ARMageddon: Cache Attacks on Mobile Devices

Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, *and* Stefan Mangard
Graz University of Technology, Austria

Abstract

In the last 10 years, cache attacks on Intel x86 CPUs have gained increasing attention among the scientific community and powerful techniques to exploit cache side channels have been developed. However, modern smartphones use one or more multi-core ARM CPUs that have a different cache organization and instruction set than Intel x86 CPUs. So far, no cross-core cache attacks have been demonstrated on non-rooted Android smartphones. In this work, we demonstrate how to solve key challenges to perform the most powerful cross-core cache attacks *Prime+Probe*, *Flush+Reload*, *Evict+Reload*, and *Flush+Flush* on non-rooted ARM-based devices without any privileges. Based on our techniques, we demonstrate covert channels that outperform state-of-the-art covert channels on Android by several orders of magnitude. Moreover, we present attacks to monitor tap and swipe events as well as keystrokes, and even derive the lengths of words entered on the touchscreen. Eventually, we are the first to attack cryptographic primitives implemented in Java. Our attacks work across CPUs and can even monitor cache activity in the ARM TrustZone from the normal world. The techniques we present can be used to attack hundreds of millions of Android devices.

1 Introduction

Cache attacks represent a powerful means of exploiting the different access times within the memory hierarchy of modern system architectures. Until recently, these attacks explicitly targeted cryptographic implementations, for instance, by means of cache timing attacks [10] or the well-known *Evict+Time* and *Prime+Probe* techniques [44]. The seminal paper by Yarom and Falkner [61] introduced the so-called *Flush+Reload* attack, which allows an attacker to infer which specific parts of a binary are accessed by a victim program with an unprecedented accuracy and probing frequency. Recently, Gruss et al. [20] demonstrated the possibility to use *Flush+Reload* to automatically exploit cache-based side channels via cache template attacks on Intel platforms. *Flush+Reload* does not only allow for efficient attacks against cryptographic implementations [9, 27, 57], but also to infer keystroke information and even to build keyloggers on Intel platforms [20]. In contrast to attacks on cryptographic algorithms, which are typically triggered multiple times, these attacks require a significantly higher accuracy as an attacker has only one single chance to observe a user input event.

Although a few publications about cache attacks on AES T-table implementations on mobile devices exist [11, 51–53, 58], the more efficient cross-core attack techniques *Prime+Probe*, *Flush+Reload*, *Evict+Reload*, and *Flush+Flush* [19] have not been applied on smartphones. In fact, there was reasonable doubt [61] whether these cross-core attacks can be mounted on ARM-based devices at all. In this work, we demonstrate that these attack techniques are applicable on ARM-based devices by solving the following key challenges systematically:

1. *Last-level caches are not inclusive on ARM and thus cross-core attacks cannot rely on this property.* Indeed, existing cross-core attacks exploit the inclusiveness of shared last-level caches [19, 20, 23, 25, 36, 38, 39, 43, 61] and, thus, no cross-core attacks have been demonstrated on ARM so far. We present an approach that exploits coherence protocols and L1-to-L2 transfers to make these attacks applicable on mobile devices with non-inclusive shared last-level caches, irrespective of the cache organization.[1]
2. *Most modern smartphones have multiple CPUs that do not share a cache.* However, cache coherence

Original publication in the Proceedings of the 25th Annual USENIX Security Symposium (USENIX Security 2016).
https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/lipp

[1] Simultaneously to our work on ARM, Irazoqui et al. [26] developed a technique to exploit cache coherence protocols on AMD x86 CPUs and mounted the first cross-CPU cache attack.

protocols allow CPUs to fetch cache lines from remote cores faster than from the main memory. We utilize this property to mount both cross-core and cross-CPU attacks.

3. *Except ARMv8-A CPUs, ARM processors do not support a flush instruction.* In these cases, a fast eviction strategy must be applied for high-frequency measurements. As existing eviction strategies are too slow, we analyze more than 4 200 eviction strategies for our test devices, based on Rowhammer attack techniques [18].
4. *ARM CPUs use a pseudo-random replacement policy* to decide which cache line to replace within a cache set. This introduces additional noise even for robust time-driven cache attacks [51, 53]. For the same reason, *Prime+Probe* has been an open challenge [52] on ARM, as an attacker needs to predict which cache line will be replaced first and wrong predictions destroy measurements. We design reaccess loops that interlock with a cache eviction strategy to reduce the effect of wrong predictions.
5. *Cycle-accurate timings require root access on ARM* [4] and alternatives have not been evaluated so far. We evaluate different timing sources and show that cache attacks can be mounted in any case.

Based on these building blocks, we demonstrate practical and highly efficient cache attacks on ARM.[2] We do not restrict our investigations to cryptographic implementations but also consider cache attacks as a means to infer other sensitive information—such as inter-keystroke timings or the length of a swipe action—requiring a significantly higher measurement accuracy. Besides these generic attacks, we also demonstrate that cache attacks can be used to monitor cache activity caused within the ARM TrustZone from the normal world. Nevertheless, we do not aim to exhaustively list possible exploits or find new attack vectors on cryptographic algorithms. Instead, we aim to demonstrate the immense attack potential of the presented cross-core and cross-CPU attacks on ARM-based mobile devices based on well-studied attack vectors. Our work allows to apply existing attacks to millions of off-the-shelf Android devices without any privileges. Furthermore, our investigations show that Android still employs vulnerable AES T-table implementations.

Contributions. The contributions of this work are:

- We demonstrate the applicability of highly efficient cache attacks like *Prime+Probe*, *Flush+Reload*, *Evict+Reload*, and *Flush+Flush* on ARM.
- Our attacks work irrespective of the actual cache organization and, thus, are the first last-level cache attacks that can be applied cross-core and also cross-CPU on off-the-shelf ARM-based devices. More specifically, our attacks work against last-level caches that are instruction-inclusive and data-non-inclusive as well as caches that are instruction-non-inclusive and data-inclusive.
- Our cache-based covert channel outperforms all existing covert channels on Android by several orders of magnitude.
- We demonstrate the power of these attacks by attacking cryptographic implementations and by inferring more fine-grained information like keystrokes and swipe actions on the touchscreen.

Outline. The remainder of this paper is structured as follows. In Section 2, we provide information on background and related work. Section 3 describes the techniques that are the building blocks for our attacks. In Section 4, we demonstrate and evaluate fast cross-core and cross-CPU covert channels on Android. In Section 5, we demonstrate cache template attacks on user input events. In Section 6, we present attacks on cryptographic implementations used in practice as well the possibility to observe cache activity of cryptographic computations within the TrustZone. We discuss countermeasures in Section 7 and conclude this work in Section 8.

2 Background and Related Work

In this section, we provide the required preliminaries and discuss related work in the context of cache attacks.

2.1 CPU Caches

Today's CPU performance is influenced not only by the clock frequency but also by the latency of instructions, operand fetches, and other interactions with internal and external devices. In order to overcome the latency of system memory accesses, CPUs employ caches to buffer frequently used data in small and fast internal memories.

Modern caches organize cache lines in multiple sets, which is also known as set-associative caches. Each memory address maps to one of these cache sets and addresses that map to the same cache set are considered congruent. Congruent addresses compete for cache lines within the same set and a predefined replacement policy determines which cache line is replaced. For instance, the last generations of Intel CPUs employ an undocumented variant of least-recently used (LRU) replacement policy [18]. ARM processors use a pseudo-LRU replacement policy for the L1 cache and they support two different cache replacement policies for L2 caches, namely round-robin and pseudo-random replacement policy. In

[2] Source code for ARMageddon attack examples can be found at `https://github.com/IAIK/armageddon`.

practice, however, only the pseudo-random replacement policy is used due to performance reasons. Switching the cache replacement policy is only possible in privileged mode. The implementation details for the pseudo-random policy are not documented.

CPU caches can either be virtually indexed or physically indexed, which determines whether the index is derived from the virtual or physical address. A so-called tag uniquely identifies the address that is cached within a specific cache line. Although this tag can also be based on the virtual or physical address, most modern caches use physical tags because they can be computed simultaneously while locating the cache set. ARM typically uses physically indexed, physically tagged L2 caches.

CPUs have multiple cache levels, with the lower levels being faster and smaller than the higher levels. ARM processors typically have two levels of cache. If all cache lines from lower levels are also stored in a higher-level cache, the higher-level cache is called *inclusive*. If a cache line can only reside in one of the cache levels at any point in time, the caches are called *exclusive*. If the cache is neither inclusive nor exclusive, it is called *non-inclusive*. The last-level cache is often shared among all cores to enhance the performance upon transitioning threads between cores and to simplify cross-core cache lookups. However, with shared last-level caches, one core can (intentionally) influence the cache content of all other cores. This represents the basis for cache attacks like *Flush+Reload* [61].

In order to keep caches of multiple CPU cores or CPUs in a coherent state, so-called coherence protocols are employed. However, coherence protocols also introduce exploitable timing effects, which has recently been exploited by Irazoqui et al. [26] on x86 CPUs.

