Return-Oriented Flush-Reload Side Channels on ARM and Their Implications for Android Devices

Xiaokuan Zhang Yuan Xiao Yinqian Zhang
Department of Computer Science and Engineering
The Ohio State University
{zhang.5840, xiao.465}@buckeyemail.osu.edu, yinqian@cse.ohio-state.edu

ABSTRACT

Cache side-channel attacks have been extensively studied on x86 architectures, but much less so on ARM processors. The technical challenges to conduct side-channel attacks on ARM, presumably, stem from the poorly documented ARM cache implementations, such as cache coherence protocols and cache flush operations, and also the lack of understanding of how different cache implementations will affect side-channel attacks. This paper presents a systematic exploration of vectors for FLUSH-RELOAD attacks on ARM processors. FLUSH-RELOAD attacks are among the most well-known cache side-channel attacks on x86. It has been shown in previous work that they are capable of exfiltrating sensitive information with high fidelity. We demonstrate in this work a novel construction of flush-reload side channels on last-level caches of ARM processors, which, particularly, exploits return-oriented programming techniques to reload instructions. We also demonstrate several attacks on Android OS (e.g., detecting hardware events and tracing software execution paths) to highlight the implications of such attacks for Android devices.

Keywords

Cache side channels; flush-reload

1. INTRODUCTION

Cache side-channel attacks have been gaining attraction in recent years, in part due to their noteworthy security implications in computing environment where processor caches are shared among mutually distrustful software programs, *e.g.*, public multi-tenant clouds. Due to the popularity of x86 processors in cloud data centers, most prior studies on cache side-channel attacks focus on x86 architectures. In contrast, much less research has been done on side-channel attacks on ARM architectures. Although it is tempting to presume similar attacks can be easily migrated from x86 to ARM processors, in fact due to significant differences in the cache design and implementation, conclusions drawn on Intel processors about these hardware-dependent security threats cannot be directly applied to ARM. To date, we have not seen much work on the exploitability of ARM caches in side-channel attacks. Such studies, however, are of growing significance due to the increasing popularity of ARM processors in mobile devices and even cloud servers [20].

In this paper, we present a systematic exploration of FLUSH-RELOAD side-channel attacks on ARM caches. FLUSH-RELOAD attacks have been extensively studied on x86 platforms and are well known for their high accuracy and efficiency. The adversary who has control of an application running on a shared computer system can exploit the unprivileged `clflush` instruction on x86 to FLUSH cache lines out of the entire cache hierarchy, and then measure the time to RELOAD it back. The key to the attack is that such flush operations, though taking virtual addresses as input, work on physical addresses of a cache line, so that cache lines shared with a victim application will also be evicted in the procedure. Therefore, the time to RELOAD the cache line back reveals whether this line has been recently (after FLUSH and before RELOAD) accessed, and thus loaded into the shared cache, by the victim application (*i.e.*, fast RELOAD) or not (*i.e.*, slow RELOAD).

However, replicating the FLUSH-RELOAD attacks on ARM is not as straightforward as one might imagine. The following research questions are yet to be explored: *First*, what is ARM's alternative for x86's unprivileged `clflush` instruction? *Second*, without a user-space accessible high-precision clock, *e.g.*, x86's `rdtsc`, how does the adversary measure time with high fidelity on ARM (to perform this timing attack)? *Third*, how does ARM's cache coherence (*e.g.*, point of coherency/unification [10], inclusiveness of last-level caches) affect FLUSH-RELOAD attacks? Unfortunately, none of these questions has been answered in prior research. Many of these questions (*e.g.*, the first one) are even considered the fundamental obstacles for conducting such attacks on ARM[1] [50].

Our exploration of these questions is driven by Android operating systems (OS) that run on ARM processors. Android is arguably the most popular operating system in mobile devices. We aim, in exploring these research questions, to demonstrate practical cache side-channel attacks on commodity Android-based smartphones from zero-permission Android apps. However, we stress the results might be ex-

CCS'16, October 24 - 28, 2016, Vienna, Austria

ISBN 978-1-4503-4139-4/16/10...$15.00

DOI: http://dx.doi.org/10.1145/2976749.2978360

[1] It was asserted that "ARM architecture does not allow user process to selectively evict memory lines and the FLUSH-RELOAD is not applicable in this architecture" [50].

tended to iOS and other computing environments should they be powered by ARM processors.

More specially, in this paper, we show a cross-core FLUSH-RELOAD side-channel attack on ARM that operates in ways that are similar to *return-oriented programming* (ROP) attacks. The first notable novelty in this attack is the use of a cache-flush interface that is available on ARM-based operating systems. This interface is designed to support self-modifying code (*e.g.*, Just-in-Time compilation) due to ARM's lack of coherence between data caches and instruction caches—instruction caches must be flushed explicitly to reflect changes made in data caches. The exact implementation of this interface, however, is processor-specific (see Sec. 3.2). Particularly, on our testbed, a Samsung Galaxy S6 smartphone, we are constrained to conduct FLUSH-RELOAD attacks using instruction FLUSHes and RELOADs, which in contrast to previously shown FLUSH-RELOAD attacks on data caches, is a brand new attempt. However, efficient exploitation of instruction RELOADs is non-trivial. We show by cleverly leveraging *gadgets* in shared libraries, an adversary may redirect the control flow of cache RELOADs to instructions in the libraries and *return* back from the gadgets immediately after loading the cache lines into the shared last-level cache (LLC). This type of instruction-based RELOADs effectively and efficiently replace the conventional data-based RELOADs. We call our new construction of FLUSH-RELOAD attacks on ARM the *return-oriented* FLUSH-RELOAD *attacks*.

We further demonstrate two categories of attacks on Android enabled by our presented FLUSH-RELOAD side channels: detecting hardware events and tracing software execution paths. We particularly show our attacks can detect the occurrence of touchscreen interrupts with high fidelity, therefore enabling the unlock pattern inference attack shown in [21] even without `procfs`; detect the use of hardware components, *e.g.*, scanning credit cards using the camera from an Uber app, thus facilitating other attack goals (such as those in [19, 31]); detect updates in the frame buffer of the smartphone display, hence monitoring the user's private actions on the device. We argue the applicability of the attacks is beyond these examples, and we leave a thorough exploration of attack spaces as future work.

Contributions. To summarize, we make the following contributions in this paper.

- A systematic exploration of vectors for FLUSH-RELOAD side-channel attacks on ARM in two aspects: cache flush operations and last-level cache inclusiveness. Particularly, we study the effects of the `clearcache` system call on the caches of five different ARM processors. We also design novel approaches to programmatically determine the inclusiveness of ARM's last-level caches.
- A novel construction of return-oriented FLUSH-RELOAD cache side-channel attacks on ARM processors that work on last-level caches. To our knowledge, our paper presents the first attempt to implement FLUSH-RELOAD side channels on ARM. Conducting these attacks in return-oriented manners by exploiting *gadgets* in shared libraries is also innovative.
- A demonstration of the presented FLUSH-RELOAD attacks in Android. We show these cache-based side channels have similar power as many `procfs`-based side channels, and therefore opening new, and hard to mitigate, attack vectors once `procfs` side channels are eliminated. We also show novel UI tracing attacks to illustrate the new capabilities of our FLUSH-RELOAD attacks compared to existing side-channel attacks in Android.

Roadmap. In the rest of the paper, we first provide the background knowledge of ARM processors and cache side-channel attacks in Sec. 2. A systematic exploration of ARM's cache flush operations and cache coherence implementation is presented in Sec. 3. We then elaborate our construction of the return-oriented FLUSH-RELOAD side channels on ARM processors in Sec. 4. Next, we demonstrate two categories of security attacks on Android that are enabled by our side channels in Sec. 5. Countermeasures to our attacks are discussed in Sec. 6 and related work in Sec. 7. Finally, we conclude the paper in Sec. 8.

2. BACKGROUND

2.1 ARM Cache-Memory Hierarchy

Similar to x86 processors, ARM processors also adopt a *modified Harvard architecture*, in which the upper-level caches (*e.g.*, L1) are split into instruction caches and data caches so that the processors can access the data bus and instruction bus simultaneously, while the lower-level caches (*e.g.*, L2) and the main memory are unified so instructions can be manipulated as data.

Supporting self-modifying code. One difference between ARM and x86 processors is that ARM does not maintain coherence between the main memory and instruction caches [10]. As such, memory writes to the code sections will not be automatically reflected in the instruction cache, causing the processors to execute staled code. This design feature affects the processors' capability to execute *self-modifying code*, which is common in Just-in-Time compilation. Accordingly, operating systems, *e.g.*, Android OS, provide a system call (*i.e.*, `clearcache`) to flush a range of virtual addresses out of the caches. This system call in implemented in the kernel by instructing the cp15 coprocessor.

