be performed in each loop iteration so as to avoid undesired write combining behavior [22]. To meet this requirement, we can either simply use the DC CIVAC instruction or access corresponding cache eviction sets, as indicated by the third and fourth lines in Fig. 4 (a) and (b). A cache eviction set is a set of congruent addresses that are mapped to the same cache set. (To speed up eviction a bit, we can follow the work described in [11, 20] to derive good cache eviction strategies.)

Note that, so far, we have only found the DC ZVA instruction of the Cortex-A53 processors has the aforementioned *no-write-allocate* property. Cache misses incurred by DC ZVA in other 64-bit Cortex-A family members (like Cortex-A57) still cause allocations in the hierarchy. However, we argue that this situation *does not* limit

[6]As stated in the ARMv8-A architecture reference manual [18], despite its mnemonic, DC ZVA is not a data cache maintenance instruction.

```
put addr1 into X9
put addr2 into X10
invalidate/evict addr1
invalidate/evict addr2
for i := 0 to N − 1 do
    DC ZVA, X9
    DC ZVA, X10
    DSB 0xB
```

(a) w/ mem. barriers

```
put addr1 into X9
put addr2 into X10
invalidate/evict addr1
invalidate/evict addr2
for i := 0 to N − 1 do
    DC ZVA, X9
    DC ZVA, X10
```

(b) w/o mem. barriers

Figure 4: Using the DC ZVA instruction to perform rowhammer, where N is the number of iterations of the hammering loop.

the practical utility of our DC ZVA-based approach too much. The main reason is that the use of Cortex-A53 actually prevails among the 64-bit ARM-powered computing devices (e.g., smartphones and tablets): Due to its high power efficiency and good performance, Cortex-A53 is either used in a standalone fashion in many low-/middle-end products, or integrated in the big.LITTLE architecture alongside more powerful cores for many high-end products. Tab. 3 in the appendix shows the dominant presence of Cortex-A53 in the popular SoCs used by major mobile device companies.

Despite its effectiveness in bypassing the cache, the direct use of the DC ZVA instruction from userspace can be disabled by the kernel. Unlike the cache maintenance instructions, there is no trap handler in the kernel to deal with the exception caused by executing the disabled DC ZVA instruction, namely, the disabling-based countermeasure works in this case. Nevertheless, we have not seen any operating system that disables the use of the instruction by default, so it is still a valid exploitation primitive. Moreover, there are many applications/libraries using this instruction for performance benefits, e.g., the instruction is found in the memset() function of the Bionic, glibc, uClibc, and Newlib C library implementations, which makes it very difficult, if not impossible, to disable the use of this instruction.

4.3 Discussion on Evicting the Cache

Except for the work presented in [8], which triggers the rowhammer bug on ARM platforms by cleverly evicting GPU FIFO caches, to the best of our knowledge, there is no existing work that can successfully induce this hardware fault on ARM by only evicting CPU caches. As mentioned before, the main reason is that a good eviction strategy can be much slower than required for triggering the bug [27]. Due to the pseudo-random replacement policy and relatively large associativity used in the last level cache (LLC), it is very challenging to speed up the eviction process. As reported in [20], thousands of clock cycles may be consumed when reaching a desirable eviction rate.

Recently, an undocumented transparent lockdown feature, named as autolock, has been reported in [9]. It is claimed that cache lines in the inclusive LLC will be implicited locked if their contents are also in core-private caches, in which case cross-core cache eviction will be impeded. According to [9], this autolock feature seems to

appear in many ARMv8-A processors. Since almost every widely used ARMv8-A processor has at least quad cores, one idea is to spawn as many threads as there are cores to lock down some cache lines of the same cache sets in LLC; the effect will be equivalent to reducing the associativity of LLC. Thus, the eviction process can be accelerated. (A similar idea taking advantage of Intel's cache allocation technology has been demonstrated on x86 platforms [1].)