In this paper, we demonstrate attacks on three smartphones as listed in Table 1. The Krait 400 is an ARMv7-A CPU, the other two processors are ARMv8-A CPUs. However, the stock Android of the Alcatel One Touch Pop 2 is compiled for an ARMv7-A instruction set and thus ARMv8-A instructions are not used. We generically refer to ARMv7-A and ARMv8-A as "ARM architecture" throughout this paper. All devices have a shared L2 cache. On the Samsung Galaxy S6, the flush instruction is unlocked by default, which means that it is available in userspace. Furthermore, all devices employ a cache coherence protocol between cores and on the Samsung Galaxy S6 even between the two CPUs [7].

2.2 Shared Memory

Read-only shared memory can be used as a means of memory usage optimization. In case of shared libraries it reduces the memory footprint and enhances the speed by lowering cache contention. The operating system implements this behavior by mapping the same physical memory into the address space of each process. As this memory sharing mechanism is independent of how a file was opened or accessed, an attacker can map a binary to have read-only shared memory with a victim program. A similar effect is caused by content-based page deduplication where physical pages with identical content are merged.

Android applications are usually written in Java and, thus, contain self-modifying code or just-in-time compiled code. This code would typically not be shared. Since Android version 4.4 the Dalvik VM was gradually replaced by the Android Runtime (ART). With ART, Java byte code is compiled to native code binaries [2] and thus can be shared too.

2.3 Cache Attacks

Initially, cache timing attacks were performed on cryptographic algorithms [10, 31, 32, 41, 42, 45, 56]. For example, Bernstein [10] exploited the total execution time of AES T-table implementations. More fine-grained exploitations of memory accesses to the CPU cache have been proposed by Percival [46] and Osvik et al. [44]. More specifically, Osvik et al. formalized two concepts, namely *Evict+Time* and *Prime+Probe*, to determine which specific cache sets were accessed by a victim program. Both approaches consist of three basic steps.

***Evict+Time*:**

1. Measure execution time of victim program.
2. Evict a specific cache set.
3. Measure execution time of victim program again.

***Prime+Probe*:**

1. Occupy specific cache sets.
2. Victim program is scheduled.
3. Determine which cache sets are still occupied.

Both approaches allow an adversary to determine which cache sets are used during the victim's computations and have been exploited to attack cryptographic implementations [25, 36, 44, 55] and to build cross-VM covert channels [38]. Yarom and Falkner [61] proposed *Flush+Reload*, a significantly more fine-grained attack that exploits three fundamental concepts of modern system architectures. First, the availability of shared memory between the victim process and the adversary. Second, last-level caches are typically shared among all cores. Third, Intel platforms use inclusive last-level caches, meaning that the eviction of information from the last-level cache leads to the eviction of this data from all lower-level caches of other cores, which allows any program to evict data from other programs on other cores. While the basic idea of this attack has been proposed by Gullasch et al. [22], Yarom and Falkner extended this idea to shared last-level caches, allowing cross-core attacks. *Flush+Reload* works as follows.

Table 1: Test devices used in this paper.

Device	SoC	CPU (cores)	L1 caches	L2 cache	Inclusiveness
OnePlus One	Qualcomm Snapdragon 801	Krait 400 (2) 2.5 GHz	2× 16 KB, 4-way, 64 sets	2 048 KB, 8-way, 2 048 sets	non-inclusive
Alcatel One Touch Pop 2	Qualcomm Snapdragon 410	Cortex-A53 (4) 1.2 GHz	4× 32 KB, 4-way, 128 sets	512 KB, 16-way, 512 sets	instruction-inclusive, data-non-inclusive
Samsung Galaxy S6	Samsung Exynos 7 Octa 7420	Cortex-A53 (4) 1.5 GHz Cortex-A57 (4) 2.1 GHz	4× 32 KB, 4-way, 128 sets 4× 32 KB, 2-way, 256 sets	256 KB, 16-way, 256 sets 2 048 KB, 16-way, 2 048 sets	instruction-inclusive, data-non-inclusive instruction-non-inclusive, data-inclusive

***Flush+Reload*:**

1. Map binary (e.g., shared object) into address space.
2. Flush a cache line (code or data) from the cache.
3. Schedule the victim program.
4. Check if the corresponding line from step 2 has been loaded by the victim program.

Thereby, *Flush+Reload* allows an attacker to determine which specific instructions are executed and also which specific data is accessed by the victim program. Thus, rather fine-grained attacks are possible and have already been demonstrated against cryptographic implementations [23, 28, 29]. Furthermore, Gruss et al. [20] demonstrated the possibility to automatically exploit cache-based side-channel information based on the *Flush+Reload* approach. Besides attacking cryptographic implementations like AES T-table implementations, they showed how to infer keystroke information and even how to build a keylogger by exploiting the cache side channel. Similarly, Oren et al. [43] demonstrated the possibility to exploit cache attacks on Intel platforms from JavaScript and showed how to infer visited websites and how to track the user's mouse activity.

Gruss et al. [20] proposed the *Evict+Reload* technique that replaces the flush instruction in *Flush+Reload* by eviction. While it has no practical application on x86 CPUs, we show that it can be used on ARM CPUs. Recently, *Flush+Flush* [19] has been proposed. Unlike other techniques, it does not perform any memory access but relies on the timing of the flush instruction to determine whether a line has been loaded by a victim. We show that the execution time of the ARMv8-A flush instruction also depends on whether or not data is cached and, thus, can be used to implement this attack.

While the attacks discussed above have been proposed and investigated for Intel processors, the same attacks were considered not applicable to modern smartphones due to differences in the instruction set, the cache organization [61], and in the multi-core and multi-CPU architecture. Thus, only same-core cache attacks have been demonstrated on smartphones so far. For instance, Weiß et al. [58] investigated Bernstein's cache-timing attack [10] on a Beagleboard employing an ARM Cortex-A8 processor. Later on, Weiß et al. [59] investigated this timing attack in a multi-core setting on a development board. As Weiß et al. [58] claimed that noise makes the attack difficult, Spreitzer and Plos [53] investigated the applicability of Bernstein's cache-timing attack on different ARM Cortex-A8 and ARM Cortex-A9 smartphones running Android. Both investigations [53, 58] confirmed that timing information is leaking, but the attack takes several hours due to the high number of measurement samples that are required, *i.e.*, about 2^{30} AES encryptions. Later on, Spreitzer and Gérard [51] improved upon these results and managed to reduce the key space to a complexity which is practically relevant.

Besides Bernstein's attack, another attack against AES T-table implementations has been proposed by Bogdanov et al. [11], who exploited so-called wide collisions on an ARM9 microprocessor. In addition, power analysis attacks [14] and electromagnetic emanations [15] have been used to visualize cache accesses during AES computations on ARM microprocessors. Furthermore, Spreitzer and Plos [52] implemented *Evict+Time* [44] in order to attack an AES T-table implementation on Android-based smartphones. However, so far only cache attacks against AES T-table implementations have been considered on smartphone platforms and none of the recent advances have been demonstrated on mobile devices.

3 ARMageddon Attack Techniques

We consider a scenario where an adversary attacks a smartphone user by means of a malicious application. This application *does not require any permission* and, most importantly, it can be executed in unprivileged userspace and *does not require a rooted device*. As our attack techniques do not exploit specific vulnerabilities of Android versions, they work on stock Android ROMs as well as customized ROMs in use today.

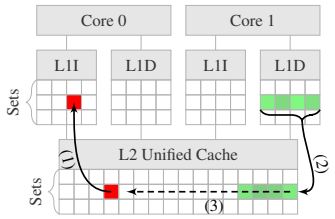

Figure 1: Cross-core instruction cache eviction through data accesses.

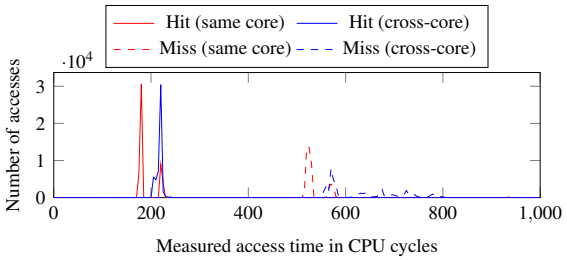

Figure 2: Histograms of cache hits and cache misses measured same-core and cross-core on the OnePlus One.

3.1 Defeating the Cache Organization

In this section, we tackle the aforementioned challenges 1 and 2, *i.e.*, the last-level cache is not inclusive and multiple processors do not necessarily share a cache level.

When it comes to caches, ARM CPUs are very heterogeneous compared to Intel CPUs. For example, whether or not a CPU has a second-level cache can be decided by the manufacturer. Nevertheless, the last-level cache on ARM devices is usually shared among all cores and it can have different inclusiveness properties for instructions and data. Due to cache coherence, shared memory is kept in a coherent state across cores and CPUs. This is of importance when measuring timing differences between cache accesses and memory accesses (cache misses), as fast remote-cache accesses are performed instead of slow memory accesses [7]. In case of a non-coherent cache, a cross-core attack is not possible but an attacker can run the spy process on all cores simultaneously and thus fall back to a same-core attack. However, we observed that caches are coherent on all our test devices.