Inclusive vs. exclusive LLCs. An inclusive LLC, in the case of most ARM processors—the L2 cache, guarantees that every cache line in the L1 cache also has a copy in the L2 cache. In contrast, if the L2 cache is exclusive to L1, only one copy of the same memory block is stored in either the L1 cache or the L2 cache. A third option is usually called non-inclusive cache [29], which behaves in between of the other two—a cache line evicted out of the L2 cache is not also evicted from the L1 cache. Processors may implement different LLC inclusiveness. For example, older Intel processors (*e.g.*, Core 2 processors) have non-inclusive L2 caches; recent Intel processors all come with inclusive L3 caches; in contrast, AMD processors usually have exclusive LLCs [27]. ARM's L2 caches can be configured to be inclusive, exclusive or non-inclusive to L1 instruction or data caches.

2.2 Cache Side-Channel Attacks

Sensitive information of a software program can be leaked through CPU caches. Because the cache data cannot be read by the adversary directly, such leakage is usually indirect, through "side" information. Therefore, this type of attacks is called cache side-channel attacks. Prior studies have explored three types of cache side channels: time-driven, access-driven and trace-driven. They differ in their threat models. Time-driven attacks assume only the overall execution time of certain operation is observable by the adversary;

trace-driven attacks assume the adversary is able to observe the power consumption traces of the execution; and access-driven attacks assume the adversary has logical access to a cache shared with the victim and infers the victim program's execution through its own use of the shared cache.

In this paper, we study access-driven cache side-channel attacks on ARM. The other two types are less practical in either their threat models (*e.g.*, knowledge of power consumption in trace-driven attacks) or their unrealistic assumptions (*e.g.*, assumptions of noise-free network communication in time-driven attacks). Access-driven attacks can be performed in several ways. Here, we highlight two approaches that are widely studied in recent years: PRIME-PROBE [37] and FLUSH-RELOAD [23]. We omit variations of these attacks, such as FLUSH-FLUSH and EVICT-RELOAD.

PRIME-PROBE attacks work on cache sets. By pre-loading every cache line in the target cache set with his own memory blocks, the adversary makes sure his future memory accesses (to these blocks) will be served by the cache, unless some of the cache lines are evicted by the victim program during its execution. Therefore, his own cache misses will reveal the victim's cache usage in the target cache set. In FLUSH-RELOAD attacks, the adversary shares some physical memory pages (*e.g.*, through dynamic shared libraries) with the victim. By issuing cache flush instructions (*e.g.*, `clflush` on x86) on certain virtual address range (mapped to the shared pages), the adversary can flush the (physical) cache lines that correspond to this address range out of the entire cache hierarchy. Therefore future reading (*i.e.*, RELOAD) of the cache lines will be slower because they are loaded from the memory, unless they have been accessed by the victim (and thus have been fetched into the shared cache).

2.3 Threat Model

We assume the adversary is a regular Android app with no additional permission than the default settings. Moreover, we do not assume the device itself is rooted to facilitate the attack (*e.g.*, through kernel extensions). To differentiate our attacks from prior work on `procfs`-based side channels (see Sec. 7), we do not require this third-party app to have access to `procfs`. As such, our attack will work even when these side channels are eliminated. The only assumption we need to make is that the malicious app is packaged together with a native component that is compiled with Android NDK. This configuration is very common on Android app markets. According to a recent study published in 2016, at least 37% Android apps execute native code [14].

3. DISSECTING ARM CACHES

Unlike their counterparts in x86 processors, caches in ARM processors are much less understood in the context of side-channel attacks. In order to exploit ARM caches for FLUSH-RELOAD side-channel attacks, we need to understand how ARM caches operate in both FLUSH and RELOAD operations. More particularly, we aim to explore the follow aspects:

- *Cache flush interfaces.* The cache flush interfaces on x86 is well documented: the entire cache hierarchy can be invalidated together using the privileged `WBINVD` instruction, and individual cache lines can be flushed using the unprivileged `clflush` instruction. In contrast, no userspace-accessible cache flush instruction is available on ARM. We will study a less-known attack vector—`clearcache` system call—on Android OS, and determine its impact on all levels of caches.
- *Cache inclusiveness.* The cache coherence design, particularly inclusiveness of the last-level cache to upper-level caches, is crucial to cross-core FLUSH-RELOAD attacks: Whether the victim's memory access on a different CPU core will affect the adversary's RELOADs. However, such information is seldom mentioned in ARM specification or manufacturers' documentation[2]. We aim to design novel methods to empirically determine cache inclusiveness from Android apps.

In this section, we empirically evaluate these cache properties on three most popular Android smartphones, *i.e.*, Samsung Galaxy S5 and S6 and Google Nexus 6, and the five processors they are equipped with. The Samsung Galaxy S5 implements an octa-core architecture—one quad-core 1.9 GHz Cortex-A15 CPU and one quad-core 1.3 GHz Cortex-A7 CPU—on the Exynos 5422 system-on-chip (SoC). The Samsung Galaxy S6 is equipped with one quad-core 1.5 GHz Cortex-A53 CPU and one quad-core 2.1 GHz Cortex-A57 CPU, on the Exynos Octa 7420 SoC. Google Nexus 6 comes with single quad-core Krait 450 processors on the Snapdragon 805 SoC. Cortex-A53 and Cortex-A57 are based on 64-bit ARMv8, while other CPUs are 32-bit ARMv7-based.

Roadmap of the section. In Sec. 3.1, we study the latency of several available clocks on Android to perform our timing channel attacks. In Sec. 3.2, we explore the effects of `clearcache` system call and in Sec. 3.3, we empirically determine the LLC (*i.e.*, L2) inclusiveness to both L1 caches on these processors. We discuss our findings in Sec. 3.4.

3.1 Time Measurement Facilities

Android apps do not enjoy the unprivileged `rdtsc` instruction available to their x86 counterparts. However, accurate and low-latency time measurement is critical to cache timing attacks on ARM. In this section, we examine the latency of the POSIX `clock_gettime` system call, which is a fine-grained time measurement facility that is accessible in all Android versions.

We considered using `clock_gettime()` with three clocks: CLOCK_REALTIME, CLOCK_MONOTONIC, and the per-thread clock CLOCK_THREAD_CPUTIME_ID. To select one clock with minimum latency, we conducted the following experiments: In an Android app compiled with NDK on Samsung Galaxy S6, we used one of the three clocks to measure the execution time of a loop that is expected to consume (roughly) constant time. We show in Fig. 1 the measurements of running $1, 2, \cdots, 10$ of such loops using the three clocks, respectively. The results shown are mean values and one standard deviation of 20 runs. The latency of each clock can be roughly estimated by the time measurements of executing i loops subtracting the estimated execution time of i loops. From the figure, we can see that CLOCK_REALTIME and CLOCK_MONOTONIC performs much better in terms of measurement latency (*i.e.*, roughly 130 ns) than CLOCK_THREAD_CPUTIME_ID (*i.e.*, about 780 ns). In our paper, the monotonic clock was chosen because the other may be unexpectedly adjusted by Network Time Protocol (NTP) daemon.

[2]Prior knowledge on cache inclusiveness of a particular processor implementation is only available through anecdotes [42].

3.2 Cache Flushes

We empirically study how the `clearcache` system call can be used in FLUSH-RELOAD attacks. We focused, however, only on its effects on instruction caches, because the kernel source code that implements the `clearcache` system call only *cleaned* the data caches but *invalidated* the instruction caches[3]. Cache *cleaning* means writing `dirty` cache lines out to the next level of cache/memory hierarchy and then clear the `dirty` bits; cache *invalidation* means clearing the `valid` bits of the cache lines [2]. Therefore, it is only necessary to conduct experiments to understand the behavior of `clearcache` on the instruction cache—will the L1 instruction cache and the L2 cache both be flushed?

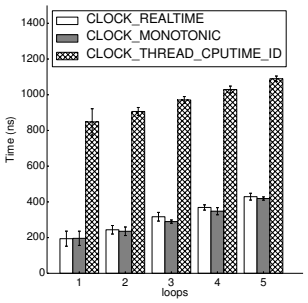


Figure 1: Time measurement using three different clocks (evaluated on A53 running at highest frequency).

We developed an Android app with Android NDK and a native shared library that exports a dummy library function (consisting of "mov x0, x0" in the 64-bit version, and "mov r0,r0" in the 32-bit version), which, after compilation, occupies exactly 1KB of memory space. To eliminate unexpected side-effects, we intentionally aligned the offset of the beginning of the function to be a multiple of 4KB within the shared library. Because of the coarse-grained address space layout randomization (ASLR) in Linux, when the library is loaded at runtime, the function will still be page-aligned. The Android app dynamically linked this self-developed library into their address space using `dlopen` at runtime. Then it split into two threads, which used `sched_setaffinity` system call to pin themselves on two different cores sharing the same L2 cache. In the following tests, all experiments were repeated 1000 times (run at the maximum CPU frequency) to measure the mean and standard deviation.

In the first experiment, thread A repeatedly executed the function code while thread B stayed idle. The average time to execute the entire function in this way was measured as T_1. Essentially, T_1 measures the time to execute the function from the L1 instruction cache (*i.e.*, all L1 cache hits).

In the second experiment, while keeping thread B idle, thread A called the `clearcache` system call before starting to execute the function. The time of the function execution itself was measured as T_2. Hence T_2 represents the effects of the `clearcache` system call on the local instruction cache and the unified L2 cache.