Take the commonly-seen quad-core Cortex-A53 for an example, in which the autolock feature is reported to exist [9]. A Cortex-A53 processor by design has a unified L2 cache which is 16-way set associative, and each core privately has a 2-way instruction cache and a 4-way data cache. The L2 cache is only inclusive to the L1 instruction caches, but not to the L1 data caches [9, 25][7]. Thus, in terms of an L2 cache set, each core can try to lock down two ways of the set (by a carefully designed return-oriented method [31]). Since only eight ways in the cache set are left for allocation, the speed of the eviction process may at least be doubled. Unfortunately, we do not observe the transparent locking behavior on our experimental Cortex-A53 platform. On the other hand, even if these cache lines were successfully locked, it is not clear to us whether the 2x speedup could possibly trigger the rowhammer bug, since each good eviction might still take more than a thousand clock cycles. We leave this as our future work for further investigation.

5 EVALUATION

In this section, we evaluate the approaches described in Section 4 by testing their ability to trigger the rowhammer bug. In order to facilitate comparison, we will use the number of unique bits that are flipped as a metric.

We carry out all our experiments on a single-board computer equipped with a quad-core Cortex-A53 processor (Amlogic S905X SoC) and 2GB LPDDR3 memory. A Linux runs on this board, whose kernel version is 4.14. Since we only concentrate on approaches to triggering the rowhammer bug on ARM in this paper, for the sake of simplicity, we just take advantage of the /proc/self/pagemap interface to easily acquire the physical addresses for double-sided rowhammer.

5.1 Effectiveness of Approaches

At first, we perform 4 pairs of experiments, which cover well the combinations of the hammering techniques discussed in this paper. Inside each pair, the only difference is whether the memory barrier instruction `DSB` is used. The experiment pairs are listed as follows:

- `{LDR + DC CIVAC + DSB}` and `{LDR + DC CIVAC}`
- `{STR + DC CIVAC + DSB}` and `{STR + DC CIVAC}`
- `{STR + DC CVAC + DSB}` and `{STR + DC CVAC}`
- `{DC ZVA + DSB}` and `{DC ZVA}`

For these experiments, 10,000 pairs of aggressor rows are selected to perform double-sided rowhammer. The results are shown in Fig. 5, where the horizontal axis represents the hammered aggressor row pairs and the vertical axis gives the number of unique bit flips (both $1 \rightarrow 0$ and $0 \rightarrow 1$).

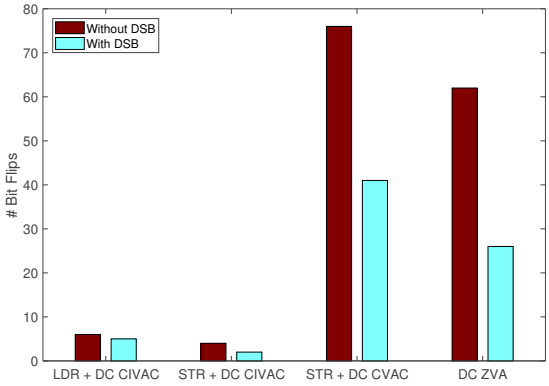

Figure 5: Number of bit flips induced by different hammering techniques.

From the results, we can clearly see that all the approaches are effective in inducing bit flips, but with different performance (i.e., how many unique bits can be flipped). We find the four approaches based on the most well-known `DC CIVAC` instruction perform much worse than other approaches – they only induce 6 bit flips at most. (Later, we will show how to make them more effective and efficient by taking advantage of parallelization.) The best performance is given by the `{STR + DC CVAC}` approach, which can flip 76 bits. In addition, the `{STR + DC CVAC + DSB}` approach as well as both the `DC ZVA`-based approaches also perform well. This phenomenon may be due to the fact that the `DC CIVAC` instruction makes the subsequent memory access take much longer time than other cases, as cache lines have been invalidated and need to be refilled.

When using the combination of the `STR` and `DC CIVAC` instructions in the hammering loop, **four** accesses to the DRAM should be generated in a single iteration – two of them are due to the cache misses of the `STR` instructions, and the other two are due to cleaning the dirty cache lines by the `DC CIVAC` instructions. At first, we have expected that this situation may help the hammering efficiency. However, by comparing the results in Fig. 5, it shows this combination actually reduces the number of bit flips, which is contrary to our expectation. We can also observe that the use of the memory barrier instruction `DSB` decreases the number of bit flips in all the cases, particularly, with respect to the approaches based on the `DC CVAC` and `DC ZVA` instructions. A similar observation is also reported in [29] for the x86 architecture. Therefore, they show that memory barrier instructions are not necessarily required when performing rowhammer on either x86 or ARM.

