Profiling encryption algorithms using ARM-based cache eviction attacks

Jan-Jaap Korpershoek
jjkorpershoek96@gmail.com

September 14, 2020

In a forensic setting breaking the encryption of Android phones of suspects is very relevant to obtain evidence. Cache eviction attacks are a method that can be used for this. Those attacks can be used to profile an application, i.e. obtain information on timings within the execution. These timings could be used as the basis for further attacks, such as fault-injection attacks or to obtain information about the execution flow of the victim program. We show FLUSH+RELOAD profiling on OpenSSL AES, and PRIME+PROBE profiling on the RSA implementation used to verify ARM TrustZone applications (trustlets). These attacks are executed in a bare-metal environment, which leads to reliable results due to the lack of operating system interference. In addition to the attacks we provide an overview of the state of the art of ARM cache side-channel attacks.

1. Acknowledgements

I would like to thank Jeffrey Rongen and Nico van Heijningen from Netherlands Forensic Institute for their advice during this project and for the initial bare-metal implementation. Lipp et al. [1] for the practical code samples and the libflush library. Tang et al. [2] for the clear documentation of the reverse engineering work on the TrustZone RSA implementation, and the provided github repository [3].

Contents

2. Introduction

Android smartphones have supported full-disk encryption (FDE) from version 4.4, which was released in 2014 and have supported file-based encryption (FBE) from android 7.0 which was released in 2016 [5]. As a result, nowadays, many devices of criminal suspects are encrypted. Therefore, attacks that can break the encryption of smartphones are very useful in a forensic setting as decrypted phones lead to more evidence.

Cache eviction attacks are a useful method for this. A multitude of cache attacks have been published for the x86 platform [6, 7, 8, 9, 10, 11]. For ARM several attacks have been published as well [12, 1, 13, 14, 15, 16, 17, 2, 18]. Lyu and Mishra [19] have provided a survey of cache side-channel attacks and Ge et al. [20] have provided a survey of micro-architectural attacks. This work, in contrast to the previous surveys, aims to provide in-depth summaries of several ARM cache attacks.

Tang et al. [2] describe the CLKSCREW attack, which can be used to obtain an AES secret key used within TrustZone. They do not use runtime profiling to find in which state the implementation it is at the moment of attack, since the timings of their textbook implementation can be predicted. To attack a more complex AES implementation, it could be profiled to find the timings of different states in the program in order to induce a fault at the right time.

Tang et al. [2] also execute a CLKSCREW attack on RSA. For that attack they profile the implementation using instruction-based cache attacks. However, the profiling implementation of Tang et al. is not published.

We show methods to profile an AES implementation at runtime in order to find the exact timing to use in a fault-injection attack. In addition, we describe a TrustZone RSA side-channel. Our experiments are run in a bare-metal environment, which means that we run without any kernel or user space interference, which minimizes the amount of noise in the measurements. The experiments are performed on the Motorola Nexus 6 XT1100 (codename shamu) with firmware version ngi77b [21].

Threat model We can choose a strong adversary because we are modelling a forensic setting where the attacker can use any means to break the security of the device. The adversary has full access to the device and can execute a custom kernel. The adversary does not have access to TrustZone keys, so cannot run code in TrustZone except by using an exploit.

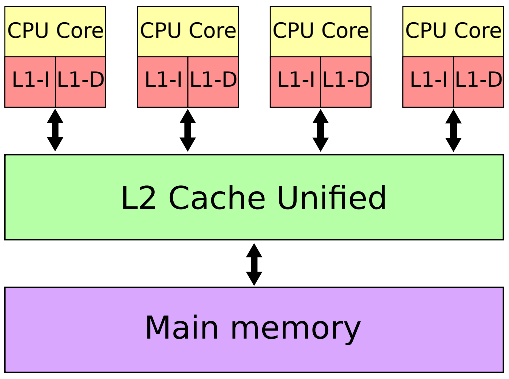
2.1. Contributions

The contributions of this work are:

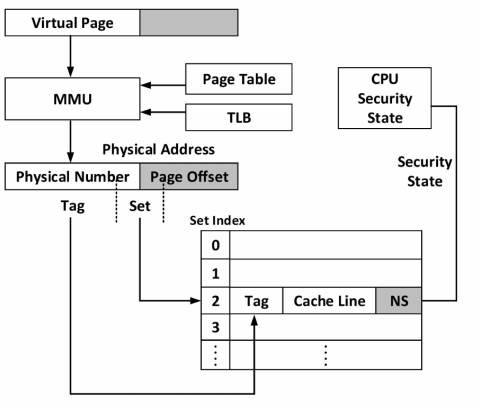
- Providing a survey of cache-based attacks in general and specifically on ARM (section 3).
- Profiling an OpenSSL AES implementation using FLUSH+RELOAD (section 7.2).
- Reproducing the TrustZone RSA profiling of Tang et al. [2] (section 8.2). This has also been reproduced by Qiu et al. [18], but both their descriptions are very minimal. We reproduce part of their profiling and highlight difficulties and unknowns.
- Performing cache side-channel attacks in a bare-metal environment, which to our knowledge has not been done in previous research. Green et al. [15] perform tests in a bare-metal environment, but no actual attack.
- Providing plain bare-metal implementations of several features of the kernel for the Nexus 6, such as enabling the memory management unit (MMU), symmetric multiprocessing (SMP) and obtaining hardware randomness [22].
- Showing different results for the L2 (level 2) inclusiveness test by X. Zhang et al. [14] (section 6.5.2).
- Providing practical implementations of profiling using cache side-channels [22].
- Providing information about the caching behaviour of the APQ8084 processor of the Nexus 6 (section 3.6).

2.2. Outline

In sections 3 and 4 we describe the background and related work. In section 5 we describe the bare-metal environment which we use for the experiments. In section 6 we describe and test primitives necessary to perform our side-channel attacks. In sections 7 and 8 we describe our AES and RSA profiling attacks. We finalize in sections 9 to 11 with our discussion, conclusion and future work.



(a) CPU cache architecture



(b) Cache addressing [13]

Figure 1: Cache architecture and addressing

3. Background

3.1. CPU cache

CPU caches are a system used for speeding up memory I/O. If an address has to be read from memory, first the cache is queried, when the cache contains the desired address, it can be used without doing the slower memory lookup. When the address is not in the cache, it is first loaded from memory into the cache (replacing an existing entry chosen by the replacement algorithm) and the value is used by the CPU after that. The next time the same address is looked up, it is returned faster since it can be obtained from the cache. For writing to memory there are different policies, write-through and write-back. With write-through, the data is written to the cache and the storage simultaneously, while with write-back the data is written to the cache first, and to the storage later, e.g. when the address is evicted. Usually caches are arranged in multiple layers of increasingly larger, but slower caches, typically there are two or three layers. The lower-level caches are faster, smaller and closer to the processor core. The cache layers are typically named L1 (level 1), L2 (level 2), etc, where a lower number indicates a lower level. The highest cache level is often called the last-level cache (LLC) and is often shared between CPUs. The L1 cache is sometimes split into a data cache (L1-D) and an instruction cache (L1-I). An example of a cache hierarchy can be seen in fig. 1a.

Caches today are usually set-associative, i.e. organized as S sets of W lines each, called a W-way set-associative cache. A specific cache line number across all sets is called a *cache way*. A cache line address has three parts. The offset (the lower bits of the address) locates the value inside a cached block. The index (the middle bits of the address) locates the used cache set.

The tag (the higher bits of the address) is matched against the tag of the W lines in the set to identify if one of the cache lines is a cache hit [12].

Inclusiveness 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. Inclusiveness of a cache can be different for data values or instruction values.