In the third experiment, while thread A executed the function in the same way as the first experiment, thread B repeatedly called the `clearcache` system call without any interval to flush the entire function. The execution time of the function by thread A is denoted as T_3. Therefore, T_3 reflects the effects of cross-core instruction cache flushing.

In the last experiment, we still kept thread B idle. Thread A measured the time taken (T_4) to execute the function code with L2 cache misses. In order to achieve this effect, thread A cleansed the entire L1 instruction cache and unified L2 cache in between of two function executions. The method to do so with guarantees to cleanse the entire L1 and L2 caches, however, is not straightforward. We developed our own method as described in Appendix B. The results for running the experiments on all three smartphones (five CPUs) are shown in Fig. 2. We will discuss these results shortly in Sec. 3.4.

[3] `clearcache` is implemented in the `__flush_cache_user_range` function in `mm/cache.S` of the Android's kernel source code (v3.10.9 on Galaxy S5 and Nexus 6, v3.10.61 on Galaxy S6)

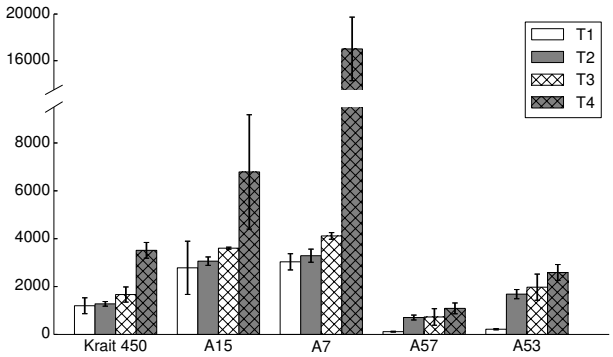


Figure 2: Effects of `clearcache` on instruction caches.

3.3 Cache Inclusiveness

We design a method using only cache timing to determine whether the L2 cache is inclusive, exclusive or non-inclusive to L1 data cache and L1 instruction cache, respectively. To do so, we first developed a shared native library which exports a dummy function (*e.g.*, 1KB) in exactly the same way as in Sec. 3.2. Then in the native component of an Android app, the following test was conducted:

Detecting inclusive L2 caches. In the first experiment, the function was loaded into the L1 data cache by reading each cache line. The average time needed to load the entire function once is denoted T_1. Then in the second experiment, the Android app completely cleansed the L2 cache without polluting the L1 data cache—by executing instructions as described in Appendix B—in between of two function code readings. The time to read the function was measured as T_2. If $T_1 \ll T_2$, T_2 reflects L1 data (and also L2) cache misses. Therefore cleansing L2 cache from instruction cache also cleanses the L1 data cache, and therefore the L2 cache is inclusive to the L1 data cache. Otherwise it is either exclusive or non-inclusive to the L1 data cache.

Because the same L2 cache may have different inclusiveness to the L1 data cache and the instruction cache, we have to conducted a similar test for L1 instruction cache. Specially, the dummy function was executed and the time to complete one execution was measured as T_1. Then in the second experiment, the L2 cache was cleansed completely from the data-cache side so that the instruction cache was not polluted (again, using method described in Appendix B) in between of two function execution. The execution time was measured as T_2. If $T_1 \ll T_2$, T_2 represents L1 instruction (and L2) cache misses. Hence, cleansing L2 cache from the data cache also cleanses the L1 instruction cache, and therefore the L2 cache is inclusive to the L1 instruction cache. Otherwise it is either exclusive or non-inclusive to the L1 instruction cache.

Smartphone	T_1	T_2	inclusiveness
Krait 450 (dcache)	1169	3700	inclusive
Krait 450 (icache)	1020	4350	inclusive
Cortex-A15 (dcache)	2600	6469	inclusive
Cortex-A15 (icache)	2484	5474	inclusive
Cortex-A7 (dcache)	3378	15460	inclusive
Cortex-A7 (icache)	3551	15822	inclusive
Cortex-A57 (dcache)	223	907	inclusive
Cortex-A57 (icache)	150	794	inclusive
Cortex-A53 (dcache)	325	1633	inclusive
Cortex-A53 (icache)	275	1287	inclusive

Table 1: L2 cache inclusiveness tests.

From Table 1 we can clearly see that on all the tested processors the L2 caches are inclusive to both data caches and instruction caches. Therefore, there is no need to conduct further experiments to distinguish exclusive and non-inclusive L2 caches. However, we describe an algorithm in Appendix C with which these two types of cache implementations can be programmatically differentiated. We hope it can be helpful to other research on similar topics.

3.4 Discussion

Cache flushes. In order to perform FLUSH-RELOAD side-channel attacks on the L2 cache, the flush operations must evict the targeted memory block out of (1) the local L1 caches, (2) the shared L2 cache, (3) and the L1 caches of other cores. If condition 1 is not met, RELOAD will only observe L1 hits; if condition 2 is not met, RELOAD will only observe L2 hits; if condition 3 is not met, the victim can continue using its local copy, so its operation will not make any changes to the shared L2 cache. We already know data caches cannot be used FLUSH-RELOAD attacks because `clearcache` only *cleans* but not *flushes* the L1 data cache—condition 1 not met. Moreover, as is seen from Fig. 2, not all instruction caches on these ARM processors satisfy these requirements, either: T_2 (local `clearcache`) and T_3 (cross-core `clearcache`) of Krait 450, Cortex A15 and Cortex A7 are merely larger than T_1 (L1 hits) and are much smaller than T_4 (L2 misses). Therefore, `clearcache` does not flush the L2 caches on these processors (condition 2 not satisfied). Cache flush operations on the instruction cache of Cortex A53 and A57 meet all three requirements: T_2 and T_3, though slightly less than T_4, are significantly greater than T_1—both the local L1 instruction cache and the L1 instruction cache on other cores, and the shared L2 cache are flushed by the cache invalidation operation.

The difference of cache flush implementation can be explained by the different implementation of point of coherency (PoC) and point of unification (PoU) on ARM [10]. PoC specifies the point at which all CPU cores are guaranteed to observe the same copy of a memory block; PoU specifies the point at which the data cache and the instruction cache on the same core are guaranteed to see the same copy of a memory block. However, ARM does not explicitly specify the choice of PoU and PoC, leaving them highly implementation dependent. Our conjecture, therefore, is that on A53 and A57 the PoU is implemented to be the memory, while on other processors the PoU is specified as the L2 cache.

Inclusiveness. Table 1 suggests all the L2 caches we evaluated are inclusive to both L1 data and instruction caches. This is in line with the limited information available from ARM official documentations: According to ARM specifications, Cortex-A57 and Cortex-A15 implement strict inclusion property with L1 data caches [5, 6], but in all other cases, these properties are not specified and therefore are implementation dependent.

Conclusion. Because the `clearcache` system call on Cortex A57 and A53 processors will flush instructions to the main memory, and at the same time the L2 caches on these processors are inclusive to the instruction cache, these two ARMv8 processors satisfy all requirements for conducting FLUSH-RELOAD attacks on shared instruction pages. As they represent the latest processor generations on the market, we anticipate future processors may have similar features. In this paper, we demonstrate FLUSH-RELOAD attacks on the instruction side of Samsung Galaxy S6.

4. RETURN-ORIENTED FLUSH-RELOAD ATTACKS ON ARM

In Sec. 3, we have shown that on ARM Cortex A57 and A53 processors we are constrained to use only instruction caches in FLUSH and RELOAD operations. Hence in this section, we first outline a basic construction of a FLUSH-RELOAD side channel on these processors by FLUSHing and RELOADing instructions (Sec. 4.1). Then we detail our novel design of return-oriented FLUSH-RELOAD side-channel attacks (Sec. 4.2). We next empirically characterize the presented return-oriented side channels (Sec. 4.3) and discuss practical considerations of exploiting such side channels on Android (Sec. 4.4).

4.1 Basic Flush-Reload Side-Channel Attacks

We first describe a basic construction of a FLUSH-RELOAD side-channel attack using the `clearcache` system call on Android. The side channel works on shared LLCs (*i.e.*, L2 caches). Therefore it can be exploited by an Android app to attack another running on a different CPU core.

The attacker Android app from which side-channel attacks are conducted is a *zero-permission* Android app packaged together with a native library. The Java component of the app interacts with the native C code through standard Java Native Interface (JNI). To enable physical memory sharing between the attacker and victim apps, the native code uses the `dlopen` system call to dynamically link a certain shared library (*i.e.*, `so` file) used by the victim app into the attacker app's own address space. When the attack starts, the service component inside the attacker app creates a new thread, which calls into its native component to conduct FLUSH-RELOAD operations in the background:

- FLUSH*:* The attacker app calls `clearcache` to flush a function in the code section of this *shared* library.
- FLUSH-RELOAD *interval:* The attacker app waits for a fixed time period (may be zero), during which the victim app may execute the function.
- RELOAD*:* The attacker app executes the function and measures the time of execution. Shorter execution time indicates the function has been executed (thus fetched into the L2 cache) by some other apps (possibly the victim app) during the FLUSH-RELOAD interval.