According to [27], if the time between two reactivations of an aggressor row is above 260 ns, it can barely trigger the rowhammer bug on ARM. We measure the time needed by a hammering loop iteration, and the times for different approaches are shown in Tab. 1. From the measured times, we can observe that a hammering loop iteration needs more than 260 ns only in the last two approaches (i.e., by combining the `STR` and `DC CIVAC` instructions). However, as mentioned above, there are four alternating accesses to the two aggressor rows within a single loop iteration in terms of the last two

[7] The Cortex-A53 reference manual does not explicitly describe the inclusiveness of its L2 cache, and conflicting statements on the data cache side exist in the literature. Some state the L2 cache is inclusive to the L1 data caches [31], while others claim it is non-inclusive [9, 25]. We confirm that the Cortex-A53 L2 cache is not inclusive on the data cache side through many experiments, but we are not sure whether it is non-inclusive or exclusive (as some exclusiveness behavior appears).

Table 1: Time of a hammering loop iteration.

Technique	Average Time
{DC ZVA}	~70 ns
{DC ZVA + DSB}	~72 ns
{STR + DC CVAC}	~99 ns
{STR + DC CVAC + DSB}	~119 ns
{LDR + DC CIVAC}	~249 ns
{LDR + DC CIVAC + DSB}	~250 ns
{STR + DC CIVAC + DSB}	~313 ns (two)
{STR + DC CIVAC}	~331 ns (two)

approaches, namely, each row is hammered **twice**. Even though the time may not be evenly distributed within a loop iteration, the time between two reactivations of the same row should be less than 260 ns.

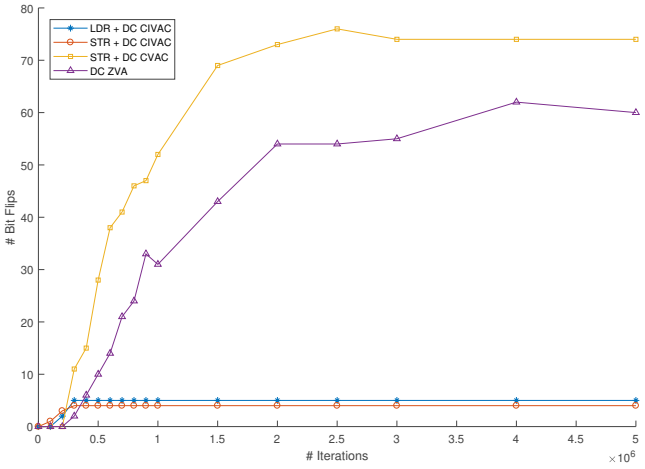

Figure 6: The impact of the number of iterations on the number of bit flips in terms of different hammering techniques.

Another important factor we evaluate is how many iterations a pair of aggressor rows should be hammered. Interestingly, we have found that the number of hammering loop iterations may have a significant impact on flipping bits in both the DC ZVA- and DC CVAC-based techniques. Fig. 6 shows the relationship between the number of iterations and the number of bit flips for different approaches (without using the DSB instruction) adopted to hammer the same set of aggressor row pairs in a double-sided manner. As we can observe, when some pairs of aggressor rows are hammered about 200,000 times, bits start flipping. However, in terms of the DC ZVA- and DC CVAC-based techniques, when the number of loop iterations increases, more bits become flipped. (When the number of iterations is above 2,500,000, there are much more bit flips than that of iterating below 1 million times.) Since a hammering loop with millions of iterations spans several refresh intervals, we conjecture that some bits can be flipped with a relatively small probability when aggressor rows are hammered above certain times within a refresh interval (yet the DC CIVAC-based approaches are too slow to reach this limit) and these bits become very likely to be flipped after several consecutive attempts.

5.2 Disabling Special Instructions and Parallelization

We also use a kernel module to disable the use of the DC CIVAC, DC CVAC, and DC ZVA instructions in the userspace. As expected, the hammering process relying on the DC ZVA instruction is terminated due to an illegal instruction exception; yet, the other processes using the cache maintenance instructions are still runnable. To verify this is because there exists a trap handler for cache maintenance instructions, we also repeat the same procedure on another installed Linux, whose kernel version is 4.4. In this case, all the processes are terminated, which proves our reasoning.