Cache coherence protocols Different cores make sure their caches contain the same values for the same memory blocks using cache coherence protocols. The cores communicate by messages over a shared bus or by using a directory to determine which core to ask for the desired data. Using cache coherence protocols, a core can obtain a value from a different core with a much smaller latency than main memory. For more information on cache coherence protocols see [23].

Eviction If an entry is removed from the cache, this is called eviction. Eviction can happen by performing a cache flush or by loading other values into the same cache set. A flush instruction will invalidate a specified part of the cache, i.e. a line, a set or the whole cache. Values in the same cache set compete for space, so if new values are cached in the same set, older values are removed based on the effective replacement policy. These two methods are called FLUSH and EVICT and are used to build cache side channels.

3.2. ARM architecture

ARM uses two different sets of instructions, the 32-bit (or 64-bit) ARM instructions and the 16-bit (or 32-bit) Thumb instructions. The processor can switch between the instruction sets by setting the T-bit in the current program status register (CPSR), this bit is set based on the least-significant bit of the program counter (PC) [24].

3.2.1. ARM cache

According to Lipp et al. [1] except for ARMv8-A CPUs, ARM processors do not support a flush instruction. In addition, most LLCs are not inclusive on ARM.

The caches of the Krait 450 cores of the Nexus 6 are data-inclusive and instruction-inclusive however [14]. On the Nexus 6 and many other ARM devices, the LLC is the L2 cache.

Addressing L2 caches are usually physically indexed, physically tagged (PIPT) [1]. Programs, however run in virtual address space. When the processor needs to fetch a value in memory, it passes the virtual address to the memory management unit (MMU). The MMU first attempts to retrieve the virtual address to physical address translation from the translation lookaside buffer (TLB). If there is no hit, the MMU parses the page table for the translation and places it inside the TLB. Parsing of the page table is often referred to as a page walk. Once the physical address is obtained, the MMU goes through the system memory hierarchy to retrieve the value at that address. fig. 1b illustrates the process of retrieving data from memory.

AutoLock AutoLock is an undocumented feature of ARM processors that has been described in detail by Green et al. [15]. AutoLock is a performance enhancement that inhibits cross-core LLC evictions. It is only defined in an inclusive-cache setting and works by setting an inclusion bit on L2 cache lines if the lines are allocated in the L1 cache. If this bit is set, the cache line cannot be evicted. If all lower-level copies of the cache-line have been evicted, the inclusion bit is reset, and the L2 cache line can also be evicted.

Trying to evict a cache set from the L2 cache will thus not always work, so the effect of AutoLock is similar to that of a non-inclusive cache. However, cache maintenance operations, such as a flush instruction, override AutoLock.

AutoLock is not implemented on the Krait 450 cores of the Nexus 6.

3.2.2. TrustZone

ARM TrustZone is a hardware-based security technology built into ARM CPUs to provide a secure execution environment without using a separate CPU. This trusted execution environment (secure world) is isolated from the normal world using hardware support. 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. Such services are provided by so-called trustlets, i.e., applications that run in the secure world. The world that the processor operates in is indicated by the 33rd processor bit, also known as the *non-secure* bit, which value can be read as the first bit of the secure configuration register (SCR) [25]. On some (older) devices the cache is flushed when entering or leaving the secure world [1]. On newer devices a *non-secure* bit is used to indicate which cache lines are used by TrustZone and which are used by normal code [13]. So, the secure world does not use the cache lines of the normal world and vice versa. With the old system, where a cache flush is used, cache attacks on TrustZone are not very different from cache attacks on normal code [1]. With the new system however, novel techniques are required to leak information. Since the cache is no longer flushed on a world switch, the

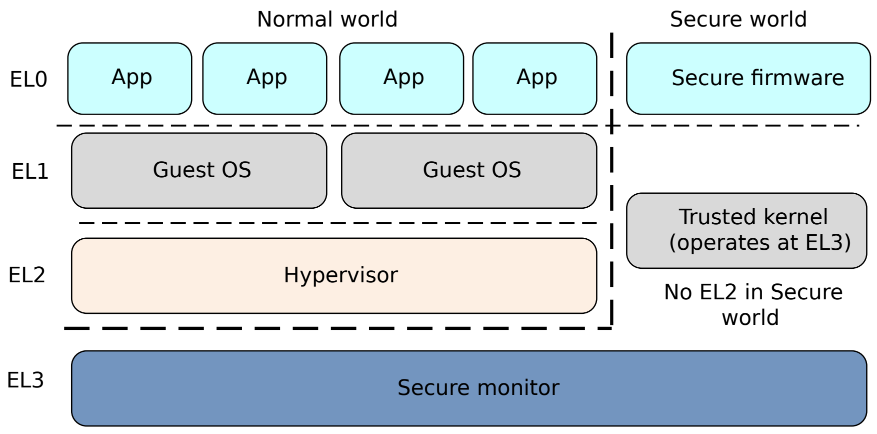


Figure 2: ARMv8 Exception levels [26]

cache lines of the normal world and secure world can evict each other (cache contention). N. Zhang et al. [13] have shown that this can be exploited.

The normal world can communicate with the secure world via privileged secure monitor call (SMC) instructions. These instructions are handled by the secure monitor that runs on a higher exception level than both the normal world and the secure world OSes, see fig. 2 for an overview of the exception levels. Exceptions and interrupts can also be handled by the secure monitor. The separation of memory in normal world memory and secure world memory is done by the TrustZone address space controller (TZASC) and the TrustZone memory adapter (TZMA), this ensures that the secure world can access normal world memory, but not the other way around. The TZASC and TZMA are optional components in the TrustZone specification and hence may not exist on all system on chip (SoC) implementations. TrustZone technology also allows for devices to be restricted to secure world or normal world, through the TrustZone protection controller (TZPC), which is also an optional component [25].

3.3. Cache attacks

Cache attacks can be divided in the following three categories of increasing attacker capabilities [27].

Time-driven The attacker uses the overall encryption time to leak the secret key. This is very general, but does not reduce the key space much [28].

Trace-driven The attacker can notice when the victim has a cache hit or miss.

Access-driven The attacker can notice which cache sets the victim is accessing.

The attacks described below are examples of access-driven attacks, except for PRIME+COUNT, which is trace-driven.

Flush+Reload

1. Choose an address to monitor.
2. Flush the cache line corresponding to the address.
3. Schedule the victim program.
4. Measure the time it takes to access the address.
5. A short access time indicates that the cache set was used by the victim.

This is a fast and fine-grained attack, but it requires shared memory with the victim program.

Evict+Reload Same as FLUSH+RELOAD, but replacing the flush by a series of evictions. This is useful for systems where a flush instruction is not available.

Prime+Probe

1. Occupy specific cache sets.
2. Run victim program.
3. Measure the time it takes to reload (probe) the addresses of the eviction set.
4. A long probe time indicates that the cache set was used by the victim.

This attack is slower than FLUSH+RELOAD but it does not require shared memory. Therefore, this attack can be used to attack a victim in TrustZone or across memory boundaries.

Prime+Count [17] Similar to PRIME+PROBE but instead of determining *which* cache sets are occupied it only determines *how many* cache sets are occupied. The benefit of this coarser-grained attack is that it is less sensitive to cache noise. A drawback is that more measurements are necessary to get accurate information.

Flush+Flush [29] FLUSH+FLUSH works similar to a FLUSH+RELOAD attack, but instead of reloading the cache line, it is flushed again. Benefits of this attack are that it is faster than a PRIME+PROBE or FLUSH+RELOAD attack and that it is more stealthy because no memory is accessed by the attacker. Similar to FLUSH+RELOAD, FLUSH+FLUSH requires shared-memory. If the timing of the flush instruction depends on whether the cache line is in use or not, this difference can be measured by the attacker to determine if the victim accessed the cache line. A long flush duration indicates that the cache set was used by the victim.