The primary difference between our work and previous study [16, 49, 50, 53] is that we exploit the instruction cache, while prior studies use the data cache. Nevertheless, the seemingly minor distinction imposes considerable technical challenges to our attack. *First*, to call library functions, the

attacker app needs to re-construct the program semantics (*e.g.*, preparing parameters, global variables, *etc.*) before calling, which is very tedious and does not work for some functions. *Second*, the execution time of a function may vary from one run to another, which makes differentiating cache misses and cache hits very challenging in the RELOAD phase. This is typically true if the function call also involves system calls. *Third*, FLUSH and RELOAD take too much time; many fast victim operations will be missed by such *slow* FLUSH-RELOAD attacks. To address these challenges, we next design a return-oriented FLUSH-RELOAD attack.

4.2 Return-Oriented Reloads

Instead of calling the entire function in the RELOAD phase, we touch upon selected memory blocks of the function code in a return-oriented manner. Particularly, much similar to control-flow hijacking attacks using return-oriented programming [18, 41], a number of small gadgets are collected from the target function, and then in our attacker app, an auxiliary function will be constructed to jump to these gadgets (and then jump back) one after another. The overall execution time of these gadgets will be measured as the outcome of the RELOAD phase. It is important to avoid having more than one gadgets in the same 64-byte cache line, because only the execution of the first gadget will fetch the memory block into the cache, and subsequent gadgets in the same cache line merely introduce noise.

The set of ARM instructions that can lead to indirect control flow transfers are listed in Table 2. On 32-bit ARM v7 architectures, `bx Rm` sets the current `PC` value to be the value of the register `Rm` (*i.e.*, indirect jump); `blx Rm` also sets the value of `LR` (*i.e.*, `R14`) to be the address of the next instruction before jumping to the address specified by `Rm` (*i.e.*, indirect call). Direct manipulation of `PC` is also allowed by using `mov` or `pop` or `ldm` instructions[4]. On 64-bit ARM v8 architectures, in addition to the difference in the size of the registers (64 bits), `ret` instructions are also available: `ret` instruction changes the `PC` value to the value of `LR`; `ret Xm` sets `PC` to the value stored in `Xm` rather than `LR`. However, the `PC` register can no longer be manipulated directly on ARM v8 architectures.

Architecture	Instructions	Effects
ARM v7 (32 bit)	`bx LR`	`PC := LR`
	`bx Rm`	`PC := Rm`
	`blx Rm`	`LR(R14) :=` next instr. `PC := Rm`
	`mov PC LR`	`PC := LR`
	`pop {pc}`	`PC :=` top of stack
	`ldm {pc}`	load multiple regs
ARM v8 (64 bit)	`br Xm`	`PC := Xm`
	`blr Xm`	`LR(X30) :=` next instr. `PC := Xm`
	`ret Xm`	`PC := Xm`
	`ret`	`PC := LR(X30)`

Table 2: Indirect control-flow transfer instructions on ARM.

The gadgets used in our attack will be easier to construct compared to those in ROP attacks—our gadgets do not need to complete any meaningful operations. We only need to jump to one of these indirect control transfer instructions (listed in Table 2) so that the memory block that holds the instruction is loaded into the cache. The control flow will be immediately transferred back to the auxiliary function after the instruction is fetched and executed.

We illustrate the use of the 64-bit `ret` and `blr Xm` instructions to construct gadgets in Fig. 3. In particular, in this example, the adversary hopes to FLUSH-RELOAD a function, `clock_gettime`, in `libc.so`. To exploit the `blr X4` instruction as a gadget, the attacker app calculates the virtual address of the instruction at runtime (*i.e.*, the base address of the library's code section plus the offset of the instruction within the library). Here, let's assume the virtual address is `0x246a0`. This address is first loaded into register `X19`, so that the control flow will be transferred to the gadget later by `br X19`. The adversary then makes a copy of `X30` to another register, say `X20`, because it will be modified by the gadget instruction `blr X4`. Then the adversary prepares the value of `X4`, the target address of `blr X4`, so that the control flow will be redirected back to the auxiliary function once the gadget is executed. It is important to maintain the correctness of the subsequent execution by restoring the value of `X30` from register `X20`.

Exploiting `ret` is much easier. The adversary first stores the address of the gadget in a register, say `X19`. Then the control flow is transferred by `blr X19`, which sets the value of `X30` to be the address of the next instruction and then changes the PC value to the address stored in `X19`. The control flow will be transferred back to the address stored in `X30` by the `ret` instruction.

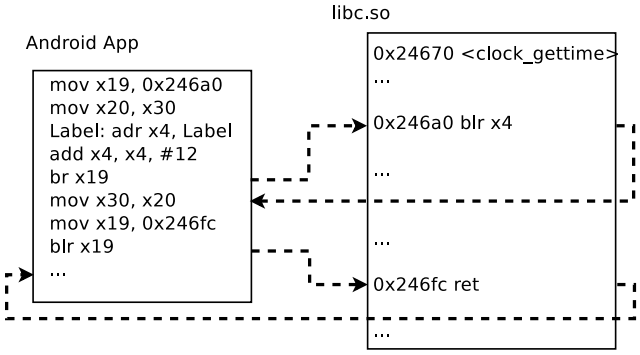


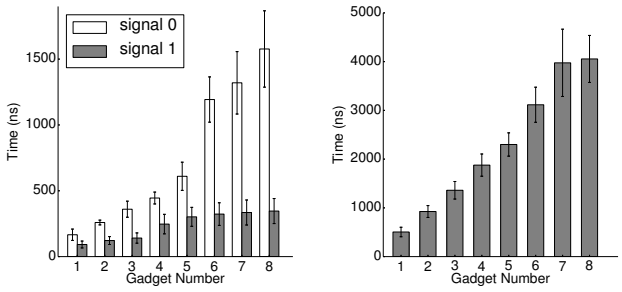
Figure 3: Examples of gadgets.

Availability of the reload gadgets. To investigate the availability of the reload gadgets in Android shared libraries, we used `objdump` to disassemble five widely-used shared libraries used on a 64-bit Android OS (Samsung Galaxy S6, Android version 5.1.1). We then wrote a Python script to count the number of indirect control transfer instructions and the number of useful gadgets (those in separate cache lines) in these libraries. The results are listed in Table 3.

Libraries	code size	branch instr.	gadgets
libc.so	912 KB	2755	1547
libc++.so	1050 KB	3714	2174
libinput.so	186 KB	585	283
libcrypto.so	2065 KB	6897	4246
libandroid.so	92 KB	430	180

Table 3: Availability of the reload gadgets.

[4]There are four variations of `ldm`: `ldmia`, `ldmib`, `ldmda`, `ldmdb`. Their usage can be found in [3].



(a) Minimum gadgets. (b) Min. FLUSH-RELOAD cycles.
Figure 4: Characteristics of FLUSH-RELOAD side channels.

4.3 Characterizing ARM Flush-Reload Attacks

In this section, we evaluate two important characteristics of the return-oriented FLUSH-RELOAD side-channel attacks described above: (1) the minimum number of gadgets needed to FLUSH-RELOAD at the same time in order to reliably differentiate cache hits from cache misses, and (2) the shortest FLUSH-RELOAD cycles (*i.e.*, time to finish one round of FLUSH and RELOAD with zero FLUSH-RELOAD interval) for one gadget. The experiments were run on A53 with CPU frequency set as 1.5GHz.

Minimum gadgets for successful attacks. In a successful FLUSH-RELOAD side-channel attack, the adversary exfiltrates one bit information pertaining to a target system-level event—happen or not happen—during a certain time period. Such information is learned by determining if RELOADs lead to cache hits or cache misses. To reliably detect the occurrence of the events, the adversary might need to FLUSH-RELOAD more than one gadgets from the same function (*e.g.*, Fig. 3), or from different functions that will be called sequentially during the same event.

We evaluate the minimum number of gadgets that the adversary needs in FLUSH-RELOAD attacks by testing the strength of the signal of a FLUSH-RELOAD *covert channel*. Specially, we chose 10 functions from the `libc.so`[5], and constructed one gadget from each function, by using the last `ret` instruction. Then two Android apps were developed: The receiver repeatedly FLUSHed k gadgets one after another (k ranges from 1 to 8), then after zero FLUSH-RELOAD interval he RELOADed the gadgets in the same order; the sender sent '0' by running in an empty loop, and sent '1' by calling the corresponding functions repeatedly. We want to find out the minimum number of gadgets that allows the adversary to differentiate the signal '0' from '1'. The results for running the experiment 100 times (with mean and one standard deviation) are shown in Fig. 4a. The white bars show the RELOAD time when the sender sent '0' and the solid bars show the RELOAD time when the sender sent '1'. We can see the signal is clear even when only one gadget is used, and becomes more reliable when more gadgets are used.