We perform the first three experiment pairs again after the direct use of the corresponding instructions is disabled. Unfortunately, we observe that the execution time of a hammering loop iteration becomes too long to trigger the rowhammer bug (e.g., ~560 ns in terms of the {STR + DC CIVAC}). However, we discover that, with parallelization, it becomes possible to trigger the rowhammer bug again after the cache maintenance instructions are disabled. Since almost every widely used ARMv8-A processor has at least quad cores, we can parallelize the hammering loop using two threads, each of which runs on a separate core and accesses one of the aggressor rows. Therefore, the independent memory access requests from different cores may flow into the memory controller at almost twice the speed than before.

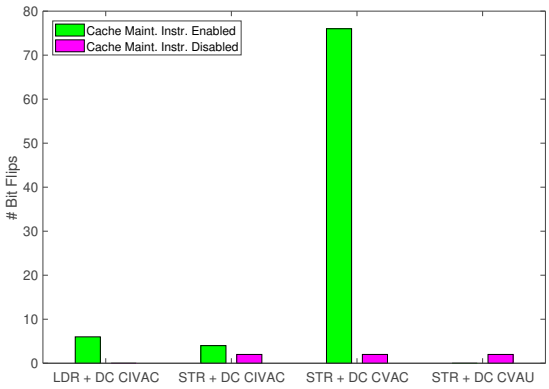

Figure 7: The number of bit flips induced before and after the unprivileged instructions are disabled (parallelization is used after the instructions are disabled).

By using this parallelization technique, the hammering loop can again induce bit flips after the cache maintenance instructions are disabled. The results of parallel hammering attempts are shown in Fig. 7. (For comparison, the numbers of bit flips induced before the instructions are disabled are also shown in the figure.) Since the overhead of the trap handler is still very large, the numbers of bit flips become less than before in all the cases. Nevertheless, we can still observe that it is feasible to trigger the rowhammer bug when using a parallelized hammering process. Surprisingly, we do not identify any bit flip in the case of the {LDR + DC CIVAC} approach, and we notice that the PoU-related DC CVAU instruction even becomes capable of reaching PoC to hammer the DRAM. By carefully studying the trap handler, we find that both the DC CVAC

and DC CVAU instructions are treated as the DC CIVAC instruction in that handler. This explains why they perform very similarly and the DC CVAU instruction can be used to trigger the rowhammer bug when the use of the unprivileged cache maintenance instructions in user mode is disabled.

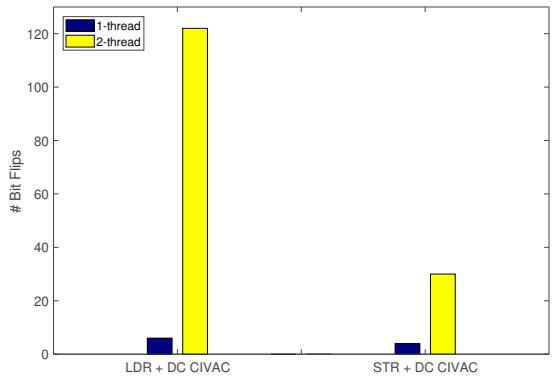

Figure 8: The number of bit flips induced by one hammering thread and two hammering threads when cache maintenance instructions are still usable in user mode.

Additionally, this parallelization technique can be applied to the DC CIVAC-based approaches to increase their performance even if the cache maintenance instructions are not disabled for use in user mode. Fig. 8 shows the results when using two threads to repeat the experiments with respect to the {LDR + DC CIVAC} and {STR + DC CIVAC}. From the results, we can see the number of bit flips grows drastically when two threads are used. (In order to facilitate comparison, the numbers induced in single-thread hammering processes are also shown in the figure.) Interestingly, in this case, the combination of the LDR and DC CIVAC instructions performs much better than that of the STR and DC CIVAC instructions (122 and 30 bit flips respectively). Originally, we think this may be due to synchronization issues. However, after a few trials, the big difference still remains. We will investigate this interesting phenomenon in detail in the future.