Evict+Time [30]

1. Measure execution time of victim program.
2. Evict a specific cache set.
3. Measure execution time of victim program again.
4. A short execution time indicates that the cache set was used by the victim.

Flush The flush operation is used to clean and invalidate a specific part of a cache. There are several methods to perform a flush. Firstly, on x86 there is a dedicated flush instruction called `CLFLUSH` which flushes an address from all caches. This is a very fast method because only a single instruction is needed. On ARM there is no such flush instruction [1]. Secondly, there are several coprocessor registers, like data cache clean and invalidate by set/way (DCCISW) that can be used to clean and invalidate (flush) a specified part of the cache. These registers are only accessible from kernel space. This method is slower than a dedicated instruction because some code is necessary to flush all the required cache lines. Finally, there is the `clearcache` system call. This system call uses the coprocessor registers, but allows calls from user space. This has the drawbacks that it does not invalidate the cache on all devices and that because it is a system call, it has overhead which makes it slower than using the coprocessor registers directly.

Eviction strategies On processors that use a least recently used (LRU) replacement policy, accessing one set-congruent address per cache line is enough to evict a cache set. However, ARM processors use a pseudo-random replacement policy. Therefore, more complicated patterns of memory accesses are necessary to evict a cache set with high probability. This is called an eviction strategy. The algorithm of sliding window eviction (algorithm 1) is used for this purpose. It uses three parameters:

N The number of windows

A The number of repetitions per window

D The number of addresses per window

The algorithm accesses D set-congruent addresses A times for N windows.

Algorithm 1: Sliding window eviction by Gruss et al. [31, p. 305]

```
Data: C: A list of set-congruent addresses
for $i = 0..N-1$ do
    for $j = 0..A-1$ do
        for $k = 0..D-1$ do
            access(C[i+k]);
        end
    end
end
```

3.4. AES

AES is a commonly used symmetric encryption algorithm designed by Daemen and Rijmen [32]. The algorithm looks as shown in algorithm 2. It contains the following functions

KeyExpansion Expand the initial key into the necessary number of round keys.

SubBytes Substitute the bytes of the current plaintext using S-Boxes.

ShiftRows Shift the rows of the input matrix cyclically to the left by a specified offset.

MixColumns Perform a linear transformation on the columns of the input matrix.

AddRoundKey Add the current round key to the current input.

Algorithm 2: AES cipher

```
Data: N: number of rounds, 9 for 128-bit, 11 for 192-bit, 13 for
      256-bit.
KeyExpansion()
AddRoundKey()
for $i = 0..N-1$ do
    SubBytes()
    ShiftRows()
    MixColumns()
    AddRoundKey()
end
SubBytes()
ShiftRows()
AddRoundKey()
```

3.4.1. T-Tables

Some AES implementations like the OpenSSL implementation [33] use T-Tables to speed up the calculations. T-Tables contain precomputed AES round transformations, which can be looked up based on the bytes of the current state. This reduces the encryption and decryption calculations to simple XORs and table lookups. If the used T-Table is detected, this can be used to infer information on the key bytes that were used, since the state used for the lookup is based on an XOR between the previous state and the key bytes.

3.5. Differential fault analysis of AES using a Single Fault

Tunstall et al. [34] have conducted a differential fault analysis (DFA) on AES. They target an AES implementation which uses a loop for the rounds (iterative) in contrast to an unrolled version. A one-byte random fault is introduced in the state matrix that is the input to the 8th round of a 10-round AES encryption. The fault propagates through the state as the encryption continues. From the difference between the faulty ciphertext and the correct ciphertext equations can be derived for the relation between certain key-bytes and the fault. By guessing values for the fault in the state after the 9th round ShiftRow operation, candidates for the key-bytes can be obtained. There are four sets of equations that each yield hypotheses for 4 key-bytes. Each set yields on average 1 fitting hypothesis for each candidate for the fault. This results in 256 unique hypotheses per set of equations, so in total $256^4 = 2^{32}$ different key hypotheses. These hypotheses can be subjected to a second test, which examines the relation between the fault after the 8th round MixColumn and the difference between the correct and faulty ciphertexts. This yields another 4 sets of equations. The probability that a hypothesis satisfies these equations for a specific fault value is $(\frac{1}{2^8})^4 = \frac{1}{2^{32}}$. Therefore, since there are 256 different fault values, the total probability that a hypothesis satisfies the equations is $256 \cdot \frac{1}{2^{32}} = \frac{1}{2^{24}}$. So, after the second test there are $\frac{2^{32}}{2^{24}} = 2^8$ hypotheses left.

The attack is based on the assumption that we know the location of the fault, i.e. which byte was faulted. If we do not know the location, the analysis has to be performed for each of the 16 bytes of the state. This results in $16 \cdot 2^8 = 2^{12}$ key hypotheses and an analysis time that is 16 times as high.

3.6. Motorola Nexus 6 XT1100

3.6.1. Snapdragon 805 Processor

The Snapdragon 805 (APQ8084) [35] has four Krait 450 cores. It has an ARMv7 instruction set architecture (ISA) (32-bit) and a Thumb-2 ISA (16-

bit).

Cache layout Per core, the APQ8084 has a 16 KiB 4-way L1-D (level 1 data) and a 16 KiB 4-way L1-I (level 1 instruction) cache, with a cache line size of 64 bytes. The number of sets for the L1-D cache is 64. In addition, it has a 2 MiB 8-way unified L2 cache, with a line size of 128 bytes, which is inclusive to L1-D and L1-I [14]. The number of sets for the L2 cache is 2048. All measurements about cache size, associativity and cache line size have been obtained by interpreting the `CCSIDR` register. AutoLock is not implemented on the APQ8084 [15].

4. Related work

4.1. Last-Level Cache Side-Channel Attacks are Practical

Liu et al. [12] have executed Prime+Probe attacks on the LLC in a cross-core scenario. This was done on Intel processors. Instead of probing the whole LLC, first cache sets relevant to security-critical victim code and data are determined. This is done by monitoring one cache set at a time and looking for temporal access patterns to this cache set that are consistent with the victim performing security-critical accesses. The attacks are shown to work with secret-dependent instructions as well as secret-dependent data.

Large pages are used for the virtual to physical address translation. If a page is large enough (i.e. 2 MiB on the hardware of Liu et al.), the index bits of the cache line are within the page offset and thus are the same for the virtual and physical address. This removes the need for address translation, since the cache set can be determined directly from the virtual address. Due to the hash-based indexing on some Intel processors, Liu et al. have to take some additional steps in order to find an eviction set. This is not necessary on ARM.

An attack is shown on a square-and-multiply algorithm used in ElGamal encryption, which uses secret-dependent instructions. All lines in the LLC are monitored for a time in order to find which cache line corresponds to the square-and-multiply algorithm. This is the one that shows pulses of usage, where each pulse is a square operation. Once this line is found, the bits of the exponent (secret key) can be found by observing the intervals between squares, a long interval indicating a 1, a short interval a 0.

4.2. Armageddon

Lipp et al. [1] show that Prime+Probe, Flush+Reload, Evict+Reload, and Flush+Flush work on non-rooted ARM-based devices without any privileges. Among other attacks, an attack on a T-Table based AES implementation is done using Prime+Probe. This is done in a cross-core setting where the attacker triggers the execution of the victim process (synchronous).

Lipp et al. use several devices, but not the Nexus 6 that is used in this work.