Shortest Flush-Reload cycles. The granularity of the side channel is characterized by the shortest FLUSH-RELOAD cycles—the time to finish one round of FLUSH and RELOAD with zero FLUSH-RELOAD interval. The shortest cycle indicates the highest frequency with which the FLUSH-RELOAD attacks can be performed. To empirically determine this property, we executed the same receiver app developed in the paragraph above. It FLUSH-RELOADs on k gadgets (k ranges from 1 to 10) without FLUSH-RELOAD intervals. The mean execution time of one FLUSH-RELOAD cycle and one standard deviation are reported in Fig. 4b. We can see that FLUSH-RELOAD cycles for one gadget on A53 is around 500ns, and that for more gadgets increases (roughly) linearly.

4.4 Practical Considerations

To make these attacks practical on Android, however, we consider several factors that may constrain the attacks.

CPU frequency. Mobile devices dynamically scale up and down the operation frequency of each CPU core independently to reduce power consumption. The timing channel, when executed on CPUs with varying frequency, may lead to unstable results. In the practical attacks that we will demonstrate in Sec. 5, fixing CPU frequency from the kernel is not an option. Therefore, we evaluated to what extent will the frequency vary during our FLUSH-RELOAD attacks. To do so, we conducted an experiment in which we measured the frequency of the CPU core on which the FLUSH-RELOAD attack runs. This can be done by reading `sysfs`[6] within the attacker app itself, because no additional permission is required. We found in all cases after starting running our FLUSH-RELOAD attacks from an otherwise idle CPU core, the operation frequency would reach the maximum and stay unchanged until the attacks finished. Therefore, CPU frequency scaling will not impact our attack once the malicious app warms up the CPU.

Thread scheduling and cgroups. To limit the resource consumption of background threads, Android employs two control groups (cgroups). The background apps and threads are assigned to the *background* cgroups where up to 5% CPU resources can be used when contending with other apps. However, we found in our experiments that the attacks are not affected by such mechanisms as long as the device is not running computation-intensive apps that occupy all CPU resources, in which case the CPU caches will be highly polluted and cache attacks will hardly work anyway.

Dual CPU architectures. More recent smartphones (*e.g.*, Samsung Galaxy S5 and S6) come with octa-core processors—two processor packages with four cores each. For example, recent Samsung Exynos processors usually have two asymmetric processors, one with higher operation frequency to support CPU-intensive applications and one with lower frequency to save power when the demand is low. However, in our experiments, although the malicious app and the victim app run on different CPUs, the return-oriented FLUSH-RELOAD attacks can still successfully differentiate whether the victim touched the shared library functions or not (though the differences are slightly smaller). We presume this is because of the ARM Cache Coherent Interconnect [4]. Similar cross-core FLUSH-RELOAD side channels have been observed by Irazoqui *et al.* on AMD processors [27]. Therefore, we only run the malicious app on a core of Cortex-A53 processor (using `sched_setaffinity` system call that requires no additional permission) and the victim app running on either of the CPUs can be targeted.

[5]The 10 functions are: `atoi`, `fflush`, `free`, `getgid`, `getuid`, `isdigit`, `isspace`, `malloc`, `strnlen`, `strtof`.

[6]`/sys/devices/system/cpu/cpu<n>/cpufreq/scaling_cur_freq`.

64-bit vs. 32-bit devices and libraries. Our return-oriented FLUSH-RELOAD attacks work differently on 64-bit apps and 32-bit apps. On the 64-bit Samsung Galaxy S6, both 64-bit and 32-bit apps can be executed. However, if the attacker app is compiled as a 64-bit app, it cannot conduct a return-oriented attacks on a shared 32-bit library. Similarly, a 32-bit attacker app cannot exploit a shared 64-bit library, either. Therefore, two versions of the malicious app were developed to attack both types.

Background noise. Similar to prior work on cache side-channel attacks, our return-oriented FLUSH-RELOAD side-channel attacks are also subject to background noise. The most notable noise comes from a third app that shares the same library and calls the functions that are being RELOADed by the attacker app. To address the problem, we FLUSH-RELOAD multiple gadgets from different rarely-used functions at the same time, so that the likelihood of these functions being called together by another app is very low. Hence, most background noise of this kind can be eliminated.

Power consumption. When the attacker app runs in the background, one entire CPU core (*i.e.*, 12% of all CPU resources) is taken, and 1.5% battery was consumed every 20 minutes. The power consumption is probably on par with (or slightly lower than) that reported in Diao *et al.* [21]. Because FLUSH-RELOAD attacks do not need to evict an entire cache set (as is the case in PRIME-PROBE or EVICT-RELOAD attacks), we suspect our techniques may consume less power than ARMageddon [32].

Vulnerability analysis. The following steps can be taken to analyze the vulnerabilities of an app in an *offline* procedure: First, all libraries linked in an app can be learned from `/proc/<pid>/maps` at runtime. We could extract the symbols for all exported functions of the library of interest using `objdump`. Then, using a Python script, we can generate a `gdb` initialization file, which contains breakpoint information of all (or a subset) of the functions in the `objdump` result. Next, by employing `gdb` debugger on Android [8], we attach to the victim app remotely and insert all the breakpoints by loading the initialization file. After that, we manually act on the app and see if any breakpoint is triggered. This will provide a coarse-grained call graphs for identifying the most critical execution path of the program. Of course, this approach is manual-intensive and error-prone. We leave a fully automated analysis for future work.

5. CASE STUDIES ON ANDROID

In this section, we demonstrate a few real-world examples to illustrate how the return-oriented FLUSH-RELOAD side-channel attacks can be applied in practice. Specially, we show two categories of attacks: detecting hardware events and tracing software execution paths. The attacks were all demonstrated on Samsung Galaxy S6 (SM-G920F), with Android version 5.1.1 and Linux kernel version 3.10.

5.1 Detecting Hardware Events

Our return-oriented FLUSH-RELOAD attacks can be exploited to detect hardware events, such as occurrences of hardware interrupts and software's interactions with hardware devices (*e.g.*, GPS, microphones, cameras, *etc.*). To demonstrate such capabilities in concrete contexts, we conduct two case studies: In the first case study, we exploit the established side channel to accurately detect hardware interrupts due to touchscreen events; in the second case study, we show how an attacker can learn from the side channel when the camera is used by the Uber app[7] to scan credit cards (using *card.io* libraries).

5.1.1 Touchscreen Interrupts

The FLUSH-RELOAD side channel does not detect interrupt directly; it only detects these events by monitoring system libraries that are triggered to dispatch these events to user-space applications. Specially, in Linux's multi-touch protocol, the user's interaction with the touchscreen generates a sequence of multi-touch transfers—each transfer may include multiple event packets if the user has multiple concurrent contacts with the device. By the end of each multi-touch transfer, a `SYN_REPORT` event is delivered to userspace software [40]. In fact, each of these multi-touch transfer correlates with one touch event [7]. It has been shown in a recent study by Diao *et al.* [21] that side channels can be established through `procfs` (*i.e.*, `/proc/interrupts`) to infer unlock pattern. Here in our paper, we show that our return-oriented FLUSH-RELOAD side-channel attacks can be used to replace this `procfs` side channel—should future Android OS restricts unauthorized usage of `procfs`, the security threats still exist.

The attack. To detect these `SYN_REPORT` events, the malicious app FLUSH-RELOADs three gadgets in three different functions (*i.e.*, `TouchInputMapper::sync`, `CursorMotionAccumulator::clearRelativeAxes`, and `MultiTouchMotionAccumulator::finishSync` in `libinputflinger.so`) that will be called together when the driver calls the `input_sync()` function to deliver `SYN_REPORT` events. We assign a fast RELOAD a value '1'; a slow one a value '0'. To reduce noise in measurements, we group every 20 consecutive data points: If there are more than 10 '1's within these 20 points, we consider it the beginning of a `SYN_REPORT` event. Similarly, within 20 consecutive data points, if all values are '0's, we consider the event has finished. In practice, these functions may sometimes take longer to finish so that two consecutive `SYN_REPORT` events cannot be separated. To address this problem, we first estimated the average interval between every two consecutive `SYN_REPORT` events between a pair of `BTN_TOUCH_DOWN` and `BTN_TOUCH_UP` events. An average value of 11.659ms was calculated from 500 pairs of consecutive `SYN_REPORT` events. Therefore we used such a value as the threshold in our detection: If the same `SYN_REPORT` event does not finish after 11.659ms, we artificially add another event at this point.

Results. We first show that our FLUSH-RELOAD-based event detection can be correlated with both `SYN_REPORT` events and counter increments in `/proc/interrupts`. To do so, we collected three sequences of events simultaneously while touching the touchscreen: `SYN_REPORT` events (through the `getevent` command in Android Debug Bridge), counter increments of `/proc/interrupts` (on Samsung Galaxy S6, the *fts_touch* device), and our aggregated return-oriented FLUSH-RELOAD detection (described in the above paragraph). These values are synchronized using timestamps, and reported in Fig. 5. It is clear the three events can be correlated with one another (with occasional mismatches). Therefore the FLUSH-RELOAD side channel can replace the `procfs` side channels.