5.3 Monitoring Cache Misses

Many performance monitoring counter-based rowhammer detection approaches have been proposed for x86 [4, 12, 13, 21]. Because a large number of cache misses will be incurred if cache lines are repeatedly invalidated during a rowhammer attempt, the basic idea of such approaches is to capture and analyze activities that generate anomalous amount of cache misses. Thus, these defense techniques inevitably depend on the cache miss event counters inside the CPU. There is a performance monitoring unit (PMU) in each ARMv8-A processor, which has a register counting how many LLC misses have been encountered. As a result, these performance monitoring counter-based rowhammer detection approaches can also be applied to ARMv8-A.

We expect that the approaches based on the DC CVAC instruction will not increase the LLC cache miss counter, since the DC CVAC instruction only cleans the dirty cache line but does not invalidate it. Moreover, we anticipate other approaches based on either the DC CIVAC instruction or the DC ZVA instruction will increase the counter. The reason in terms of the DC CIVAC instruction is obvious, and the reason why we expect such a result in terms of the DC ZVA instruction is that the DC ZVA instruction only accesses the main memory when it misses in the cache hierarchy.

Table 2: Number of L2 cache misses in 5,000,000 hammering loop iterations

Tech.	{LDR + DC CIVAC}	{STR + DC CIVAC}	{STR + DC CVAC}	{DC ZVA}
Miss	10,000,058	9,999,725	70	5

For each pair of the experiments performed in Section 5.1, we measure how many L2 (which is the LLC in the used processor) cache misses are incurred during 5,000,000 iterations of the hammering loop. Since the memory barrier instruction does not affect L2 cache misses at all, we only list the numbers corresponding to the techniques without using the DSB instruction in Tab. 2. To our surprise, one of our anticipations is not correct – the execution of the DC ZVA instruction does not increase the counter although it misses in the cache. On the other hand, it means the cache miss counter-based rowhammer detection approaches will not work with respect to the two most efficient rowhammer-triggering instructions. (Most of the L2 cache misses in the last two columns are probably due to context switches.) Another interesting phenomenon is the number of L2 cache misses when using the combination of the STR and DC CIVAC instructions is less than 10,000,000. This may be because some writing operations are reordered and combined in the memory system since the DSB instruction is not used.

6 MITIGATION APPROACHES

There have been several mitigation techniques being proposed to cope with the rowhammer problem in general. Since the rowhammer bug is a hardware reliability issue, some of the approaches try to solve the problem from the perspective of hardware: By using error correcting codes (ECC) memory, single-bit flips can be detected and corrected, but it cannot deal with multiple-bit flips. In [15], a probabilistic adjacent row activation (PARA) approach is proposed to refresh neighboring rows on a row activation with a low probability so that the potential victim rows are very likely to be refreshed during a rowhammer attack. However, memory controller modifications need to be made, which prevents it from being practically used. The target row refresh (TRR) technique is introduced in the LPDDR4 standard [3], which identifies possible victim rows by counting the number of row activations, and refreshes these rows to prevent bit flips. However, TRR is not a mandatory feature, so some manufacturers may not adopt it in their products. Besides, all the hardware-based mitigation approaches are really hard to deploy on existing ARM platforms.

To mitigate the effectiveness of the rowhammer methods discussed in this paper, the simplest software-based approach is to increase the DRAM refresh rate. Different from PCs on which the refresh rate has to be changed via BIOS update, the memory controller on an ARM SoC is normally configurable through memory-mapped I/O. Although it will make rowhammer more difficult, increasing the refresh rate too much will hurt the system performance and

energy efficiency, which may not be acceptable in many cases. Another straightforward countermeasure is to disable the use of these exploitable instructions in the userspace. While this may be relatively easy with respect to the `DC ZVA` instruction, as discussed in this paper, the cache maintenance instructions can be hardly disabled in the userspace. However, dummy operations can be added into the cache maintenance trap handler to properly increase its execution time, so that it becomes very hard to rapidly flush the cache.

Since a lot of cache misses usually accompany a rowhammer attempt, performance monitoring counters can be used to facilitate detection [4, 12, 13, 21]. For instance, ANVIL uses the cache miss performance counter to capture suspicious activities that cause intensive cache misses, and selectively refresh possible victim rows after some analysis of the addresses associated to the cache misses [4]. However, as shown in Section 5.3, the rowhammer approach based on either the `DC CVAC` instruction or the `DC ZVA` instruction will barely cause cache misses, which makes the performance monitoring counter-based detection ineffective. Because special instructions are used in the proposed approaches, static binary analysis can in effect help to identify possibly malicious code [13]. After potentially malicious code is revealed, more advanced techniques or even human can get involved to analyze this code.