To perform a cross-core attack we load enough data into the cache to fully evict the corresponding last-level cache set. Thereby, we exploit that we can fill the last-level cache directly or indirectly depending on the cache organization. On the Alcatel One Touch Pop 2, the last-level cache is instruction-inclusive and thus we can evict instructions from the local caches of the other core. Figure 1 illustrates such an eviction. In step 1, an instruction is allocated to the last-level cache and the instruction cache of one core. In step 2, a process fills its core's data cache, thereby evicting cache lines into the last-level cache. In step 3, the process has filled the last-level cache set using only data accesses and thereby evicts the instructions from instruction caches of other cores as well.

We access cache lines multiple times to perform transfers between L1 and L2 cache. Thus, more and more addresses used for eviction are cached in either L1 or L2. As ARM CPUs typically have L1 caches with a very low associativity, the probability of eviction to L2 through other system activity is high. Using an eviction strategy that performs frequent transfers between L1 and L2 increases this probability further. Thus, this approach also works for other cache organizations to perform cross-core and cross-CPU cache attacks. Due to the cache coherence protocol between the CPU cores [7, 34], remote-core fetches are faster than memory accesses and thus can be distinguished from cache misses. For instance, Figure 2 shows the cache hit and miss histogram on the OnePlus One. The cross-core access introduces a latency of 40 CPU cycles on average. However, cache misses take more than 500 CPU cycles on average. Thus, cache hits and misses are clearly distinguishable based on a single threshold value.

3.2 Fast Cache Eviction

In this section, we tackle the aforementioned challenges 3 and 4, *i.e.*, not all ARM processors support a flush instruction, and the replacement policy is pseudo-random.

There are two options to evict cache lines: (1) the flush instruction or (2) evict data with memory accesses to congruent addresses, *i.e.*, addresses that map to the same cache set. As the flush instruction is only available on the Samsung Galaxy S6, we need to rely on eviction strategies for the other devices and, therefore, to defeat the replacement policy. The L1 cache in Cortex-A53 and Cortex-A57 has a very small number of ways and employs a least-recently used (LRU) replacement policy [6]. However, for a full cache eviction, we also have to evict cache lines from the L2 cache, which uses a pseudo-random replacement policy.

Eviction strategies. Previous approaches to evict data on Intel x86 platforms either have too much overhead [24] or are only applicable to caches implementing an LRU replacement policy [36, 38, 43]. Spreitzer and Plos [52] proposed an eviction strategy for ARMv7-A CPUs that requires to access more addresses than there are cache lines per cache set, due to the pseudo-random replacement policy. Recently, Gruss et al. [18] demonstrated how to automatically find fast eviction strategies on Intel x86 architectures. We show that their algorithm is applicable to ARM CPUs as well.

Table 2: Different eviction strategies on the Krait 400.

N	A	D	Cycles	Eviction rate
-	-	-	549	100.00%
11	2	2	1 578	100.00%
12	1	3	2 094	100.00%
13	1	5	2 213	100.00%
16	1	1	3 026	100.00%
24	1	1	4 371	100.00%
13	1	2	2 372	99.58%
11	1	3	1 608	80.94%
11	4	1	1 948	58.93%
10	2	2	1 275	51.12%

Table 3: Different eviction strategies on the Cortex-A53.

N	A	D	Cycles	Eviction rate
-	-	-	767	100.00%
23	2	5	6 209	100.00%
23	4	6	16 912	100.00%
22	1	6	5 101	99.99%
21	1	6	4 275	99.93%
20	4	6	13 265	99.44%
800	1	1	142 876	99.10%
200	1	1	33 110	96.04%
100	1	1	15 493	89.77%
48	1	1	6 517	70.78%

Thereby, we establish eviction strategies in an automated way and significantly reduce the overhead compared to [52]. We evaluated more than 4 200 access patterns on our smartphones and identified the best eviction strategies. Even though the cache employs a random replacement policy, average eviction rate and average execution time are reproducible. Eviction sets are computed based on physical addresses, which can be retrieved via `/proc/self/pagemap` as current Android versions allow access to these mappings to any unprivileged app without any permissions. Thus, eviction patterns and eviction sets can be efficiently computed.

We applied the algorithm of Gruss et al. [18] to a set of physically congruent addresses. Table 2 summarizes different eviction strategies, *i.e.*, loop parameters, for the Krait 400. N denotes the total eviction set size (length of the loop), A denotes the shift offset (loop increment) to be applied after each round, and D denotes the number of memory accesses in each iteration (loop body). The column *cycles* states the average execution time in CPU cycles over 1 million evictions and the last column denotes the average eviction rate. The first line in Table 2 shows the average execution time and the average eviction rate for the privileged flush instruction, which gives the best result in terms of average execution time (549 CPU cycles). We evaluated 1 863 different strategies and our best identified eviction strategy ($N = 11$, $A = 2$, $D = 2$) also achieves an average eviction rate of 100% but takes 1 578 CPU cycles. Although a strategy accessing every address in the eviction set only once ($A = 1$, $D = 1$, also called LRU eviction) performs significantly fewer memory accesses, it consumes more CPU cycles. For an average eviction rate of 100%, LRU eviction requires an eviction set size of at least 16. The average execution time then is 3 026 CPU cycles. Considering the eviction strategy used in [52] that takes 4 371 CPU cycles, clearly demonstrates the advantage of our optimized eviction strategy that takes only 1 578 CPU cycles.

We performed the same evaluation with 2 295 different strategies on the ARM Cortex-A53 in our Alcatel One Touch Pop 2 test system and summarize them in Table 3. For the best strategy we found ($N = 21, A = 1, D = 6$), we measured an average eviction rate of 99.93% and an average execution time of 4 275 CPU cycles. We observed that LRU eviction ($A = 1$, $D = 1$) on the ARM Cortex-A53 would take 28 times more CPU cycles to achieve an average eviction rate of only 99.10%, thus it is not suitable for attacks on the last-level cache as used in previous work [52]. The reason for this is that data can only be allocated to L2 cache by evicting it from the L1 cache on the ARM Cortex-A53. Therefore, it is better to reaccess the data that is already in the L2 cache and gradually add new addresses to the set of cached addresses instead of accessing more different addresses.

On the ARM Cortex-A57 the userspace flush instruction was significantly faster in any case. Thus, for *Flush+Reload* we use the flush instruction and for *Prime+Probe* the eviction strategy. Falling back to *Evict+Reload* is not necessary on the Cortex-A57. Similarly to recent Intel x86 CPUs, the execution time of the flush instruction on ARM depends on whether or not the value is cached, as shown in Figure 3. The execution time is higher if the address is cached and lower if the address is not cached. This observation allows us to distinguish between cache hits and cache misses depending on the timing behavior of the flush instruction, and therefore to perform a *Flush+Flush* attack. Thus, in case of shared memory between the victim and the attacker, it is not even required to evict and reload an address in order to exploit the cache side channel.

A note on *Prime+Probe*. Finding a fast eviction strategy for *Prime+Probe* on architectures with a random replacement policy is not as straightforward as on Intel x86. Even in case of x86 platforms, the problem of cache trashing has been discussed by Tromer et al. [55]. Cache trashing occurs when reloading (probing) an address evicts one of the addresses that are to be accessed next. While Tromer et al. were able to overcome this problem by using a doubly-linked list that is accessed

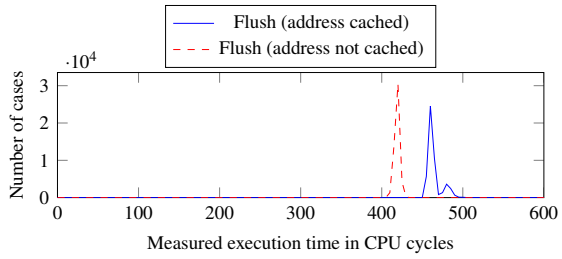

Figure 3: Histograms of the execution time of the flush operation on cached and not cached addresses measured on the Samsung Galaxy S6.

forward during the prime step and backwards during the probe step, the random replacement policy on ARM also contributes to the negative effect of cache trashing.

We analyzed the behavior of the cache and designed a prime step and a probe step that work with a smaller set size to avoid set thrashing. Thus, we set the eviction set size to 15 on the Alcatel One Touch Pop 2. As we run the *Prime+Probe* attack in a loop, exactly 1 way in the L2 cache will not be occupied after a few attack rounds. We might miss a victim access in $\frac{1}{16}$ of the cases, which however is necessary as otherwise we would not be able to get reproducible measurements at all due to set thrashing. If the victim replaces one of the 15 ways occupied by the attacker, there is still one free way to reload the address that was evicted. This reduces the chance of set thrashing significantly and allows us to successfully perform *Prime+Probe* on caches with a random replacement policy.

3.3 Accurate Unprivileged Timing

In this section, we tackle the aforementioned challenge 5, *i.e.*, cycle-accurate timings require root access on ARM.