[7] https://www.uber.com/

Unlike `procfs` side channels, however, cache side channels are subject to noise and may have both false positives and false negatives. We manually generated 10294 `SYN_REPORT` events and here report our detection accuracy in Table 4. Each column represents the maximal allowed latency: only detection within the allowed latency (detection should happen after the `SYN_REPORT` event) is counted as accurate detection. As such, a false positive (FP) is defined as an event detection reported without an corresponding `SYN_REPORT` event proceeding it; and a false negative (FN) is counted as one `SYN_REPORT` event that is not detected within the allowed latency. We see from the table when the allowed latency is small (*e.g.*, 400μs), the accuracy is low. This is because the interrupt dispatcher functions are called by the driver later than the actual events. But if the allowed latency is larger (*e.g.*, 1 or 2ms), the FP and FN rates are much lower. For attacks in [21], latency of 1 or 2ms can be tolerated because the maximum frequency of touchscreen IRQs is only 135Hz, which means touch events will be dispatched every 7ms at most; the level of inaccuracy (less than 10%) should still permit the unlock pattern inference attack in [21].

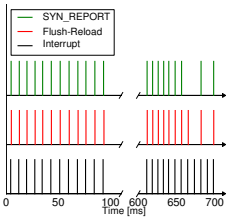


Figure 5: Correlations between events, from top to bottom: `SYN_REPORT` events, FLUSH-RELOAD, `/proc/interrupts`.

Latency (ms)	0.4	0.6	0.8	1.0	1.2	1.4	1.6	1.8	2.0
FP(%)	52.8	25.6	9.6	4.5	3.3	3.0	2.7	2.5	2.5
FN(%)	59.2	32.0	16.0	11.0	9.7	9.4	9.1	8.9	8.9

Table 4: Detection accuracy.

5.1.2 Credit Card Scanning

Some Android apps, such as Uber and PayPal, use *card.io* libraries developed by PayPal[8] to scan credit card information from their apps. Although *card.io* is a 32-bit third-party library, we can still perform FLUSH-RELOAD side-channel attacks on gadgets collected from it. We demonstrate that a malicious app can accurately detect when the user scans her credit card from the Uber app using FLUSH-RELOAD channels. Such capabilities, though not dangerous by itself, may take the places of various `procfs` side channels [19, 31] and facilitate other security attacks, such as taking pictures from the background [19] at the right moment and taking screenshots when sensitive information is displayed [31]. Gadgets were collected from `setScanCardNumberResult`, `setDetectedCardImage`, and `scanner_add_frame_with_expiry` in `libcardioRecognizer.so`. By FLUSH-RELOAD these three gadgets together, we could *reliably* detect when the user scans her credit card using the app.

5.2 Tracing Software Execution Paths

It has been shown x86 FLUSH-RELOAD attacks can be used to trace software execution paths [53] in clouds, here we demonstrate our return-oriented FLUSH-RELOAD side channels have similar power in mobile devices. Specially, we illustrate an interesting attack against *SurfaceFlinger*, dubbed the *UI tracing attacks*. SurfaceFlinger is an important Android component that accepts graphic buffers from multiple sources, composes the buffers together to resolve inconsistency, and then, upon receiving a `vsync` signal, sends the composed graphic buffer to the display (by calling `SurfaceFlinger::postFramebuffer()`) if there is an update in the buffer. We show by conducting a FLUSH-RELOAD attack on this function, the malicious app can (1) detect when a notification appears and disappears in the status bar, and (2) infer the number of characters that the user has typed in a password field.

Detecting push notifications. An Android push notification will be shown on the status bar temporarily once it is received. Detecting the occurrence of push notifications may reveal the user's private action on the smartphone. By FLUSH-RELOADing the `postFramebuffer` function, one can accurately pinpoint the time period that the notification is listed on the status bar. This capability is shown in Fig. 6a. In this figure, and also the other two figures in Fig. 6, the X-axis shows real-time FLUSH-RELOAD events (roughly 250K points per second), and the Y-axis shows the raw values of RELOADs: values higher than 240ns are considered 240ns. All data points are connected with lines. Therefore a vertical "bar" is actually several data points connected with lines. From the figure, we can see that the notification showed up at around 1s, and disappeared at 4s, which is consistent with the ground truth.

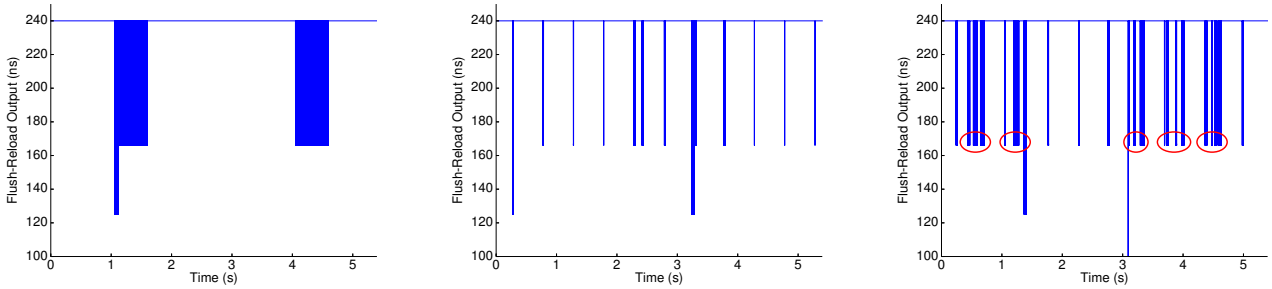
Detecting display updates. Updates to the display can be reflected by FLUSH-RELOADing the `postFramebuffer` function. For instance, On a *Discover* banking app we downloaded from the Google Play Store, a password field in the user login activity, when focused, will show a blinking cursor at the frequency of 500ms [1]. As seen in Fig. 6b, The blinking cursor can be detected with a sequence of fast RELOADs (*i.e.*, about 170ns) of the `postFramebuffer` gadget every 500ms. Moreover, whenever the user types a password, the display needs to be updated accordingly. Fig. 6c shows five abnormal display update activities, corresponding to the five characters typed in the password field. This capability can leak inter-keystroke information that may lead to password cracking.

6. COUNTERMEASURES

Disallow user-space cache flushes. By disabling the system interfaces to flush the instruction caches, the FLUSH-RELOAD side channels can be removed entirely from ARM-based devices. However, because ARM does not maintain cache coherence between data caches and instruction caches, disallowing user-space cache flushes entirely also disables self-modifying code. That means features like just-in-time compilation (*e.g.*, heavily used in Dalvik VM) will not work on Android. An alternative solution is to only disable explicit cache flush system calls, *e.g.*, `clearcache`, but allow implicit cache flushes after `mprotect`. The feasibility and security of such a design requires further investigation.

Restrict fine-grained time measurements. Removing `clock_gettime` system call and other fine-grained timers from Android will mitigate all timing side channels. How-

[8] https://www.card.io



(a) Notification detection. (b) Cursor blinking. (c) Five characters entered.

Figure 6: UI tracing attacks.

ever, doing so will make many apps that rely on accurate time measurement unusable. Moveover, we also note removing these fine-grained timers alone does not guarantee elimination of timing channels. It has been argued by Wray [48] that reference clocks can be established using other approaches, such as I/O or memory subsystems.

Prevent physical memory sharing. The return-oriented Flush-Reload attack on ARM will be completely eliminated if no memory sharing is allowed between apps. However, the expansion of the memory footprint because of this method will stress the availability of the physical memory. The *copy-on-access* mechanism proposed by Zhou *et al.* [55] appears to be the only effective and efficient countermeasure against Flush-Reload attacks. The method keeps a state machine to track the *sharing* of each physical page between security domains (*e.g.*, containers). Accessing shared page by any security domain will trigger a page copy thus preventing Flush-Reload based attacks entirely. Given the low performance overhead of the method, it is probable for future Android OS or even mainstream Linux kernels to implement such defense methods.

7. RELATED WORK

Cache side-channel attacks. Most prior studies on cache side-channel attacks focused on caches in x86 architectures, including data caches (and also per-core L2 unified caches) [23, 24, 35, 37, 38, 46], instruction caches [12, 13, 52], and inclusive LLCs [16, 22, 25, 26, 28, 33, 36, 49, 50, 53]. ARM-based cache side-channel attacks have also been studied, but most of them were in the context of time-driven attacks [43, 45, 47] (see Sec. 2). Access-driven cache side-channel attacks on ARM have only been explored by two recent studies [32, 44]. Particularly, the attacks presented by Spreitzer *et al.* [44] required root privilege and kernel modules to facilitate the attacks, which have been considered impractical. Most relevant to our work is due to Lipp *et al.* [32], who explored Evict-Reload and Prime-Probe attacks on ARM. The major advantage of the Flush-Reload attacks presented in our paper is that we do not require knowledge of virtual-to-physical address translation, which is a necessity in [32]. On Android, such knowledge can only be learned by reading `/proc/<pid>/pagemaps`, which is considered a vulnerability and has already been restricted from mainstream Linux kernels [9] and Android OS [11][9].

[9]We noticed that [32] vaguely discussed a Flush-Reload attack exploiting an inadvertently unlocked flush instruction on Galaxy S6 only a few days before the deadline for the final version. We weren't able to confirm their result in our study.