7 RELATED WORK

In [15], Kim *et al.* lay the groundwork of the rowhammer-related research, which thoroughly studies the rowhammer bug on DDR3 and points out concerns about security risks. It shows the first approach to triggering the rowhammer bug by using the `CLFLUSH` instruction. Since then, much work has been done regarding how to exploit such a hardware bug to compromise a system. In [24], the initial exploitation attempt is shown, which successfully utilizes the rowhammer bug to gain privilege escalation as well as escape the Google Chrome NaCl sandbox. In [22], non-temporal store instructions are used to bypass the cache for triggering the bug. In [11], it is demonstrated that the rowhammer bug can be triggered not only by the native code but also by some Javascript code running in a browser through evicting the cache. More powerful rowhammer attacks are also formulated by security practitioners. In [23], a general exploitation technique, Flip Feng Shui (FFS), is proposed and instantiated by using memory deduplication to break several cryptographic software. In [5], cryptographic software is also targeted, in which the rowhammer bug is exploited to generate a faulty RSA signature so that the secret key can be recovered. In [29], a graph-based approach to physical-to-DRAM address translation is illustrated, and how to break Xen paravirtualized memory isolation in a cross-VM setting is demonstrated. In [10], one-location rowhammer combined with Intel SGX and memory waylaying is shown to be able to defeat all the proposed rowhammer defenses. More recently, the possibility that rowhammer attacks can be performed through network has been proven [19, 26], which points out another research direction.

As mentioned before, there are two existing approaches that can trigger the rowhammer bug on ARM. In [27], DMA buffers, exposed to the userspace by the Android ION memory management interface, are utilized to bypass the cache. However, in [28], a mitigation approach is proposed to make the DMA-related hammerable area non-exploitable. In [8], fast accesses to the main memory are enabled by evicting the integrated GPU FIFO caches.

8 CONCLUSION

So far, the rowhammer bug has been found in various types of DRAM, which are deployed in numerous systems. The rowhammer-related research has been performed on x86 extensively. Due to the prevalence of ARM processors in the mobile computing devices and network infrastructures, we believe that it is equally important to perform similar research with respect to ARM. In this paper, we have shown that not every factor has been considered, and many of the overlooked conditions may have left possible vulnerabilities behind. In the future, we plan to investigate more on how to efficiently evict the cache hierarchy to trigger the rowhammer bug. Moreover, we will also study how to exploit such vulnerabilities to perform rowhammer attacks.

ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation (CNS-1739328). The authors would like to thank the anonymous reviewers for their comments and suggestions which help us improve the quality of the paper.

REFERENCES

[1] Misiker Tadesse Aga, Zelalem Birhanu Aweke, and Todd Austin. 2017. When good protections go bad: Exploiting anti-DoS measures to accelerate rowhammer attacks. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 8–13.

[2] Barbara Aichinger. 2015. DDR memory errors caused by Row Hammer. In *2015 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–5.

[3] JEDEC Solid State Technology Association. 2017. *Low Power Double Data Rate 4 (LPDDR4)*.

[4] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. 2016. ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. 743–755.

[5] Sarani Bhattacharya and Debdeep Mukhopadhyay. 2016. Curious case of rowhammer: flipping secret exponent bits using timing analysis. In *International Conference on Cryptographic Hardware and Embedded Systems*. 602–624.

[6] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2016. Dedup est machina: Memory deduplication as an advanced exploitation vector. In *2016 IEEE symposium on security and privacy (S&P)*. 987–1004.

[7] Yueqiang Cheng, Zhi Zhang, Surya Nepal, and Zhi Wang. 2018. Still Hammerable and Exploitable: on the Effectiveness of Software-only Physical Kernel Isolation. *CoRR* abs/1802.07060 (2018). arXiv:1802.07060 http://arxiv.org/abs/1802.07060

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

[9] Marc Green, Leandro Rodrigues-Lima, Andreas Zankl, Gorka Irazoqui, Johann Heyszl, and Thomas Eisenbarth. 2017. AutoLock: Why Cache Attacks on ARM Are Harder Than You Think. In *26th USENIX Security Symposium (USENIX Security 17)*. 1075–1091.