4.2.1. Virtual to physical address conversion

To obtain the correct cache set for an address, Lipp et al. use the pagemap file (`/proc/<pid>/pagemap`), which contains mappings for virtual to physical addresses. If the physical address is known, obtaining the cache set is trivial. According to Lipp et al., the pagemap file can be accessed by an unprivileged app.

4.2.2. Cross-core eviction

Since most ARM caches are not inclusive and often cores do not share a cache, coherence protocols and L1-to-L2 transfers are exploited to make cross-core attacks work. All devices used by Lipp et al. have a shared L2 cache and employ a cache coherence protocol between cores. By choosing an efficient eviction strategy the amount of transfers between L1 and L2 cache is maximized in order to evict the caches of the other cores.

4.2.3. Accurate Timing

A good timing source is the cycle counter from the performance monitoring unit. However, this is not accessible without privileges, so other timing sources have to be used. Three options are given: an unprivileged system call, a POSIX function and a dedicated thread timer (a loop that increments a variable in another thread). Despite the delays and noise cache hits and misses can clearly be distinguished with all methods.

4.2.4. Covert channels

Lipp et al. develop high-performance covert channels using Evict+Reload, Flush+Reload and Flush+Flush. They test these covert channels by running two unprivileged applications in the background of a Samsung Galaxy S6 running Android, while the phone was mostly idle and an unrelated app is running in the foreground. Their Flush+Reload covert channel can achieve a speed of about 1 Mbit per second at an error rate of 1.10% in a cross-core scenario. Their other covert channels are slower, but still order of magnitude faster than previously existing covert channels.

4.2.5. Shared library attack

An attack is executed on the `libinput.so` library. First a template of cache usage is determined by performing an event while simultaneously measuring the number of cache hits that occur on a set of addresses using Prime+Probe. These templates are created for several events, like a press of the power button, long touch events, swipe events, short touch events and text input events. Using these template matrices Lipp et al. are able to distinguish between these events using Flush+Reload and Evict+Reload side channels.

4.2.6. AES T-Table attack

An attack on the AES implementation of the Bouncy Castle crypto library is conducted. This is done in a cross-core and synchronized setting, i.e. the attacker starts the victim. The attack follows the strategy described by Osvik et al. [30]. Only a first-round attack is performed, so only part of the key is recovered, however, full-key recovery is possible with the same techniques by targeting different rounds. The addresses of the first T-Table are monitored using Flush+Reload or Evict+Reload for each of the 256 values for the first plaintext byte and a key that is fixed to 0, while the remaining plaintext bytes are random. They observe a clear correlation between the used address of the T-Table and the plaintext byte value. This information reveals the upper 4 key bits of the first key byte. The attack can be extended in several ways to reduce the key space more.

If no shared memory on the T-Tables can be achieved, Flush+Reload and Evict+Reload are not sufficient. Lipp et al. perform the same attack using Prime+Probe.

4.2.7. TrustZone attack

A Prime+Probe attack is executed on a TrustZone implementation that flushes the cache on context switch but uses the same cache lines for the normal world and the secure world. Lipp et al. only perform a simple attack, since the source code of the TrustZone OS is not available. They observe a clear difference between the cache sets used for a correct key and for an incorrect key. This is in contrast with the implementation used in TruSpy (section 4.3).

4.3. TruSpy

N. Zhang et al. [13] perform a Prime+Probe attack on T-Table based AES implementation of OpenSSL. According to N. Zhang et al., attacks on TrustZone had not yet been shown in detail. They also, to our knowledge, show the first attack on a TrustZone implementation that uses an NS-bit (Non-Secure) to indicate which cache lines are used by TrustZone and which by the normal world. Since the cache lines are independent of each other, there is no need to flush the caches on a switch between normal and secure world. The lingering cache content can be exploited since normal and secure world cache lines can evict each other (cache contention). TruSpy is performed on a single core, so further research is necessary to extend it to a cross-core attack.

N. Zhang et al. perform two attacks. The first is done from the OS-level, i.e. as a kernel module with full permissions. The second is done from an Android app without any permissions. The attack follows five steps.

1. Identify the memory that will be used for cache priming, for this, set-congruent addresses have to be found, i.e. addresses that use the same cache set as the victim program.
2. Prime the cache.
3. Trigger execution of the victim process.
4. Probe the cache.
5. Analyse collected information to recover the secret information (e.g. cryptographic keys).

The attack was executed on a development board without a secure-world OS [16].

4.3.1. Finding congruent memory addresses

Set-congruent addresses are found by observing cache pollution for different pages, unlike previous research which used the pagemap table. This table is not available in user space. Cache pollution of a memory location means that the victim process leaves traces in the L1 cache of access to that part of the memory. If a page has significantly more pollution than average, this page is likely congruent with the encryption operation. This technique is called *statistical matching* by N. Zhang et al..

4.3.2. Accurate timing

In the OS-level attack, the cycle counter from the performance monitoring unit can be used. However, it is not available to an Android app without permissions, so a less accurate timing source has to be used. For this, system calls are used to obtain the cycle counter. This is less accurate, so it is more difficult to distinguish an L1 cache access from an L2 cache access. A suitable threshold is chosen and the variance is reduced by running the attack many times. This seems to contrast with Armageddon (section 4.2.3) where Lipp et al. [1] describe that hits and misses can clearly be distinguished. This is probably because in Armageddon, Lipp et al. [1] are referring to distinguishing a hit (cache access) from a miss (memory access), while N. Zhang et al. try to distinguish an L1 hit from an L2 hit.

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

X. Zhang et al. [14] show a novel Flush+Reload attack on ARM devices. In contrast to previously shown Flush+Reload attacks this one works on instruction caches instead of on data caches. The benefit over the Prime+Probe attacks shown by Lipp et al. [1] is that this attack does not need virtual to physical address conversion, so it does not need access to `/proc/<pid>/pagemaps`. A drawback is that it needs shared memory with the victim to work. Also, not all ARM devices have the ability to flush the cache. X. Zhang et al. use the `clearcache` system call as the flush instruction. This system call however, does not flush the cache on all devices, for example on the Krait 450 cores (used by the Nexus 6) it only cleans the cache (write dirty data to a lower cache or main memory), but does not invalidate it.

The attack is executed by an Android app which runs on a different CPU core than the victim application. The basic idea of the attack is to find shared library calls in the victim application that are correlated with the event that the attacker wants to monitor. While the victim application is running, the attacker continually flushes and reloads the cache lines corresponding to these functions. The functions are reloaded by executing them in the attacker context. If the function executes faster than the threshold, the attacker knows that the victim accessed it and hence that an event occurred.

Since executing the complete functions as the attacker takes much time and therefore leads to noise, X. Zhang et al. use a technique that is similar to return oriented programming (ROP). They find ROP gadgets in the functions that they want to monitor and jump to each of those gadgets and return to the attacker program quickly after that. This way they only execute several parts of the functions instead of the full functions.

X. Zhang et al. use the technique to detect touchscreen inputs, detect when a credit-card is scanned in a specific app and detect several features on the display such as notifications and number of characters in password field.

4.4.1. Detecting inclusive L2 caches

In addition to the above attack X. Zhang et al. describe a way to test whether the L2 cache is inclusive to the L1-D or L1-I cache. To detect if the L2 cache is inclusive to the L1-D cache, they measure the difference in the average time it takes to load a dummy function with and without evicting the cache. They evict the L2 cache without polluting the L1-D cache by accessing instructions. Thereby the L1-I cache and L2 cache are cleared, but the L1-D cache is still filled. Then, if the average execution time after eviction is slower than before eviction, apparently the L1-D cache was evicted due to inclusiveness of the L2 cache.