Side channels on Android. Other types of side channels have also been studied previously. These research are generally divided into two categories: `procfs`-based side channels [19, 21, 30, 31, 39, 51, 54] and sensor-based side channels [15,17,34]. Our Flush-Reload attacks, as a third type, can enhance, or be enhanced by, these side-channel attacks. For instance, Chen *et al.* [19] studied the use of `procfs` to detect Android activity transition, which can facilitate our UI tracing attacks (Sec. 5.2). Moreover, our attack can replace many `procfs`-based side channels, if access to this pseudo file system is restricted, and even achieve finer-grained observation than existing techniques, for instance, via tracing software execution paths.

8. CONCLUSION

In this paper, we successfully demonstrated the feasibility of conducting Flush-Reload side-channel attacks on ARM last-level caches. Our contributions are at least three-fold: First, we showed that Flush can be implemented on ARM by leveraging the `clearcache` system call that are available on all ARM-based operating systems (*e.g.*, Android) for maintaining coherence between the data and instruction caches. Second, we designed a novel return-oriented Reload mechanism so that code segments in shared libraries can be loaded into the instruction caches in units of gadgets, rather than functions. Third, we studied how these side channels can be exploited on Android-based mobile devices. We took into consideration practical issues such as CPU frequency scaling, thread scheduling, multi-CPU architecture, power consumption, *etc.*, and demonstrated two categories of attacks on Android: detecting hardware events and tracing software execution paths.

Acknowledgements

We would like to thank the National Science Foundation for supporting our research through grant 1566444.

9. REFERENCES

[1] Android source code. https://android.googlesource.com/platform/frameworks/base/+/master/core/java/android/widget/Editor.java. Retrieved in May 2016.

[2] ARM architecture reference manual. http://infocenter.arm.com/. ARMv8, for ARMv8-A architecture profile, beta.

[3] ARM architecture reference manual. http://infocenter.arm.com/. ARMv7, for ARMv7-A architecture profile.

[4] ARM CoreLink CCI-400 Cache Coherent Interconnect Technical Reference Manual. http://infocenter.arm.com/. Revision: r1p5.

[5] ARM Cortex-A15 MPCore Processor. http://infocenter.arm.com/. Revision: r4p0.

[6] ARM Cortex-A57 MPCore Processor. http://infocenter.arm.com/. Revision: r1p3.

[7] Event codes. https://www.kernel.org/doc/Documentation/input/event-codes.txt. event codes - The Linux Kernel Archives.

[8] GDB documentation. http://www.gnu.org/software/gdb/documentation/.

[9] pagemap: do not leak physical addresses to non-privileged userspace. https://lwn.net/Articles/642074/. Retrieved in May 2016.

[10] Programmer's Guide for ARMv8-A. http://infocenter.arm.com/. Version 1.0.

[11] Upstream: pagemap: do not leak physical addresses to non-privileged userspace. https://android-review.googlesource.com/#/c/182766/. Retrieved in Aug. 2016.

[12] O. Aciiçmez. Yet another microarchitectural attack: exploiting I-Cache. In *2007 ACM workshop on Computer security architecture*, 2007.

[13] O. Aciiçmez, B. B. Brumley, and P. Grabher. New results on instruction cache attacks. In *12th international conference on Cryptographic hardware and embedded systems*, 2010.

[14] V. Afonso, A. Bianchi, Y. Fratantonio, A. Doupe, M. Polino, P. de Geus, C. Kruegel, and G. Vigna. Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In *2016 ISOC Network and Distributed System Security Symposium*, 2016.

[15] A. J. Aviv, B. Sapp, M. Blaze, and J. M. Smith. Practicality of accelerometer side channels on smartphones. In *28th Annual Computer Security Applications Conference*, 2012.

[16] N. Benger, J. van de Pol, N. P. Smart, and Y. Yarom. "Ooh Aah... Just a Little Bit": A small amount of side channel can go a long way. In *16th International Workshop on Cryptographic Hardware and Embedded Systems*, 2014.

[17] L. Cai and H. Chen. Touchlogger: Inferring keystrokes on touch screen from smartphone motion. In *6th USENIX Conference on Hot Topics in Security*, 2011.

[18] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *17th ACM Conference on Computer and Communications Security*, 2010.

[19] Q. A. Chen, Z. Qian, and Z. M. Mao. Peeking into your app without actually seeing it: UI state inference and novel Android attacks. In *23th USENIX Security Symposium*, 2014.

[20] R. Delgado. Arm-based servers: The next evolution of the cloud? http://www.cloudcomputing-news.net/news/2015/apr/17/arm-based-servers-next-evolution-cloud/.

[21] W. Diao, X. Liu, Z. Li, and K. Zhang. No pardon for the interruption: New inference attacks on android through interrupt timing analysis. In *37th IEEE Symposium on Security and Privacy*, 2016.

[22] D. Gruss, R. Spreitzer, and S. Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security Symposium*, 2015.

[23] D. Gullasch, E. Bangerter, and S. Krenn. Cache games – bringing access-based cache attacks on AES to practice. In *32nd IEEE Symposium on Security and Privacy*, 2011.

[24] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space ASLR. In *34th IEEE Symposium on Security and Privacy*, 2013.

[25] M. S. Inci, B. Gulmezoglu, G. Irazoqui, T. Eisenbarth, and B. Sunar. Seriously, get off my cloud! cross-vm rsa key recovery in a public cloud. Cryptology ePrint Archive, Report 2015/898, 2015. http://eprint.iacr.org/.

[26] G. Irazoqui, T. Eisenbarth, and B. Sunar. S$A: A shared cache attack that works across cores and defies VM sandboxing—and its application to AES. In *36th IEEE Symposium on Security and Privacy*, 2015.

[27] G. Irazoqui, T. Eisenbarth, and B. Sunar. Cross processor cache attacks. In *11th ACM Asia Conference on Computer and Communications Security*, 2016.

[28] G. Irazoqui, M. S. Inci, T. Eisenbarth, , and B. Sunar. Wait a minute! a fast, cross-vm attack on AES. In *17th International Symposium Research in Attacks, Intrusions and Defenses*, 2014.

[29] A. Jaleel, E. Borch, M. Bhandaru, S. C. Steely Jr., and J. Emer. Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (tla) cache management policies. In *43rd Annual IEEE/ACM International Symposium on Microarchitecture*.

[30] S. Jana and V. Shmatikov. Memento: Learning secrets from process footprints. In *33rd IEEE Symposium on Security and Privacy*, 2012.

[31] C.-C. Lin, H. Li, X. Zhou, and X. Wang. Screenmilker: How to milk your Android screen for secrets. In *21st ISOC Network and Distributed System Security Symposium*, 2014.

[32] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. ARMageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium*, 2016.

[33] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *36th IEEE Symposium on Security and Privacy*, 2015.

[34] Y. Michalevsky, D. Boneh, and G. Nakibly. Gyrophone: Recognizing speech from gyroscope signals. In *23rd USENIX Security Symposium*, 2014.

[35] M. Neve and J.-P. Seifert. Advances on access-driven cache attacks on AES. In *13th international conference on selected areas in cryptography*, 2007.

[36] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In

22nd ACM SIGSAC Conference on Computer and Communications Security, 2015.

[37] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES. In *6th Cryptographers' track at the RSA conference on Topics in Cryptology*, 2006.

[38] C. Percival. Cache missing for fun and profit. In *2005 BSDCan*, 2005.

[39] Z. Qian, Z. M. Mao, and Y. Xie. Collaborative TCP sequence number inference attack: How to crack sequence number under a second. In *19th ACM Conference on Computer and Communications Security*, 2012.

[40] H. Rydberg. Multi-touch (mt) protocol. Linux kernel documentation. https://www.kernel.org/doc/Documentation/input/multi-touch-protocol.txt.

[41] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *14th ACM Conference on Computer and Communications Security*, 2007.

[42] A. L. Shimpi. Answered by the experts: Arm's cortex a53 lead architect, peter greenhalgh. http://www.anandtech.com/show/7591/answered-by-the-experts-arms-cortex-a53-lead-architect-peter-greenhalgh.

[43] R. Spreitzer and B. Gérard. Towards more practical time-driven cache attacks. In *8th IFIP International Workshop on Information Security Theory and Practice, Securing the Internet of Things*, 2014.

[44] R. Spreitzer and T. Plos. In *4th International Workshop on Constructive Side-Channel Analysis and Secure Design*, 2013.

[45] R. Spreitzer and T. Plos. On the applicability of time-driven cache attacks on mobile devices. In *7th International Conference on Network and System Security*, 2013.

[46] E. Tromer, D. A. Osvik, and A. Shamir. Efficient cache attacks on AES, and countermeasures. *J. Cryptol.*, 23(2):37–71, Jan. 2010.

[47] M. Weiß, B. Heinz, and F. Stumpf. A cache timing attack on AES in virtualization environments. In *16th International Conference on Financial Cryptography and Data Security*, 2012.