In order to distinguish cache hits and cache misses, timing sources or dedicated performance counters can be used. We focus on timing sources, as cache misses have a significantly higher access latency and timing sources are well studied on Intel x86 CPUs. Cache attacks on x86 CPUs employ the unprivileged `rdtsc` instruction to obtain a sub-nanosecond resolution timestamp. The ARMv7-A architecture does not provide an instruction for this purpose. Instead, the ARMv7-A architecture has a performance monitoring unit that allows to monitor CPU activity. One of these performance counters—denoted as *cycle count register* (PMCCNTR)—can be used to distinguish cache hits and cache misses by relying on the number of CPU cycles that passed during a memory access. However, these performance counters are not accessible from userspace by default and an attacker would need root privileges.

We broaden the attack surface by exploiting timing sources that are accessible without any privileges or permissions. We identified three possible alternatives for timing measurements.

Unprivileged syscall. The `perf_event_open` syscall is an abstract layer to access performance information through the kernel independently of the underlying hardware. For instance, `PERF_COUNT_HW_CPU_CYCLES` returns an accurate cycle count including a minor overhead due to the syscall. The availability of this feature depends on the Android kernel configuration, e.g., the stock kernel on the Alcatel One Touch Pop 2 as well as the OnePlus One provide this feature by default. Thus, in contrast to previous work [52], the attacker does not have to load a kernel module to access this information as the `perf_event_open` syscall can be accessed without any privileges or permissions.

POSIX function. Another alternative to obtain sufficiently accurate timing information is the POSIX function `clock_gettime()`, with an accuracy in the range of microseconds to nanoseconds. Similar information can also be obtained from `/proc/timer_list`.

Dedicated thread timer. If no interface with sufficient accuracy is available, an attacker can run a thread that increments a global variable in a loop, providing a fair approximation of a cycle counter. Our experiments show that this approach works reliably on smartphones as well as recent x86 CPUs. The resolution of this threaded timing information is as high as with the other methods.

In Figure 4 we show the cache hit and miss histogram based on the four different methods, including the cycle count register, on a Alcatel One Touch Pop 2. Despite the latency and noise, cache hits and cache misses are clearly distinguishable with all approaches. Thus, all methods can be used to implement cache attacks. Determining the best timing method on the device under attack can be done in a few seconds during an online attack.

4 High Performance Covert Channels

To evaluate the performance of our attacks, we measure the capacity of cross-core and cross-CPU cache covert channels. A covert channel enables two unprivileged applications on a system to communicate with each other without using any data transfer mechanisms provided by the operating system. This communication evades the sandboxing concept and the permission system (cf. collusion attacks [37]). Both applications were running in

Table 4: Comparison of covert channels on Android.

Work	Type	Bandwidth [bps]	Error rate
Ours (Samsung Galaxy S6)	*Flush+Reload*, cross-core	**1 140 650**	1.10%
Ours (Samsung Galaxy S6)	*Flush+Reload*, cross-CPU	**257 509**	1.83%
Ours (Samsung Galaxy S6)	*Flush+Flush*, cross-core	**178 292**	0.48%
Ours (Alcatel One Touch Pop 2)	*Evict+Reload*, cross-core	**13 618**	3.79%
Ours (OnePlus One)	*Evict+Reload*, cross-core	**12 537**	5.00%
Marforio et al. [37]	Type of Intents	4 300	–
Marforio et al. [37]	UNIX socket discovery	2 600	–
Schlegel et al. [49]	File locks	685	–
Schlegel et al. [49]	Volume settings	150	–
Schlegel et al. [49]	Vibration settings	87	–

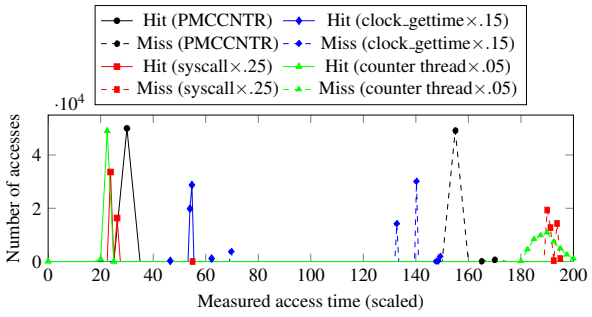

Figure 4: Histogram of cross-core cache hits/misses on the Alcatel One Touch Pop 2 using different methods. X-values are scaled for visual representation.

the background while the phone was mostly idle and an unrelated app was running as the foreground application.

Our covert channel is established on addresses of a shared library that is used by both the sender and the receiver. While both processes have read-only access to the shared library, they can transmit information by loading addresses from the shared library into the cache or evicting (flushing) it from the cache, respectively.

The covert channel transmits packets of n-bit data, an s-bit sequence number, and a c-bit checksum that is computed over data and sequence number. The sequence number is used to distinguish consecutive packets and the checksum is used to check the integrity of the packet. The receiver acknowledges valid packets by responding with an s-bit sequence number and an x-bit checksum. By adjusting the sizes of checksums and sequence numbers the error rate of the covert channel can be controlled.

Each bit is represented by one address in the shared library, whereas no two addresses are chosen that map to the same cache set. To transmit a bit value of 1, the sender accesses the corresponding address in the library. To transmit a bit value of 0, the sender does not access the corresponding address, resulting in a cache miss on the receiver's side. Thus, the receiving process observes a cache hit or a cache miss depending on the memory access performed by the sender. The same method is used for the acknowledgements sent by the receiving process.

We implemented this covert channel using *Evict+Reload*, *Flush+Reload*, and *Flush+Flush* on our smartphones. The results are summarized in Table 4. On the Samsung Galaxy S6, we achieve a cross-core transmission rate of 1 140 650 bps at an error rate of 1.10%. This is 265 times faster than any existing covert channel on smartphones. In a cross-CPU transmission we achieve a transmission rate of 257 509 bps at an error rate of 1.83%. We achieve a cross-core transition rate of 178 292 bps at an error rate of 0.48% using *Flush+Flush* on the Samsung Galaxy S6. On the Alcatel One Touch Pop 2 we achieve a cross-core transmission rate of 13 618 bps at an error rate of 3.79% using *Evict+Reload*. This is still 3 times faster than previous covert channels on smartphones. The covert channel is significantly slower on the Alcatel One Touch Pop 2 than on the Samsung Galaxy S6 because the hardware is much slower, *Evict+Reload* is slower than *Flush+Reload*, and retransmission might be necessary in 0.14% of the cases where eviction is not successful (cf. Section 3.2). On the older OnePlus One we achieve a cross-core transmission rate of 12 537 bps at an error rate of 5.00%, 3 times faster than previous covert channels on smartphones. The reason for the higher error rate is the additional timing noise due to the cache coherence protocol performing a high number of remote-core fetches.

5 Attacking User Input on Smartphones

In this section we demonstrate cache side-channel attacks on Android smartphones. We implement cache template attacks [20] to create and exploit accurate cache-usage profiles using the *Evict+Reload* or *Flush+Reload* attack. Cache template attacks have a profiling phase and an exploitation phase. In the profiling phase, a template matrix is computed that represents how many cache hits occur on a specific address when trig-

gering a specific event. The exploitation phase uses this matrix to infer events from cache hits.

To perform cache template attacks, an attacker has to map shared binaries or shared libraries as read-only shared memory into its own address space. By using shared libraries, the attacker bypasses any potential countermeasures taken by the operating system, such as restricted access to runtime data of other apps or address space layout randomization (ASLR). The attack can even be performed online on the device under attack if the event can be simulated.

Triggering the actual event that an attacker wants to spy on might require either (1) an offline phase or (2) privileged access. For instance, in case of a keylogger, the attacker can gather a cache template matrix offline for a specific version of a library, or the attacker relies on privileged access of the application (or a dedicated permission) in order to be able to simulate events for gathering the cache template matrix. However, the actual exploitation of the cache template matrix to infer events neither requires privileged access nor any permission.

5.1 Attacking a Shared Library

Just as Linux, Android uses a large number of shared libraries, each with a size of up to several megabytes. We inspected all available libraries on the system by manually scanning the names and identified libraries that might be responsible for handling user input, e.g., the `libinput.so` library. Without loss of generality, we restricted the set of attacked libraries since testing all libraries would have taken a significant amount of time. Yet, an adversary could exhaustively probe all libraries.

We automated the search for addresses in these shared libraries and after identifying addresses, we monitored them in order to infer user input events. For instance, in the profiling phase on `libinput.so`, we simulated events via the android-debug bridge (adb shell) with two different methods. The first method uses the `input` command line tool to simulate user input events. The second method is writing event messages to `/dev/input/event*`. Both methods can run entirely on the device for instance in idle periods while the user is not actively using the device. As the second method only requires a `write()` statement it is significantly faster, but it is also more device specific. Therefore, we used the `input` command line except when profiling differences between different letter keys. While simulating these events, we simultaneously probed all addresses within the `libinput.so` library, *i.e.*, we measured the number of cache hits that occurred on each address when triggering a specific event. As already mentioned above, the simulation of some events might require either an offline phase or specific privileges in case of online attacks.

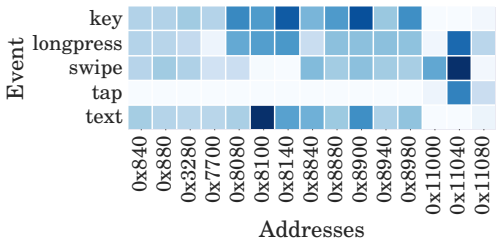

Figure 5: Cache template matrix for `libinput.so`.