A similar technique is used to test for inclusiveness to the L1-I cache. In that case the L2 cache is evicted by accessing data, so the L1-D cache and L2 cache are evicted, while the L1-I cache is still filled. Then, if the average execution time after eviction is slower than before eviction, apparently the L1-I cache was evicted due to inclusiveness of the L2 cache.

Using these techniques X. Zhang et al. determine that the L2 cache of the Nexus 6 is inclusive to both the L1-D and the L1-I cache.

4.5. AutoLock: Why Cache Attacks on ARM Are Harder Than You Think

Many attacks depend on LLC inclusiveness, i.e. that an eviction from the LLC also evicts the corresponding cache lines from the higher caches. Due to AutoLock (see section 3.2.1) this is not always the case, since higher cache lines can be locked so that a lower cache line cannot be evicted. Green et al. [15] show what AutoLock is, how it can be detected and how devices with AutoLock can still be attacked.

4.5.1. Detection

Multiple tests should be used as each test has different requirements and reliability. The tests are based on the methodology of determining the success of a cross-core eviction strategy that is known to succeed in the same-core scenario.

4.5.2. Attack

To attack a device equipped with AutoLock several methods can be used. One can try to achieve a same-core scenario, trigger self-evictions (make the victim perform the evictions) or increase the probability of an eviction occurring. This probability can be increased by increasing system load, waiting longer or targeting large data structures. These methods also enable cross-core eviction attacks on non-inclusive caches since a cache with AutoLock essentially behaves like a non-inclusive cache.

Green et al. reproduced an attack on a T-Table based AES implementation in a cross-core Evict+Reload scenario.

4.6. Navigating the Samsung TrustZone and Cache-Attacks on the Keymaster Trustlet

Lapid and Wool [16] perform the first attack on ARM TrustZone on a standard device, the Samsung Galaxy S6, which has a 64-bit ARMv8 ISA and uses AutoLock.

The ARMv8 ISA has several exception layers (ELs). The most privileged mode is the Secure Monitor which runs in EL3. The secure world and normal world OSes run in EL1 and the secure world and normal world user spaces run in EL0. An optional Normal-world hypervisor may run in EL2.

The normal world can communicate with the secure world in two ways. Firstly through world shared memory, which allows memory pages to be accessible by both the normal world and secure world. Secondly through SMC calls, that are similar to system calls, but the SMC calls are made from a kernel in EL1 or EL2 to the EL3 Secure Monitor instead of from user space in EL0 to a kernel in EL1, like system calls. See fig. 2 for an overview of the exception levels.

4.6.1. Kinibi OS

Trustonic's Kinibi OS is the secure world OS of the Samsung Galaxy S6. It runs in Thumb (32-bit) mode even though the platform has a 64-bit processor. It is protected by the TrustZone architecture, but internally does not protect itself very well, for instance address space layout randomization (ASLR), non-executable stack and stack canaries are not implemented.

4.6.2. Attacking the Keymaster trustlet

A synchronous PRIME+PROBE attack - where the cache set candidates are first primed, then the AES encryption operation in the trustlet is run, then the cache sets are probed - turns out to be too slow which results in too much noise. Also the asynchronous attack - where the PRIME+PROBE attack runs on a separate thread - fails because it does not present activity that is expected of a T-Table. According to Lapid and Wool this is due to the effects of AutoLock.

The world shared memory can be used to extract information about when in the SMC call the encryption operation takes place. This is done by monitoring specific parts of the world shared memory, namely, the trustlet connector interface (TCI) memory, which contains the request identifier and returns the response, the input and the output buffer. The monitoring is done with FLUSH+RELOAD. The time between the access to the input buffer and the output buffer is the time when the AES implementation is executed.

The final attack consists of four threads. The first makes Keymaster requests in a loop from a different core than the Kinibi OS. The second runs on the same core as the Kinibi OS and conducts a PRIME+PROBE attack on the T-Table cache sets. The third runs a FLUSH+RELOAD attack to select only the measurements from the second thread that fall within the AES execution. The fourth thread creates as many normal world interruptions as possible to increase the likelihood of interrupting the secure execution and allowing the second thread to make a useful measurement.

4.7. Cache-Attacks on the ARM TrustZone implementations of AES-256 and AES-256-GCM via GPU-based analysis [4]

Continuing on their previous paper [16] Lapid and Wool describe how they cracked assembly AES-256 implementations running in Samsung's Keymaster trustlet. They use a divide-and-conquer strategy to smartly calculate the most likely keys based on the observed side-channel traces. They do so by leveraging the parallelism of the GPU on a laptop. With 7 minutes of data collection and under a minute of processing they were able to retrieve the full AES-256 key. The Keymaster trustlet however, uses AES-256 in Galois counter mode (GCM) mode. GCM makes side-channel attacks harder because it uses the block encryption function twice during the initialization phase, creating substantial cache-access noise. Furthermore, subsequent block function invocations made by GCM are called with a counter variable as the last 4 bytes. This means that an attacker has limited control over the input to the block function. Yet Lapid and Wool were able to retrieve the full key of AES-256 in GCM mode with 40 minutes of data collection and 30 minutes of processing.

4.8. Prime+Count: Novel Cross-world Covert Channels on ARM TrustZone

Cho et al. [17] have designed a new type of cache-based covert channel, Prime+Count. The goal of this channel is to work even when security solutions like SeCReT or strong monitors are deployed. SeCReT aims to restrict access to communication channels and secure world resources from the normal world based on access control lists (ACL). Strong monitors use techniques from intrusion detection or deep packet inspection to detect illegitimate use of the communication channels. An additional reason to develop this technique is that Prime+Probe is not noise resistant enough, and can thus not achieve a practical bandwidth, according to Cho et al..

Before priming the cache, all lines in all sets are invalidated using the DCCISW instruction, which can invalidate a specific line in a specific set. They count the number of cache lines that were updated using the performance monitoring unit (PMU) feature L1/L2 Cache Refill Event. Accessing either of the DCCISW or the PMU has to be done from kernel space. To circumvent the problem of data prefetching, the instruction synchronisation barrier (ISB) instruction is used, this instruction flushes the pipeline of a core and the prefetcher buffer.

Cho et al. have compared implementations using set-counting, which counts the number of sets that were accessed, to implementations using line-counting, which counts the number of lines that were accessed across all sets. They conclude that set-counting is able to achieve a higher bandwidth even though the amount of bits transmitted per message is lower. This is because only one cache way needs to be primed which decreases the duration.

A limitation according to Cho et al. is that because of the coarseness of PRIME+COUNT it will be very difficult to use it to spy on a victim program or to extract cryptographic keys from another address space. On the Nexus 6 we were unable to read the cache refill events. This in combination with the limitation mentioned by Cho et al. leads us to not use PRIME+COUNT in this work.

4.9. CLKscrew

CLKSCREW is an attack, performed by Tang et al. [2], that uses dynamic voltage and frequency scaling (DVFS) to raise the clock speed of another core in order to cause errors. If this error occurs at exactly the right time, for example during the 7th round of an AES encryption, this can be used to obtain the secret key from the output of the encryption using differential fault analysis (section 3.5). Since CLKSCREW applies at core-level, it does not matter if the core is running a trustlet (TrustZone program) or normal code. CLKSCREW can also be used to run untrusted code in TrustZone by breaking the RSA signature chain verification routine.

4.9.1. Dynamic voltage and frequency scaling

DVFS is a system for energy management of processors. It enables kernel-level programs to adjust the voltage and frequency of a processor core. This can be done from untrusted code, but has impact also on code running inside ARM TrustZone. This is the key enabler of the CLKSCREW attack since it enables the attacker to overclock the processor from software to induce a fault in code running inside ARM TrustZone.