[48] J. C. Wray. An analysis of covert timing channels. In *1991 IEEE Computer Society Symposium on Research in Security and Privacy*, 1991.

[49] Y. Yarom and N. Benger. Recovering OpenSSL ECDSA nonces using the FLUSH+RELOAD cache side-channel attack. In *Cryptology ePrint Archive*, 2014.

[50] Y. Yarom and K. E. Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium*, 2014.

[51] N. Zhang, K. Yuan, M. Naveed, X. Zhou, and X. Wang. Leave me alone: App-level protection against runtime information gathering on android. In *36th IEEE Symposium on Security and Privacy*, 2015.

[52] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM side channels and their use to extract private keys. In *19th ACM Conference on Computer and Communications Security*, 2012.

[53] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-tenant side-channel attacks in PaaS clouds. In *21st ACM Conference on Computer and Communications Security*, 2014.

[54] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt. Identity, location, disease and more: Inferring your secrets from Android public resources. In *20th ACM Conference on Computer and Communications Security*, 2013.

[55] Z. Zhou, M. K. Reiter, and Y. Zhang. A software approach to defeating side channels in last-level caches. In *23rd ACM Conference on Computer and Communications Security*, 2016.

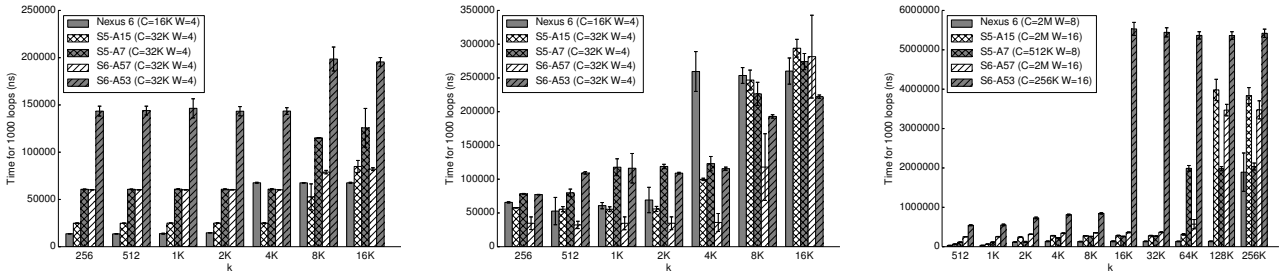
APPENDIX

A. DISSECTING CACHE DIMENSIONS

The ARM specification only specifies the size of cache lines, leaving the number of cache ways and cache sets chosen by the processor manufacturers. The ARM manufacturers, however, usually do not reveal such implementation details. Moreover, unlike their x86 counterparts, ARM does not provide unprivileged `cpuid` instructions to determine cache dimensions at runtime. We develop methods to programmatically determine the cache dimensions, which are useful information for Appendix B and Appendix C. Specially, a cache's dimension can be uniquely characterized by its cache line size L, the number of cache ways W and the number of cache sets S. The total cache size C is given by $C = L \times W \times S$.

We develop a technique to determine W and S of L1 data cache, L1 instruction cache and the unified L2 cache by only using timing information involved in memory accesses. Specially, our method is a series of hypothesis tests. The null hypotheses are $W = n$, where n is some integer. In each test for the L1 data cache and the L2 cache, we first allocate a physically consecutive memory buffer that has *twice* the size of the cache under testing, $2C$. Then we access (by loading) $2n$ memory addresses $m_1, m_2, m_3, \cdots, m_{2n}$, so that $m_i - m_{i-1} = k$, where k takes value from $\{L, 2L, 4L, \cdots, C/n, \cdots\}$. The tests for the L1 instruction cache are similar except that by loading the memory we are executing a dummy function that implements short instruction sequences starting at addresses m_i which jumps to address m_{i+1} after execution.

We measure the total execution time of loading (or executing) the set of memory 1000 times (the memory is preloaded to eliminate the impact of page faults and TLB misses). We accept the null hypothesis for each value of n if the 1000 memory access latency is much higher when $k = C/n$ than others. Otherwise we reject the null hypothesis. This is because When $W = n$ and $k = C/n$, $m_i - m_{i-1} = L \times S$. Hence, the $2n$ memory accesses land in the same cache set, and on average n cache miss will take place in each loop. Therefore, the total execution time is longer because of $1000n$ cache misses. In other cases, these memory accesses do not land in the same cache set, so no cache miss will be observed. We calibrate the L1 data cache, L1 instruction and the L2 cache in separate tests. We repeated each run 20 times for statistical significance. Fig. 7a, Fig. 7b and Fig. 7c show the memory access latency in the tests where we correctly guessed W. It is clear in such cases $k = C/W$ leads to high access latency.



(a) L1 data cache (b) L1 instruction cache (c) L2 cache

Figure 7: Cache dimension test

B. CLEANSING CACHES

In this section, we describe techniques we developed to completely clease L2 caches on ARM. These techniques are used in Sec. 3.2, Sec. 3.3 and Appendix C.

Cleansing caches by fetching data. We start by allocating a memory buffer from which we select N cache-line-sized memory blocks that all map to the same cache set. These N memory blocks consist the *eviction set*. We then from the same buffer select a disjunct set of 5 memory blocks that also map to the same cache set. We call this set of 5 blocks the *test set*. An *eviction strategy* defines the size of the eviction set, N, and the order of accessing its memory blocks. To test the effectiveness of each strategy in cleansing the L2 cache, we first access the eviction set using the underlying strategy and then immediately afterwards load k (where k ranges from 1 to 5) memory blocks from the test set. The above process is repeated 1000 times and the average execution time of each loop is measured, denoted T_k. The cache set is completely evicted using the tested eviction strategy if $T_i - T_{i-1}$ are similar in magnitude for $i = 2, 3, 4, 5$ and are comparable to the time required for a cache miss.

Cleansing caches by executing instructions. To cleanse L2 cache by code execution, both the *eviction set* and the *test set* need to be filled with binary code that will transfer the control flows from one memory block to another, following specific order. Successfully doing so requires some engineering efforts. Particularly, we need to place the instructions in N discontinuous cache-line-aligned memory blocks that map to the same cache set, and then in each memory block calculate the virtual address of the next memory block and jump to the target to fetch the instruction into the L2 cache and L1 instruction cache.

Finding the best cleansing strategies. Each eviction strategy we explore can be uniquely identified by the size of the *eviction set*, N, the shift offset between two memory access sequences, A, and the number of continuously visited blocks in each memory access sequence, D (similar to [32]). The total memory accesses in each eviction strategy is its *cost*. Our goal is to find an eviction strategy that completely evicts a cache set but at the same time has the lowest cost. In our experiments, we used a brute-force approach to search the optimal eviction scheme[10]. The strategies listed in Table 5, although not optimal, can already cleanse the entire L2 cache with relatively low costs. We used these strategies in Sec. 3.2, Sec. 3.3 and Appendix C.

[10]Due to implementation complexity to arbitrarily adjust A and D, we fix them both as 1 in the instruction-based L2 cleansing tests.

Smartphone	Cache L2	d-Strategy N	d-Strategy A	d-Strategy D	i-Strategy N	i-Strategy A	i-Strategy D
Nexus 6	2MB 8-way	10	1	3	16	1	1
Samsung S5 (A15)	2MB 16-way	17	2	3	24	1	1
Samsung S5 (A7)	512KB 8-way	10	1	2	16	1	1
Samsung S6 (A57)	2MB 16-way	17	2	5	24	1	1
Samsung S6 (A53)	256KB 16-way	20	2	4	24	1	1

Table 5: Cache cleansing strategy.

C. EXCLUSIVE VS. NON-INCLUSIVE CACHES

Once the inclusiveness of the L2 cache has been precluded using methods described in Sec. 3.3, one could conduct the following experiments (using the same shared native library and dummy function described in Sec. 3.2) to determine if it is exclusive or non-inclusive to the L1 data cache.

First, the function code is read as data a few times to load it into the L1 data cache. Then the same app executes the function to measure its execution time, T_1. The measurement is compared with the results of the second experiment, in which the L2 cache is cleansed from the instruction side (see Appendix B) before executing it to ensure L2 cache misses. The time to execute the function in this case is denoted as T_2. If $T_1 \ll T_2$, T_1 is measured with L2 cache hits, which suggests reading the function code also brings the function body into the L2 cache. In this case, the L2 cache is non-inclusive to L1 the data cache; otherwise it is exclusive to the L1 data cache.

To run the same tests for the instruction cache, we first execute function a few times to make sure it was loaded into the L1 instruction cache. Then in the same program, immediately afterwards, we read the function body as data and measured the time for loading, denoted T_1. In a second test, the L2 cache is first cleansed from the data side (see Appendix B) and then the time needed to read the same function is measured as T_2. Essentially, T_2 reflects the time needed to read the function with L2 misses. If $T_1 \ll T_2$, T_1 is measured with L2 cache hits, which suggests executing the function brings the function body into the L2 cache. In this case, the L2 cache is non-inclusive to L1 instruction cache; otherwise it is exclusive to L1 instruction cache.