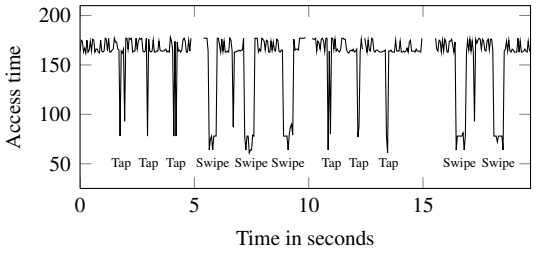

Figure 6: Monitoring address `0x11040` of `libinput.so` on the Alcatel One Touch Pop 2 reveals taps and swipes.

Figure 5 shows part of the cache template matrix for `libinput.so`. We triggered the following events: key events including the power button (*key*), long touch events (*longpress*), *swipe* events, touch events (*tap*), and text input events (*text*) via the `input` tool as often as possible and measured each address and event for one second. The cache template matrix clearly reveals addresses with high cache-hit rates for specific events. Darker colors represent addresses with higher cache-hit rates for a specific event and lighter colors represent addresses with lower cache-hit rates. Hence, we can distinguish different events based on cache hits on these addresses.

We verified our results by monitoring the identified addresses while operating the smartphone manually, *i.e.*, we touched the screen and our attack application reliably reported cache hits on the monitored addresses. For instance, address `0x11040` of `libinput.so` can be used to distinguish tap actions and swipe actions on the screen of the Alcatel One Touch Pop 2. Tap actions cause a smaller number of cache hits than swipe actions. Swipe actions cause cache hits in a high frequency as long as the screen is touched. Figure 6 shows a sequence of 3 tap events, 3 swipe events, 3 tap events, and 2 swipe events. These events can be clearly distinguished due to the fast access times. The gaps mark periods of time where our program was not scheduled on the CPU. Events occurring in those periods can be missed by our attack.

Swipe input allows to enter words by swiping over the soft-keyboard and thereby connecting single characters to form a word. Since we are able to determine the length of swipe movements, we can correlate the length

of the swipe movement with the actual word length in any Android application or system interface that uses swipe input without any privileges. Furthermore, we can determine the actual length of the unlock pattern for the pattern-unlock mechanism.

Figure 7 shows a user input sequence consisting of 3 tap events and 3 swipe events on the Samsung Galaxy S6. The attack was conducted using *Flush+Reload*. An attacker can monitor every single event. Taps and swipes can be distinguished based on the length of the cache hit phase. The length of a swipe movement can be determined from the same information. Figure 8 shows the same experiment on the OnePlus One using *Evict+Reload*. Thus, our attack techniques work on coherent non-inclusive last-level caches.

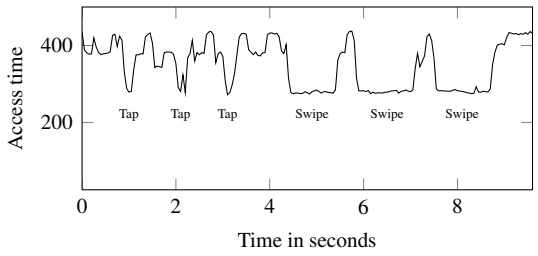

Figure 7: Monitoring address `0xDC5C` of `libinput.so` on the Samsung Galaxy S6 reveals tap and swipe events.

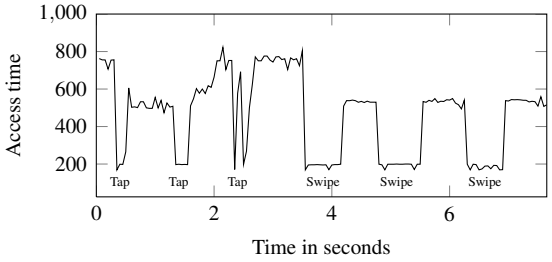

Figure 8: Monitoring address `0xBFF4` of `libinput.so` on the OnePlus One reveals tap and swipe events.

5.2 Attacking ART Binaries

Instead of attacking shared libraries, it is also possible to apply this attack to ART (Android Runtime) executables [2] that are compiled ahead of time. We used this attack on the default AOSP keyboard and evaluated the number of accesses to every address in the optimized executable that responds to an input of a letter on the keyboard. It is possible to find addresses that correspond to a key press and more importantly to distinguish between taps and key presses. Figure 9 shows the corresponding cache template matrix. We summarize the letter keys

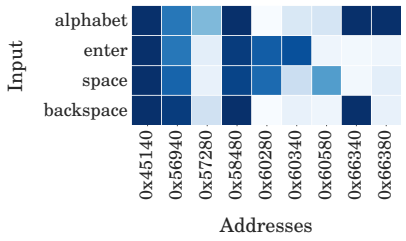

Figure 9: Cache template matrix for the default AOSP keyboard.

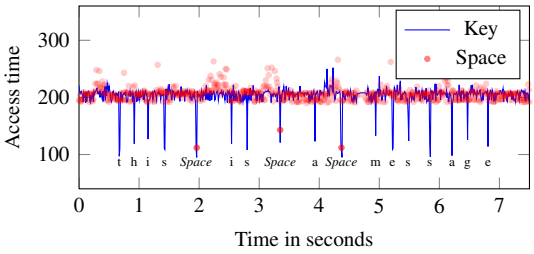

Figure 10: *Evict+Reload* on 2 addresses in `custpack@app@withoutlibs@LatinIME.apk@classes.dex` on the Alcatel One Touch Pop 2 while entering the sentence "this is a message".

in one line (*alphabet*) as they did not vary significantly. These addresses can be used to monitor key presses on the keyboard. We identified an address that corresponds only to letters on the keyboard and hardly on the space bar or the return button. With this information it is possible to precisely determine the length of single words entered using the default AOSP keyboard.

We illustrate the capability of detecting word lengths in Figure 10. The blue line shows the timing measurements for the address identified for keys in general, the red dots represent measurements of the address for the space key. The plot shows that we can clearly determine the length of entered words and monitor user input accurately over time.

5.3 Discussion and Impact

Our proof-of-concept attacks exploit shared libraries and binaries from Android apk files to infer key strokes. The cache template attack technique we used for these attacks is generic and can also be used to attack any other library. For instance, there are various libraries that handle different hardware modules and software events on the device, such as GPS, Bluetooth, camera, NFC, vibrator, audio and video decoding, web and PDF viewers. Each of these libraries contains code that is executed and data that is accessed when the device is in use. Thus, an attacker can perform a cache template attack on any

of these libraries and spy on the corresponding device events. For instance, our attack can be used to monitor activity of the GPS sensor, bluetooth, or the camera. An attacker can record such user activities over time to learn more about the user.

We can establish inter-keystroke timings at an accuracy as high as the accuracy of cache side-channel attacks on keystrokes on x86 systems with a physical keyboard. Thus, the inter-keystroke timings can be used to infer entered words, as has been shown by Zhang et al. [62]. Our attack even has a higher resolution than [62], *i.e.*, it is sub-microsecond accurate. Furthermore, we can distinguish between keystrokes on the soft-keyboard and generic touch actions outside the soft-keyboard. This information can be used to enhance sensor-based keyloggers that infer user input on mobile devices by exploiting, e.g., the accelerometer and the gyroscope [8,12,13,40,60] or the ambient-light sensor [50]. However, these attacks suffer from a lack of knowledge when exactly a user touches the screen. Based on our attack, these sensor-based keyloggers can be improved as our attack allows to infer (1) the exact time when the user touches the screen, and (2) whether the user touches the soft-keyboard or any other region of the display.

Our attacks only require the user to install a malicious app on the smartphone. However, as shown by Oren et al. [43], *Prime+Probe* attacks can even be performed from within browser sandboxes through remote websites using JavaScript on Intel platforms. Gruss et al. [17] showed that JavaScript timing measurements in web browsers on ARM-based smartphones achieve a comparable accuracy as on Intel platforms. Thus, it seems likely that *Prime+Probe* through a website works on ARM-based smartphones as well. We expect that such attacks will be demonstrated in future work. The possibility of attacking millions of users shifts the focus of cache attacks to a new range of potential malicious applications.

In our experiments with the predecessor of ART, the Dalvik VM, we found that the just-in-time compilation effectively prevents *Evict+Reload* and *Flush+Reload* attacks. The just-in-time compiled code is not shared and thus the requirements for these two attacks are not met. However, *Prime+Probe* attacks work on ART binaries and just-in-time compiled Dalvik VM code likewise.

6 Attack on Cryptographic Algorithms

In this section we show how *Flush+Reload*, *Evict+Reload*, and *Prime+Probe* can be used to attack AES T-table implementations that are still in use on Android devices. Furthermore, we demonstrate the possibility to infer activities within the ARM TrustZone by observing the cache activity using *Prime+Probe*. We perform all attacks cross-core and in a synchronized setting, *i.e.*, the attacker triggers the execution of cryptographic algorithms by the victim process. Although more sophisticated attacks are possible, our goal is to demonstrate that our work enables practical cache attacks on smartphones.