4.9.2. AES attack

Tang et al. attack a textbook AES decryption implementation that they load into TrustZone using the Trustnone exploit [36]. They measure the execution time of the AES decryption to find when to inject the fault. Then they use DVFS for delivering the fault during the seventh AES round in order to perform DFA and retrieve the secret key (section 3.5).

4.9.3. RSA attack

The RSA implementation in the TrustZone kernel is an interesting attack target since breaking the signature verification enables an attacker to load arbitrary trustlets, i.e. achieve arbitrary code execution. Tang et al. have profiled this implementation in order to find a timing anchor for their fault injection attack. The goal is to find when the FLIPENDIANNESS function is executed with the RSA modulus as input. During the execution of this function, the fault can be injected by overclocking the processor using DVFS. The fault leads to a corrupted RSA modulus which can be factored. The factorized modulus can be used to forge a signature on a custom trustlet. During the verification of this custom trustlet with forged signature, the exact same fault has to be introduced in order to use the same modules N as for the forged signature. If the custom trustlet is loaded, arbitrary code execution in TrustZone is achieved.

Tang et al. use instruction-based PRIME+PROBE since they found that this is more reliable than data-based PRIME+PROBE. They track the duration between subsequent evictions and plot those *eviction gap duration* values (fig. 18). Using this plot they are able to pinpoint the time at which FLIPENDIANNESS runs. Using the fault-injection attack they are able to load the Widevine trustlet with their own signature with a probability of $\frac{1}{65}$.

4.10. VoltJockey

Qiu et al. [18] reproduce and expand on the research of Tang et al. [2]. In contrast to CLKscrew, they focus on using DVFS to manipulate the *voltage* instead of the *clock speed* because clock speed changes are easy to detect and prevent. Qiu et al. perform an attack on AES as well as on RSA. Both attacks are executed on the normal world as well as on the secure world.

4.10.1. RSA

Similar to Tang et al. [2] Qiu et al. perform an instruction-based PRIME+PROBE attack on the RSA implementation in TrustZone. They perform the attack in a cross-core scenario, with the attacker code and the RSA_DECODING function running in parallel on different cores.

4.11. Speculative execution attacks

In this section we describe a series of microarchitectural side-channel attacks. Most of these are not directly applicable to ARM, but we consider them relevant because they are recent advances in side-channel attacks and similar attacks could also be applied to ARM in the future. In addition, several techniques described in these works can be used on ARM. Some attacks, like Spectre [6] and Meltdown [7] are readily applied to ARM. Spectre has been shown to work on ARM by Kocher et al. [6] and a “toy example” of Meltdown is shown to work by Lipp et al. [7].

4.11.1. Speculative execution

In order to gain more performance during instructions that have a long duration, such as memory reads or I/O operations, modern processors use speculative execution. Speculative execution executes ahead while the processor is waiting for a value, thereby guessing the outcome of conditional branches or indirect branches (jumps). If the guess was correct, the speculative execution can be committed, if it was not correct, the execution is discarded. However, even if the results are discarded, traces of the speculative execution are still left in the microarchitectural components of the processor. These traces are the basis of several speculative execution-based microarchitectural attacks, as described in the remainder of this section.

4.11.2. Spectre

In a Spectre attack [6], the attacker mistrains the branch predictor to execute a wrong branch or jump to an attacker-specified code segment. The wrong speculative execution leaves traces in the caches which are extracted using a covert channel. This way, the attacker can access memory for which it does not have authorization, but the victim does. For mistraining conditional branches, the attacker executes the victim program (or a similar program) with values such that the chosen branch is executed. Later when the victim program is executed with the normal arguments, the branch is mispredicted and speculatively executed. For mistraining indirect branches, the attacker makes several jumps to the desired address, thereby training the branch-target buffer (BTB). When the victim program is executed, it will also speculatively jump to the attacker-chosen address. This attack is similar to ROP, since the victim jumps to a gadget in its own address space. The main difference is that in Spectre, the jumps are executed speculatively, while in ROP they are executed in the normal application flow. Also, correctly written programs are still vulnerable to Spectre attacks. Spectre has been successfully executed on ARM devices [6].

4.11.3. Meltdown

Using Meltdown [7], an attacker process can read from memory it should not otherwise be able to access, such as from the kernel, another virtual machine or another application. The memory should be in the attacker's address space, which is not a big limitation since modern operating systems often map the entire kernel memory into every user process. This memory is protected from access through a permission bit. If a user process accesses memory that it is not allowed to access, an exception is raised. Due to out-of-order execution, there is a small time window between the illegal access and the raising of the exception. This out-of-order execution is called a transient instruction sequence. The transient instruction sequence acts as the sending end of a covert channel such as Flush+Reload. The receiving end can determine which address was accessed and hence the secret cache contents.

The attack consists of three steps

1. The content of an attacker-chosen memory location, which is inaccessible to the attacker, is loaded into a register.
2. A transient instruction accesses a cache line based on the secret content of the register.
3. The attacker uses Flush+Reload to determine the accessed cache line and hence the secret stored at the chosen memory location.

Lipp et al. have not been able to make it work on ARM yet, but their "toy example" works, which indicates that it is probably possible to do the exploit on ARM.

4.11.4. Foreshadow

Foreshadow [8] can leak secrets from Intel's secure guard extensions (SGX) enclaves. It is similar to Meltdown in that it uses the same vulnerability. The difference is that while Meltdown targets kernel memory, Foreshadow targets SGX enclaves. The attack can be executed by an unprivileged adversary, i.e. without root access to the victim machine. It does not exploit any software vulnerability or rely on knowledge of the victim enclave's source code. Before the attack, the attacker allocates a buffer of $256 \cdot 4$ KiB, this is called the oracle buffer. The attack consists of three phases. In Phase I, the secret data is cached in the L1 cache by running the victim enclave. In Phase II, a transitive execution sequence accesses an oracle buffer entry dependent on the value of the secret byte, thus caching the corresponding address. In Phase III, the secret is extracted by the receiving end of the FLUSH+RELOAD channel. Van Bulck et al. also show that an adversary with root permissions can read the entire memory of the victim enclave, without any dependence on the enclave code itself. This is done by evicting and reloading each page of the enclave memory in the L1 cache and extracting the contents using the Foreshadow attack.

4.11.5. RIDL: Rogue In-flight Data Load

RIDL [10] is a new class of attacks after Spectre-type, Meltdown and Foreshadow. It works even with the mitigations for Spectre, Meltdown and Foreshadow in place and is harder to mitigate than those attacks. While existing attacks target information at specific address, RIDL operates akin to a passive sniffer by eavesdropping in-flight data in microarchitectural buffers such as the line fill buffer (LFB), load ports or store buffers (SBs). When the victim code loads or stores data, the CPU performs the operation via internal buffers like the ones mentioned above. Then, when the attacker also performs a load, the processor speculatively uses in-flight data from the LFBs rather than valid data. This speculatively loaded data can be extracted using a covert channel such as FLUSH+RELOAD. The attack can be executed without any privileges and from high-level code such as JavaScript.

4.11.6. ZombieLoad: Cross-Privilege-Boundary Data Sampling

Load instructions that have to be re-issued internally may compute on stale value belonging to previous computations on the current CPU core. ZombieLoad [9] exploits this to reveal recent data values without adhering to any explicit address-based selectors. It is the first attack that can leak data on an Intel CPU that is reportedly resistant against all known Meltdown, Foreshadow, and microarchitectural data sampling (MDS) variants. It is not possible to select the value to leak based on an attacker-specified address. But the benefit of ZombieLoad is that it is possible to leak across any security boundary, in contrast to the limitations of previous transient-execution attacks.