[10] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. 2018. Another Flip in the Wall of Rowhammer Defenses. In *2018 IEEE Symposium on Security and Privacy (S&P)*. 489–505.

[11] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2016. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. 300–321.

[12] Nishad Herath and Anders Fogh. 2015. These are Not Your Grand Daddy's CPU Performance Counters - CPU Hardware Performance Counters for Security. In *Black Hat Briefings*.

[13] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2016. MASCAT: Stopping Microarchitectural Attacks Before Execution. Cryptology ePrint Archive, Report

2016/1196. https://eprint.iacr.org/2016/1196.
[14] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. 2017. SGX-Bomb: Locking Down the Processor via Rowhammer Attack. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution (SysTEX '17)*. Article 5, 6 pages.
[15] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. 361–372.
[16] Mark Lanteigne. 2016. How Rowhammer Could Be Used to Exploit Weaknesses in Computer Hardware. http://www.thirdio.com/rowhammer.pdf.
[17] ARM Limited. 2016. *ARM Cortex-A53 MPCore Processor Technical Reference Manual*. Revision: r0p4.
[18] ARM Limited. 2017. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile*.
[19] Moritz Lipp, Misiker Tadesse Aga, Michael Schwarz, Daniel Gruss, Clémentine Maurice, Lukas Raab, and Lukas Lamster. 2018. Nethammer: Inducing Rowhammer Faults through Network Requests. *CoRR* abs/1805.04956 (2018). arXiv:1805.04956 http://arxiv.org/abs/1805.04956
[20] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *25th USENIX Security Symposium (USENIX Security 16)*. 549–564.
[21] Mathias Payer. 2016. HexPADS: A Platform to Detect "Stealth" Attacks. In *Proceedings of the 8th International Symposium on Engineering Secure Software and Systems - Volume 9639 (ESSoS 2016)*. 138–154.
[22] Rui Qiao and Mark Seaborn. 2016. A new approach for rowhammer attacks. In *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 161–166.
[23] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. 2016. Flip Feng Shui: Hammering a Needle in the Software Stack. In *25th USENIX Security Symposium (USENIX Security 16)*. 1–18.
[24] Mark Seaborn and Thomas Dullien. 2015. Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges. In *Black Hat Briefings*.
[25] Anand Lal Shimpi. 2013. Answered by the Experts: ARM's Cortex A53 Lead Architect, Peter Greenhalgh. https://www.anandtech.com/show/7591/answered-by-the-experts-arms-cortex-a53-lead-architect-peter-greenhalgh.
[26] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2018. Throwhammer: Rowhammer Attacks over the Network and Defenses. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 213–226.
[27] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. 2016. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. 1675–1689.
[28] Victor van der Veen, Martina Lindorfer, Yanick Fratantonio, Harikrishnan Padmanabha Pillai, Giovanni Vigna, Christopher Kruegel, Herbert Bos, and Kaveh Razavi. 2018. GuardION: Practical Mitigation of DMA-Based Rowhammer Attacks on ARM. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. 92–113.
[29] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. 2016. One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. In *25th USENIX Security Symposium (USENIX Security 16)*. 19–35.
[30] Shaza Zeitouni, David Gens, and Ahmad-Reza Sadeghi. 2018. It's Hammer Time: How to Attack (Rowhammer-based) DRAM-PUFs. In *Proceedings of the 55th Annual Design Automation Conference (DAC '18)*. 65:1–65:6.
[31] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. 2016. Return-Oriented Flush-Reload Side Channels on ARM and Their Implications for Android Devices. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. 858–870.

APPENDIX

Tab. 3 lists widely used 64-bit ARM-based SoCs, which are made by major chip manufacturers for mobile computing devices. Qualcomm, Samsung, Mediatek and Hisilicon take more than 90% of the market share in Android smart phones in 2017. From the table, we can see that 83.3% of the listed SoCs have Cortex-A53 cores. (Note that Kryo is also a customized version of Cortex-A53 by Qualcomm, but, due to our lack of Kryo-based devices, we have not verified if its `DC ZVA` instruction has the same behavior as Cortex-A53.) Therefore, Cortex-A53 is a very commonly used microarchitecture in the industry.