6.1 AES T-Table Attacks

Many cache attacks against AES T-table implementations have been demonstrated and appropriate countermeasures have already been proposed. Among these countermeasures are, e.g., so-called bit-sliced implementations [30, 33, 47]. Furthermore, Intel addressed the problem by adding dedicated instructions for AES [21] and ARM also follows the same direction with the ARMv8 instruction set [5]. However, our investigations showed that Bouncy Castle, a crypto library widely used in Android apps such as the WhatsApp messenger [3], still uses a T-table implementation. Moreover, the OpenSSL library, which is the default crypto provider on recent Android versions, uses T-table implementations until version 1.0.1.[3] This version is still officially supported and commonly used on Android devices, e.g., the Alcatel One Touch Pop 2. T-tables contain the precomputed AES round transformations, allowing to perform encryptions and decryptions by simple XOR operations. For instance, let p_i denote the plaintext bytes, k_i the initial key bytes, and $s_i = p_i \oplus k_i$ the initial state bytes. The initial state bytes are used to retrieve precomputed T-table elements for the next round. If an attacker knows a plaintext byte p_i and the accessed element of the T-table, it is possible to recover the key bytes $k_i = s_i \oplus p_i$. However, it is only possible to derive the upper 4 bits of k_i through our cache attack on a device with a cache line size of 64 bytes. This way, the attacker can learn 64 key bits. In second-round and last-round attacks the key space can be reduced further. For details about the basic attack strategy we refer to the work of Osvik et al. [44, 55]. Although we successfully mounted an *Evict+Reload* attack on the Alcatel One Touch Pop 2 against the OpenSSL AES implementation, we do not provide further insights as we are more interested to perform the first cache attack on a Java implementation.

Attack on Bouncy Castle. Bouncy Castle is implemented in Java and provides various cryptographic primitives including AES. As Bouncy Castle 1.5 still employs AES T-table implementations by default, all Android devices that use this version are vulnerable to our presented

[3] Later versions use a bit-sliced implementation if ARM NEON is available or dedicated AES instructions if ARMv8-A instructions are available. Otherwise, a T-table implementation is used. This is also the case for Google's BoringSSL library.

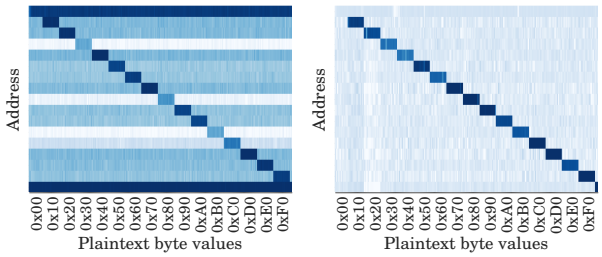

Figure 11: Attack on Bouncy Castle's AES using *Evict+Reload* on the Alcatel One Touch Pop 2 (left) and *Flush+Reload* on the Samsung Galaxy S6 (right).

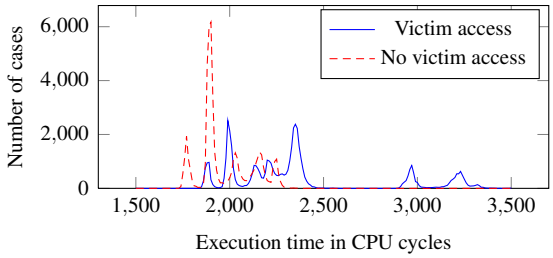

Figure 12: Histogram of *Prime+Probe* timings depending on whether the victim accesses congruent memory on the ARM Cortex-A53.

attack. To the best of our knowledge, we are the first to show an attack on a Java implementation.

During the initialization of Bouncy Castle, the T-tables are copied to a local private memory area. Therefore, these copies are not shared among different processes. Nevertheless, we demonstrate that *Flush+Reload* and *Evict+Reload* are efficient attacks on such an implementation if shared memory is available. Further, we demonstrate a cross-core *Prime+Probe* attack without shared memory that is applicable in a real-world scenario.

Figure 11 shows a template matrix of the first T-table for all 256 values for plaintext byte p_0 and a key that is fixed to 0 while the remaining plaintext bytes are random. These plots reveal the upper 4 key bits of k_0 [44, 52]. Thus, in our case the key space is reduced to 64 bits after 256–512 encryptions. We consider a first-round attack only, because we aim to demonstrate the applicability of these attacks on ARM-based mobile devices. However, full-key recovery is possible with the same techniques by considering more sophisticated attacks targeting different rounds [48, 55], even for asynchronous attackers [23, 27].

We can exploit the fact that the T-tables are placed on a different boundary every time the process is started. By restarting the victim application we can obtain arbitrary disalignments of T-tables. Disaligned T-tables allow to reduce the key space to 20 bits on average and for specific disalignments even full-key recovery without a single brute-force computation is possible [52, 54]. We observed not a single case where the T-tables were aligned. Based on the first-round attack matrix in Figure 11, the expected number of encryptions until a key byte is identified is $1.81 \cdot 128$. Thus, full key recovery is possible after $1.81 \cdot 128 \cdot 16 = 3\,707$ encryptions by monitoring a single address during each encryption.

Real-world cross-core attack on Bouncy Castle. If the attacker has no way to share a targeted memory region with the victim, *Prime+Probe* instead of *Evict+Reload* or *Flush+Reload* can be used. This is the case for dynamically generated data or private memory of another process. Figure 12 shows the *Prime+Probe* histogram for cache hits and cache misses. We observe a higher execution time if the victim accesses a congruent memory location. Thus, *Prime+Probe* can be used for a real-world cross-core attack on Bouncy Castle and also allows to exploit disaligned T-tables as mentioned above.

In a preprocessing step, the attacker identifies the cache sets to be attacked by performing random encryptions and searching for active cache sets. Recall that the cache set (index) is derived directly from the physical address on ARM, *i.e.*, the lowest n bits determine the offset within a 2^n-byte cache line and the next s bits determine one of the 2^s cache sets. Thus, we only have to find a few cache sets where a T-table maps to in order to identify all cache sets required for the attack. On x86 the replacement policy facilitates this attack and allows even to deduce the number of ways that have been replaced in a specific cache set [44]. On ARM the random replacement policy makes *Prime+Probe* more difficult as cache lines are replaced in a less predictable way. To launch a *Prime+Probe* attack, we apply the eviction strategy and the crafted reaccess patterns we described in Section 3.2.

Figure 13 shows an excerpt of the cache template matrix resulting from a *Prime+Probe* attack on one T-table. For each combination of plaintext byte and offset we performed 100 000 encryptions for illustration purposes. We only need to monitor a single address to obtain the upper 4 bits of s_i and, thus, the upper 4 bits of $k_i = s_i \oplus p_i$. Compared to the *Evict+Reload* attack from the previous section, *Prime+Probe* requires 3 times as many measurements to achieve the same accuracy. Nevertheless, our results show that an attacker can run *Prime+Probe* attacks on ARM CPUs just as on Intel CPUs.

6.2 Spy on TrustZone Code Execution

The ARM TrustZone is a hardware-based security technology built into ARM CPUs to provide a secure execution environment [5]. This trusted execution environ-

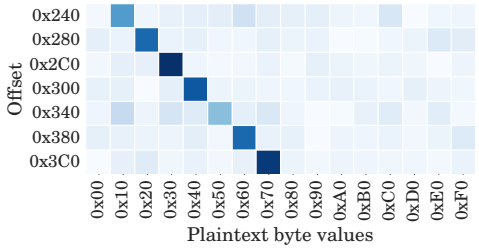

Figure 13: Excerpt of the attack on Bouncy Castle's AES using *Prime+Probe*.

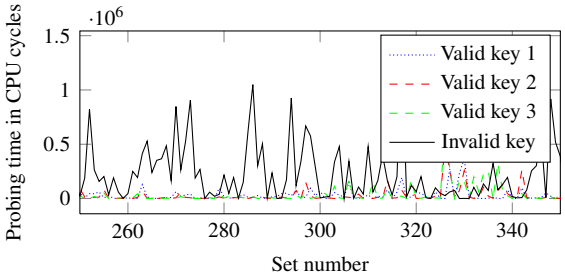

Figure 14: Mean squared error between the average *Prime+Probe* timings of valid keys and invalid keys on the Alcatel One Touch Pop 2.

ment is isolated from the *normal world* using hardware support. The TrustZone is used, e.g., as a hardware-backed credential store, to emulate secure elements for payment applications, digital rights management as well as verified boot and kernel integrity measurements. The services are provided by so-called trustlets, *i.e.*, applications that run in the secure world.

Since the secure monitor can only be called from the supervisor context, the kernel provides an interface for the userspace to interact with the TrustZone. On the Alcatel One Touch Pop 2, the TrustZone is accessible through a device driver called QSEECOM (Qualcomm Secure Execution Environment Communication) and a library `libQSEEComAPI.so`. The key master trustlet on the Alcatel One Touch Pop 2 provides an interface to generate hardware-backed RSA keys, which can then be used inside the TrustZone to sign and verify signatures.