When an L1-miss occurs, a fill-buffer entry is required to load the data. When a store misses the L1 or is evicted from the L1 it is stored in a fill-buffer entry as well. Now when a fault occurs during reading of the fill-buffer the processor may first (transiently) read a stale value before the instruction is reissued. This stale value can be leaked via cache side-channels such as Flush+Reload.

Schwarz et al. demonstrate ZombieLoad's effectiveness in a multitude of practical attack scenarios across CPU privilege rings, OS processes, virtual machines, and SGX enclaves.

4.11.7. Fallout: Leaking Data on Meltdown-resistant CPUs

Fallout [37] is similar to Meltdown in that it enables reading from memory that the attacker is not allowed to access. However, the vulnerability that it uses is more similar to RIDL and ZombieLoad. These attacks fall into the category called MDS. Fallout differs from RIDL and ZombieLoad in that it attacks the SB and store instructions instead of the LFB and load ports. Fallout can also be used to read intermediate values from transactional memory extension (TSX) transactions.

Store buffer This buffer handles store operations while the processor can continue executing in the meantime. If a load from an address in the SB is encountered, it is loaded from the SB instead of from main memory, this is called *store-to-load forwarding*.

Write transient forwarding Write transient forwarding (WTF) is a shortcut which incorrectly passes values from memory writes to subsequent faulting load instructions. Namely, if the lower bits of the address of a faulting load match, the processor assumes that the physical addresses match and thus executes store-to-load forwarding. This has no architectural implications since the faulting load instruction is not retired (committed), however, it does leave microarchitectural side effects.

Reading from kernel The WTF shortcut can be used to read data from kernel memory. A kernel module that performs a write is executed by the attacker code. Afterwards the attacker code tries to read the value from the SB. This succeeds with low probability, probably because the SB is already empty once the attacker code runs. The success probability can be increased to about 80% if the kernel code performs more than 10 arbitrary writes before the targeted write.

Reading the AES key Using the WTF shortcut and a kernel module that performs AES the last two round keys can be obtained, these can be used to derive the master key. First the offset of the subkeys in kernel memory have to be detected, this is done by using WTF on different offsets 128 bytes apart so that at least one of the offsets hits the subkeys. The frequency of leaked bytes is detected for each of the offsets and results in a clear peak for the correct offset. Then WTF is used again to read the key from that offset.

5. Bare-metal environment

In this project a bare-metal environment is used for the experiments. The bare-metal environment runs directly after the primary bootloader (PBL), secondary bootloader (SBL) and Android bootloader (ABoot), it is a very bare-bones replacement for the Linux kernel, written specifically for the Nexus 6. The environment has no scheduling, and thus no programs running in the background. This has benefits for cache side channels as the operating system noise is minimized, as stated by Green et al. [15]. In addition, we have direct access to physical addresses which makes determining the cache set belonging to an address trivial. Support for communication with TrustZone was built into the bare-metal environment before this project.

The bare-metal kernel initially did not support multiple cores or caching. These parts have been implemented in the early stages of this project and are described below. Multi-core support is necessary for this project since it enables running the attacker code and the victim code concurrently.

The source code for the bare-metal environment has been published [22].

5.1. Multi-core support

When the bare-metal environment is booted, only CPU 0 is enabled. A second CPU can be enabled by the following steps. First the boot address is set using an `SCM_SVC_BOOT` secure monitor call (SMC) with the `SCM_BOOT_ADDR` command. Then the values `0x21`,`0x20`,`0x00`,`0x80` are written successively to the power control register of the CPU, located at `0xf9088000` + (`cpu_id` ∗ `0x10000`) + 4. When powered up, the CPU jumps to the boot address, where it sets the stack pointer (SP) and waits for a function to execute.

When a function should be executed, the function pointer and arguments are written to a predefined location. The CPU detects that a function is ready to be executed and calls it with the given arguments before returning to the waiting state.

5.2. Enabling caching

When the bare-metal environment is booted, the memory management unit (MMU) is disabled, and thus all memory is used as strongly ordered memory. Strongly ordered memory means that all previous memory operations have to be finished before the next one can be started and that the addresses in the strongly ordered region are not held in cache [38]. To enable the MMU, first a page table is created, then the address of the page table is written to the translation table base register (TTBR), finally the MMU and caching are enabled in the system control register (SCTLR). The entries in the page table that correspond to the memory that is used for the cache attacks should be cacheable as indicated by the TEX, B and C bits in the translation table entry description. We have chosen the attributes write-through and no write-allocate, which means that data is written to the cache and storage simultaneously, but that no cache line is allocated for a cache miss on write. We first tried to enable a write-allocate policy, but from our experiments it seems that that combination of settings is not supported. The consequence of this policy for our experiments is that data has to be read before it is cached. Before enabling the MMU, the translation lookaside buffers (TLBs), branch-target buffer (BTB) and instruction caches should be invalidated to prevent old entries from being loaded. A TLB acts as a cache for the page table, it saves a previously created virtual-to-physical translation. A BTB is used for branch prediction based on previously executed branches. Further details about enabling the MMU can be found in the ARM reference manual [38].

For this project a one-to-one virtual-to-physical mapping is used to ease virtual-to-physical address conversion. Since Linux uses a 3-level page table, a page walk (TLB miss) on Linux will cost more time than in our bare-metal environment.

5.3. Randomness

In order to access randomness, the hardware pseudo-random number generator (PRNG) of the Krait 450 CPU is used. This PRNG is accessed using the I/O (input/output) memory at `0xf9bff000`. In order to access it, the `APCS_CLOCK_BRANCH_ENA_VOTE` clock is enabled.

5.4. Performance counters

To have more information on cache usage, the hardware performance counters of the Krait 450 CPU can be used. Using these counters several events can be counted, among which are events for L1-D (level 1 data) cache, L1-I (level 1 instruction) cache and L2 (level 2) cache usage.

To enable these we first enable performance counters by writing to the performance monitors control register (PMCR). Then we select one of the counters using the performance monitors event counter selection register (PMSELR) and set the desired event using the performance monitors event type select register (PMXEVTYPER). After that, the desired counter can be enabled by writing to the performance monitors count enable set register (PMCNTENSET). Finally, the counters are reset using the PMCR.

After these steps, we were able to read several architectural events as described in [38]. However, reading microarchitectural events like the cache events results in reading 0 always. We have not been able to find the cause for this behaviour.

Even after adding some Krait-specific initializations, we are unable to read the microarchitectural events. The full implementation can be seen in [22].

6. Attack primitives

This section describes the primitives necessary to perform a cache side-channel attacks.

6.1. Eviction

The Krait 450 cores of the APQ8084 of the Nexus 6 use a random replacement policy just like most ARM processors [1]. Therefore, a simple least recently used (LRU) eviction where an address is accessed for each cache line in a set, does not evict the set with high probability. To achieve eviction with high probability, sliding window eviction is used (see section 3.3). The `eviction_strategy_evaluator` program [39] by Lipp et al. [1] is used to evaluate all combinations of $N \in [8, 50]$, $A \in [1, 10]$ and $D \in [1, 10]$. The combination $N = 16, A = 6, D = 1$ has the best compromise between eviction rate and number of clock cycles on Android (table 1a). It also performs better than $N = 50, A = 1, D = 1$ used by Green et al. [15].

On bare-metal a similar experiment is performed with $N \in [8, 16] + [50]$, $A \in [1, 6]$ and $D = 1$. The results are shown in table 1b. We include 50 here because Green et al. [15] use a strategy with $N = 50$.

There is a clear difference between the two tables, this is probably due to the lack of background processes on bare-metal.