Table 3: Currently popular SoCs used in the industry (the SoCs having Cortex-A53 are highlighted)

SoC	Cores	SoC	Core
Qualcomm Snapdragon 410	Cortex-A53	Mediatek MT6735P	Cortex-A53
Qualcomm Snapdragon 415	Cortex-A53	Mediatek MT6737	Cortex-A53
Qualcomm Snapdragon 425	Cortex-A53	Mediatek MT6737T	Cortex-A53
Qualcomm Snapdragon 430	Cortex-A53	Mediatek MT6739	Cortex-A53
Qualcomm Snapdragon 435	Cortex-A53	Mediatek MT6750	Cortex-A53
Qualcomm Snapdragon 450	Cortex-A53	Mediatek MT6752	Cortex-A53
Qualcomm Snapdragon 610	Cortex-A53	Mediatek MT6753	Cortex-A53
Qualcomm Snapdragon 615	Cortex-A53	Mediatek MT8165	Cortex-A53
Qualcomm Snapdragon 616	Cortex-A53	Mediatek MT8173	Cortex-A72/-A53
Qualcomm Snapdragon 617	Cortex-A53	Mediatek MT8173C	Cortex-A72/-A53
Qualcomm Snapdragon 625	Cortex-A53	Mediatek MT8176	Cortex-A72/-A53
Qualcomm Snapdragon 626	Cortex-A53	Mediatek MT8732	Cortex-A53
Qualcomm Snapdragon 630	Cortex-A53	Mediatek MT8735	Cortex-A53
Qualcomm Snapdragon 636	Kryo 260	Mediatek MT8752	Cortex-A53
Qualcomm Snapdragon 650	Cortex-A72/-A53	Samsung Exynos 5433	Cortex-A57/-A53
Qualcomm Snapdragon 652	Cortex-A72/-A53	Samsung Exynos 7420	Cortex-A57/-A53
Qualcomm Snapdragon 660	Kryo 260	Samsung Exynos 7570	Cortex-A53
Qualcomm Snapdragon 808	Cortex-A57/-A53	Samsung Exynos 7578	Cortex-A53
Qualcomm Snapdragon 810	Cortex-A57/-A53	Samsung Exynos 7580	Cortex-A53
Qualcomm Snapdragon 820	Kryo	Samsung Exynos 7870	Cortex-A53
Qualcomm Snapdragon 821	Kryo	Samsung Exynos 7880	Cortex-A53
Apple A11 Bionic	Monsoon/Mistral	Samsung Exynos 7885	Cortex-A73/-A53
Apple A10X Fusion	Hurricane/Zephyr	Samsung Exynos 8890	Mongoose/Cortex-A53
Apple A10 Fusion	Hurricane/Zephyr	Samsung Exynos 8895	Mongoose/Cortex-A53
Apple A9X	Twister	HiSilicon Kirin 620	Cortex-A53
Apple A9	Twister	HiSilicon Kirin 650	Cortex-A53
Apple A8X	Typhoon	HiSilicon Kirin 655	Cortex-A53
Apple A8	Typhoon	HiSilicon Kirin 658	Cortex-A53
Apple A7	Cyclone	HiSilicon Kirin 659	Cortex-A53
Mediatek Helio P10	Cortex-A53	HiSilicon Kirin 930	Cortex-A53
Mediatek Helio P20	Cortex-A53	HiSilicon Kirin 935	Cortex-A53
Mediatek Helio P23	Cortex-A53	HiSilicon Kirin 950	Cortex-A72/-A53
Mediatek Helio P25	Cortex-A53	HiSilicon Kirin 955	Cortex-A72/-A53
Mediatek Helio P60	Cortex-A73/-A53	HiSilicon Kirin 960	Cortex-A73/-A53
Mediatek Helio X10	Cortex-A53	HiSilicon Kirin 960s	Cortex-A73/-A53
Mediatek Helio X20	Cortex-A72/-A53	HiSilicon Kirin 970	Cortex-A73/-A53
Mediatek MT6732	Cortex-A53	Nvidia Tegra X1	Cortex-A57/-A53
Mediatek MT6735	Cortex-A53	Nvidia Tegra K1 (Denver)	Denver
Mediatek MT6735M	Cortex-A53	Marvell Armada PXA1908	Cortex-A53