Our observations showed that a *Prime+Probe* attack on the TrustZone is not much different from a *Prime+Probe* attack on any application in the normal world. However, as we do not have access to the source code of the TrustZone OS or any trustlet, we only conduct simple attacks.[4] We show that *Prime+Probe* can be used to distinguish whether a provided key is valid or not. While this might also be observable through the overall execution time, we demonstrate that the TrustZone isolation does not protect against cache attacks from the normal world and any trustlet can be attacked.

We evaluated cache profiles for multiple valid as well as invalid keys. Figure 14 shows the mean squared error over two runs for different valid keys and one invalid key compared to the average of valid keys. We performed *Prime+Probe* before and after the invocation of the corresponding trustlet, *i.e.*, prime before the invocation and probe afterwards. We clearly see a difference in some sets (cache sets 250–320) that are used during the signature generation using a valid key. These cache profiles are reproducible and can be used to distinguish whether a valid or an invalid key has been used in the TrustZone. Thus, the secure world leaks information to the non-secure world.

On the Samsung Galaxy S6, the TrustZone flushes the cache when entering or leaving the trusted world. However, by performing a *Prime+Probe* attack in parallel, *i.e.*, multiple times while the trustlet performs the corresponding computations, the same attack can be mounted.

7 Countermeasures

Although our attacks exploit hardware weaknesses, software-based countermeasures could impede such attacks. Indeed, we use unprotected access to system information that is available on all Android versions.

As we have shown, the operating system cannot prevent access to timing information. However, other information supplied by the operating system that facilitates these attacks could be restricted. For instance, we use `/proc/pid/` to retrieve information about any other process on the device, e.g., `/proc/pid/pagemap` is used to resolve virtual addresses to physical addresses. Even though access to `/proc/pid/pagemap` and `/proc/self/pagemap` has been restricted in Linux in early 2015, the Android kernel still allows access to these resources. Given the immediately applicable attacks we presented, we stress the urgency to merge the corresponding patches into the Android kernel. Furthermore, we use `/proc/pid/maps` to determine shared objects that are mapped into the address space of a victim. Restricting access to procfs to specific privileges or permissions would make attacks harder. We recommend this for both the Linux kernel as well as Android.

We also exploit the fact that access to shared libraries as well as `dex` and `art` optimized program binaries is only partially restricted on the file system level. While we cannot retrieve a directory listing of `/data/dalvik-cache/`, all files are readable for any process or Android application. We recommend to allow read access to these files to their respective owner ex-

[4] More sophisticated attacks would be possible by reverse engineering these trustlets.

clusively to prevent *Evict+Reload*, *Flush+Reload*, and *Flush+Flush* attacks through these shared files.

In order to prevent cache attacks against AES T-tables, hardware instructions should be used. If this is not an option, a software-only bit-sliced implementation must be employed, especially when disalignment is possible, as it is the case in Java. Since OpenSSL 1.0.2 a bit-sliced implementation is available for devices capable of the ARM NEON instruction set and dedicated AES instructions are used on ARMv8-A devices. Cryptographic algorithms can also be protected using cache partitioning [35]. However, cache partitioning comes with a performance impact and it can not prevent all attacks, as the number of cache partitions is limited.

We responsibly disclosed our attacks and the proposed countermeasures to Google and other development groups prior to the publication of our attacks. Google has applied upstream patches preventing access to `/proc/pid/pagemap` in early 2016 and recommended installing the security update in March 2016 [16].

8 Conclusion

In this work we demonstrated the most powerful cross-core cache attacks *Prime+Probe*, *Flush+Reload*, *Evict+Reload*, and *Flush+Flush* on default configured unmodified Android smartphones. Furthermore, these attacks do not require any permission or privileges. In order to enable these attacks in real-world scenarios, we have systematically solved all challenges that prevented highly accurate cache attacks on ARM so far. Our attacks are the first cross-core and cross-CPU attacks on ARM CPUs. Furthermore, our attack techniques provide a high resolution and a high accuracy, which allows monitoring singular events such as touch and swipe actions on the screen, touch actions on the soft-keyboard, and inter-keystroke timings. In addition, we show that efficient state-of-the-art key-recovery attacks can be mounted against the default AES implementation that is part of the Java Bouncy Castle crypto provider and that cache activity in the ARM TrustZone can be monitored from the normal world.

The presented example attacks are by no means exhaustive and launching our proposed attack against other libraries and apps will reveal numerous further exploitable information leaks. Our attacks are applicable to hundreds of millions of today's off-the-shelf smartphones as they all have very similar if not identical hardware. This is especially daunting since smartphones have become the most important personal computing devices and our techniques significantly broaden the scope and impact of cache attacks.

Acknowledgment

We would like to thank our anonymous reviewers for their valuable comments and suggestions.

Supported by the EU Horizon 2020 programme under GA No. 644052 (HECTOR), the EU FP7 programme under GA No. 610436 (MATTHEW), and the Austrian Research Promotion Agency (FFG) under grant number 845579 (MEMSEC).

References

[1] Armageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, Aug. 2016), USENIX Association.

[2] ANDROID OPEN SOURCE PROJECT. Configuring ART. `https://source.android.com/devices/tech/dalvik/configure.html`, Nov. 2015. Retrieved on November 10, 2015.

[3] APPTORNADO. AppBrain - Android library statistics - Spongy Castle - Bouncy Castle for Android. `http://www.appbrain.com/stats/libraries/details/spongycastle/spongy-castle-bouncy-castle-for-android`, June 2016. Retrieved on June 6, 2016.

[4] ARM LIMITED. *ARM Architecture Reference Manual. ARMv7-A and ARMv7-R edition*. ARM Limited, 2012.

[5] ARM LIMITED. *ARM Architecture Reference Manual ARMv8*. ARM Limited, 2013.

[6] ARM LIMITED. *ARM Cortex-A57 MPCore Processor Technical Reference Manual r1p0*. ARM Limited, 2013.

[7] ARM LIMITED. *ARM Cortex-A53 MPCore Processor Technical Reference Manual r0p3*. ARM Limited, 2014.

[8] AVIV, A. J., SAPP, B., BLAZE, M., AND SMITH, J. M. Practicality of Accelerometer Side Channels on Smartphones. In *Annual Computer Security Applications Conference – ACSAC* (2012), ACM, pp. 41–50.

[9] BENGER, N., VAN DE POL, J., SMART, N. P., AND YAROM, Y. "Ooh Aah... Just a Little Bit" : A Small Amount of Side Channel Can Go a Long Way. In *Cryptographic Hardware and Embedded Systems – CHES* (2014), vol. 8731 of *LNCS*, Springer, pp. 75–92.

[10] BERNSTEIN, D. J. Cache-Timing Attacks on AES, 2004. URL: `http://cr.yp.to/papers.html#cachetiming`.

[11] BOGDANOV, A., EISENBARTH, T., PAAR, C., AND WIENECKE, M. Differential Cache-Collision Timing Attacks on AES with Applications to Embedded CPUs. In *Topics in Cryptology – CT-RSA* (2010), vol. 5985 of *LNCS*, Springer, pp. 235–251.

[12] CAI, L., AND CHEN, H. TouchLogger: Inferring Keystrokes on Touch Screen from Smartphone Motion. In *USENIX Workshop on Hot Topics in Security – HotSec* (2011), USENIX Association.

[13] CAI, L., AND CHEN, H. On the Practicality of Motion Based Keystroke Inference Attack. In *Trust and Trustworthy Computing – TRUST* (2012), vol. 7344 of *LNCS*, Springer, pp. 273–290.

[14] GALLAIS, J., KIZHVATOV, I., AND TUNSTALL, M. Improved Trace-Driven Cache-Collision Attacks against Embedded AES Implementations. In *Workshop on Information Security Applications – WISA* (2010), vol. 6513 of *LNCS*, Springer, pp. 243–257.

[15] GALLAIS, J.-F., AND KIZHVATOV, I. Error-Tolerance in Trace-Driven Cache Collision Attacks. In *COSADE* (2011), pp. 222–232.

[16] GOOGLE INC. Nexus Security Bulletin - March 2016. https://source.android.com/security/bulletin/2016-03-01.html, Mar. 2016. Retrieved on June 6, 2016.

[17] GRUSS, D., BIDNER, D., AND MANGARD, S. Practical Memory Deduplication Attacks in Sandboxed Javascript. In *European Symposium on Research – ESORICS* (2015), vol. 9326 of *LNCS*, Springer, pp. 108–122.

[18] GRUSS, D., MAURICE, C., AND MANGARD, S. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *DIMVA'16* (2016).

[19] GRUSS, D., MAURICE, C., WAGNER, K., AND MANGARD, S. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA'16* (2016).

[20] GRUSS, D., SPREITZER, R., AND MANGARD, S. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium* (2015), USENIX Association, pp. 897–912.

[21] GUERON, S. White Paper: Intel Advanced Encryption Standard (AES) Instructions Set, 2010. URL: https://software.intel.com/file/24917.

[22] GULLASCH, D., BANGERTER, E., AND KRENN, S. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *IEEE Symposium on Security and Privacy – S&P* (2011), IEEE Computer Society, pp. 490–505.

[23] GÜLMEZOGLU, B., INCI, M. S., APECECHEA, G. I., EISENBARTH, T., AND SUNAR, B. A Faster and More Realistic Flush+Reload Attack on AES. In *Constructive Side-Channel Analysis and Secure Design – COSADE* (2015), vol. 9064 of *LNCS*, Springer, pp. 111–126.

[24] HUND, R., WILLEMS, C., AND HOLZ, T. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *IEEE Symposium on Security and Privacy – S&P* (2013), IEEE, pp. 191–205.

[25] IRAZOQUI, G., EISENBARTH, T., AND SUNAR, B. S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing – and its Application to AES. In *IEEE Symposium on Security and Privacy – S&P* (2015), IEEE Computer Society.

[26] IRAZOQUI, G., EISENBARTH, T., AND SUNAR, B. Cross Processor Cache Attacks. In *ACM Computer and Communications Security – ASIACCS* (2016), ACM, pp. 353–364.

[27] IRAZOQUI, G., INCI, M. S., EISENBARTH, T., AND SUNAR, B. Wait a Minute! A fast, Cross-VM Attack on AES. In *Research in Attacks, Intrusions and Defenses Symposium – RAID* (2014), vol. 8688 of *LNCS*, Springer, pp. 299–319.

[28] IRAZOQUI, G., INCI, M. S., EISENBARTH, T., AND SUNAR, B. Know Thy Neighbor: Crypto Library Detection in Cloud. *Privacy Enhancing Technologies 1*, 1 (2015), 25–40.

[29] IRAZOQUI, G., INCI, M. S., EISENBARTH, T., AND SUNAR, B. Lucky 13 Strikes Back. In *ACM Computer and Communications Security – ASIACCS* (2015), ACM, pp. 85–96.

[30] KÄSPER, E., AND SCHWABE, P. Faster and Timing-Attack Resistant AES-GCM. In *Cryptographic Hardware and Embedded Systems – CHES* (2009), vol. 5747 of *LNCS*, Springer, pp. 1–17.

[31] KELSEY, J., SCHNEIER, B., WAGNER, D., AND HALL, C. Side Channel Cryptanalysis of Product Ciphers. *Journal of Computer Security 8*, 2/3 (2000), 141–158.

[32] KOCHER, P. C. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology – CRYPTO* (1996), vol. 1109 of *LNCS*, Springer, pp. 104–113.

[33] KÖNIGHOFER, R. A Fast and Cache-Timing Resistant Implementation of the AES. In *Topics in Cryptology – CT-RSA* (2008), vol. 4964 of *LNCS*, Springer, pp. 187–202.

[34] LAL SHIMPI, ANANDTECH. 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, Dec. 2013. Retrieved on November 10, 2015.

[35] LIU, F., GE, Q., YAROM, Y., MCKEEN, F., ROZAS, C. V., HEISER, G., AND LEE, R. B. CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing. In *IEEE International Symposium on High Performance Computer Architecture – HPCA* (2016), IEEE Computer Society, pp. 406–418.

[36] LIU, F., YAROM, Y., GE, Q., HEISER, G., AND LEE, R. B. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE Symposium on Security and Privacy – SP* (2015), IEEE Computer Society, pp. 605–622.

[37] MARFORIO, C., RITZDORF, H., FRANCILLON, A., AND CAPKUN, S. Analysis of the Communication Between Colluding Applications on Modern Smartphones. In *Annual Computer Security Applications Conference – ACSAC* (2012), ACM, pp. 51–60.

[38] MAURICE, C., NEUMANN, C., HEEN, O., AND FRANCILLON, A. C5: Cross-Cores Cache Covert Channel. In *Detection of Intrusions and Malware, and Vulnerability Assessment – DIMVA* (2015), vol. 9148 of *LNCS*, Springer, pp. 46–64.

[39] MAURICE, C., SCOUARNEC, N. L., NEUMANN, C., HEEN, O., AND FRANCILLON, A. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In *Research in Attacks, Intrusions, and Defenses – RAID* (2015), vol. 9404 of *LNCS*, Springer, pp. 48–65.

[40] MILUZZO, E., VARSHAVSKY, A., BALAKRISHNAN, S., AND CHOUDHURY, R. R. Tapprints: Your Finger Taps Have Fingerprints. In *Mobile Systems, Applications, and Services – MobiSys* (2012), ACM, pp. 323–336.

[41] NEVE, M. *Cache-based Vulnerabilities and SPAM Analysis*. PhD thesis, UCL, 2006.

[42] NEVE, M., SEIFERT, J., AND WANG, Z. A Refined Look at Bernstein's AES Side-Channel Analysis. In *ACM Computer and Communications Security – ASIACCS* (2006), ACM, p. 369.

[43] OREN, Y., KEMERLIS, V. P., SETHUMADHAVAN, S., AND KEROMYTIS, A. D. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In *Conference on Computer and Communications Security – CCS* (2015), ACM, pp. 1406–1418.

[44] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache Attacks and Countermeasures: The Case of AES. In *Topics in Cryptology – CT-RSA* (2006), vol. 3860 of *LNCS*, Springer, pp. 1–20.

[45] PAGE, D. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. *IACR Cryptology ePrint Archive 2002/169*.

[46] PERCIVAL, C. Cache Missing for Fun and Profit, 2005. URL: http://daemonology.net/hyperthreading-considered-harmful/.

[47] REBEIRO, C., SELVAKUMAR, A. D., AND DEVI, A. S. L. Bitslice Implementation of AES. In *Cryptology and Network Security – CANS* (2006), vol. 4301 of *LNCS*, Springer, pp. 203–212.

[48] SAVAS, E., AND YILMAZ, C. A Generic Method for the Analysis of a Class of Cache Attacks: A Case Study for AES. *Comput. J. 58*, 10 (2015), 2716–2737.

[49] SCHLEGEL, R., ZHANG, K., ZHOU, X., INTWALA, M., KAPADIA, A., AND WANG, X. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In *Network and Distributed System Security Symposium – NDSS* (2011), The Internet Society.

[50] SPREITZER, R. PIN Skimming: Exploiting the Ambient-Light Sensor in Mobile Devices. In *Security and Privacy in Smartphones & Mobile Devices – SPSM@CCS* (2014), ACM, pp. 51–62.

[51] SPREITZER, R., AND GÉRARD, B. Towards More Practical Time-Driven Cache Attacks. In *Information Security Theory and Practice – WISTP* (2014), vol. 8501 of *LNCS*, Springer, pp. 24–39.

[52] SPREITZER, R., AND PLOS, T. Cache-Access Pattern Attack on Disaligned AES T-Tables. In *Constructive Side-Channel Analysis and Secure Design – COSADE* (2013), vol. 7864 of *LNCS*, Springer, pp. 200–214.

[53] SPREITZER, R., AND PLOS, T. On the Applicability of Time-Driven Cache Attacks on Mobile Devices. In *Network and System Security – NSS* (2013), vol. 7873 of *LNCS*, Springer, pp. 656–662.

[54] TAKAHASHI, J., FUKUNAGA, T., AOKI, K., AND FUJI, H. Highly Accurate Key Extraction Method for Access-Driven Cache Attacks Using Correlation Coefficient. In *Australasian Conference Information Security and Privacy – ACISP* (2013), vol. 7959 of *LNCS*, Springer, pp. 286–301.

[55] TROMER, E., OSVIK, D. A., AND SHAMIR, A. Efficient Cache Attacks on AES, and Countermeasures. *Journal Cryptology 23*, 1 (2010), 37–71.

[56] TSUNOO, Y., SAITO, T., SUZAKI, T., SHIGERI, M., AND MIYAUCHI, H. Cryptanalysis of DES Implemented on Computers with Cache. In *Cryptographic Hardware and Embedded Systems – CHES* (2003), vol. 2779 of *LNCS*, Springer, pp. 62–76.

[57] VAN DE POL, J., SMART, N. P., AND YAROM, Y. Just a Little Bit More. In *Topics in Cryptology – CT-RSA* (2015), vol. 9048 of *LNCS*, Springer, pp. 3–21.

[58] WEISS, M., HEINZ, B., AND STUMPF, F. A Cache Timing Attack on AES in Virtualization Environments. In *Financial Cryptography and Data Security – FC* (2012), vol. 7397 of *LNCS*, Springer, pp. 314–328.

[59] WEISS, M., WEGGENMANN, B., AUGUST, M., AND SIGL, G. On Cache Timing Attacks Considering Multi-core Aspects in Virtualized Embedded Systems. In *Trusted Systems – INTRUST* (2014), vol. 9473 of *LNCS*, Springer, pp. 151–167.

[60] XU, Z., BAI, K., AND ZHU, S. TapLogger: Inferring User Inputs on Smartphone Touchscreens Using On-board Motion Sensors. In *Security and Privacy in Wireless and Mobile Networks – WISEC* (2012), ACM, pp. 113–124.

[61] YAROM, Y., AND FALKNER, K. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium* (2014), USENIX Association, pp. 719–732.

[62] ZHANG, K., AND WANG, X. Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems. In *USENIX Security Symposium* (2009), USENIX Association, pp. 17–32.