The best eviction strategy is not necessarily the best strategy for Prime+Probe. A strategy that is optimal for eviction could for example

(a) Android (with eviction rate > 90%)

N	A	D	Rate	Cycles
16	6	1	96.5	2591
28	5	1	96.6	4370
31	4	2	92.9	4736
21	6	3	90.4	5355
25	4	3	96.1	6522
25	3	8	94.9	6974
38	2	5	95.5	10870
32	4	3	95.2	11224
41	4	5	94.7	12140
40	5	4	97.3	12240

(b) Bare metal

N	A	D	Rate	Cycles
12	1	1	100.0	1274
10	2	1	100.0	1358
9	3	1	100.0	1640
13	1	1	100.0	1652
11	2	1	100.0	1653
14	1	1	100.0	1729
12	2	1	100.0	1779
15	1	1	100.0	1825
13	2	1	100.0	1871
10	3	1	100.0	1889

Table 1: Eviction strategies sorted by eviction time

N	A	D	Rate	Cycles
8	5	1	99.9	1071.42
8	6	1	99.9	1082.489
9	6	1	99.9	1204.027
10	3	1	99.9	1554.902
12	2	1	99.9	1712.398
11	2	1	99.6	1749.726
10	6	1	99.55	1559.683
9	4	1	96.55	1294.523
7	3	1	95.45	945.655
7	5	1	95.35	946.337

Table 2: Strategies for PRIME+PROBE (bare metal)

evict addresses used earlier in the eviction strategy, then during probing, also the evicted address would be probed, resulting in a cache miss and diminishing the accuracy. Therefore, we also tested the strategies for Prime+Probe, the results of which are shown in table 2. It is not surprising that the optimal number of addresses for Prime+Probe is the amount of lines per cache set.

6.2. Flush

ARMv7 does not have a flush instruction, as stated by Lipp et al. [1], however, the coprocessor (cp15) does have some cache maintenance operations which are only accessible from kernel level, including the data cache clean and invalidate by set/way (DCCISW) operation. This operation can be used in a loop over all lines in a cache set in order to flush a cache set. This is much faster than the most optimal eviction strategy (595 clock cycles vs 1274 clock cycles, see table 1b).

A proof of concept Flush+Flush attack was attempted, but no difference in timing between flushing a cached set and an uncached set was observed. So probably the DCCISW instruction is not dependent on whether addresses are cached.

6.3. Timing

As the timing source for measurements, the cycle count register (PMCCNTR) is used. This register can be read from the cp15 coprocessor using an MRC (move to ARM register from coprocessor) instruction. Since our attacker model permits kernel-level access, reading this timer is not a problem, unlike previous work by Lipp et al. [1].

6.4. Delays

In order to wait for a given amount of clock cycles a function is used that takes as input the amount to wait and executes a number of loop iterations. Each loop contains a single no operation (NOP) instruction. The optimal number of loop iterations is experimentally determined to be the input amount divided by 24. This results in a linear mapping, which is shown in fig. 3.

6.5. Test cache-attack methods

To verify our attack primitives, tests were conducted to see if accessing a cache-set results in a measurable difference for the attacker. An experiment is conducted for Flush+Reload and Prime+Probe, both on data and on instructions. In each experiment, first n attacks are executed where the monitored cache-set is not accessed during the measurement, and after that n attacks where it is accessed. If there is a significant difference, we know that the attack primitive works. In our experiments $n = 1000$.

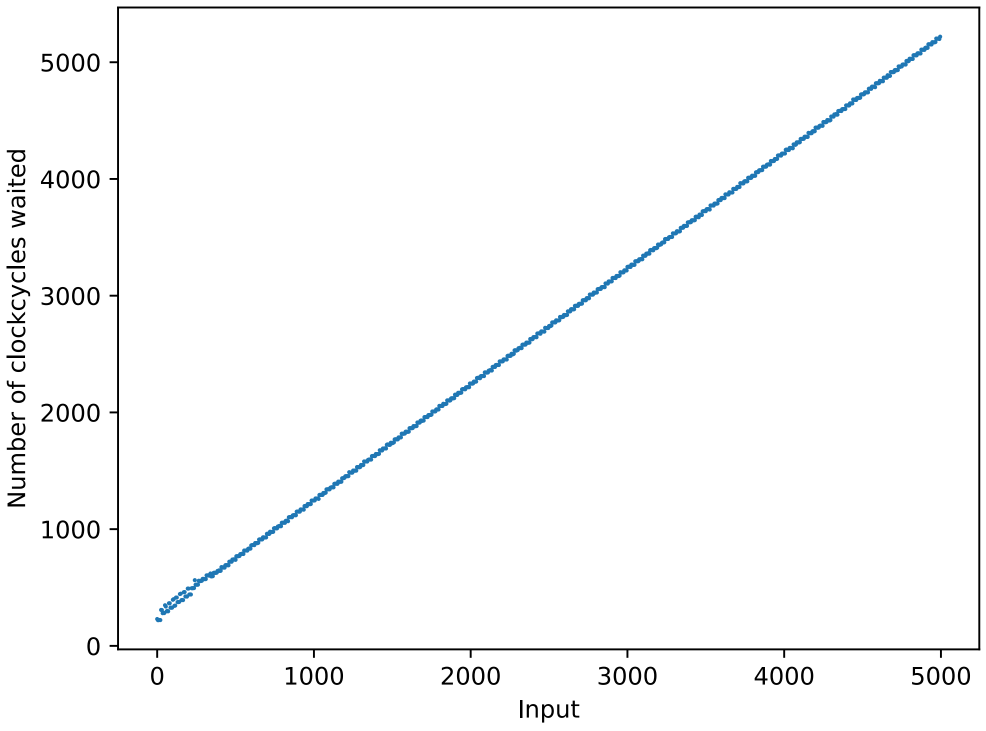


Figure 3: Cycle count for values from 0 to 5000

Algorithm 3: Data-based FLUSH+RELOAD where address is accessed by the victim

```
   target: address to monitor
1  procedure test
2      index = getIndex(target);                          /* Cache set */
3      flush(index);                  /* Attacker flushes the cache set */
4      access(target);                  /* Victim accesses the address */
5      start = getTiming();               /* Attacker measures timing */
6      access(target);
7      stop = getTiming();
8      return stop-start;
9  end
10 procedure getTiming
11     ISB;
12     DSB;
13     value = read(PMCCNTR);
14     ISB;
15     DSB;
16     return value;
17 end
```

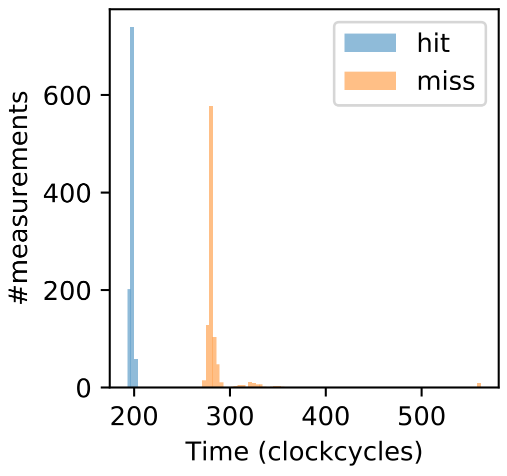
The pseudo code for a data-based FLUSH+RELOAD where the address is accessed before measurement is shown in algorithm 3. For the data-based tests the cache-set is accessed by executing an LDR (load register) instruction on the targeted address (the `access` function in algorithm 3). For instruction-based tests `target` is a function that does a NOP and it is executed instead of accessed.

For PRIME+PROBE the algorithm is similar to algorithm 3 but instead of `flush`, `prime` is executed, and instead of the `access` on line 6, `probe` is executed.

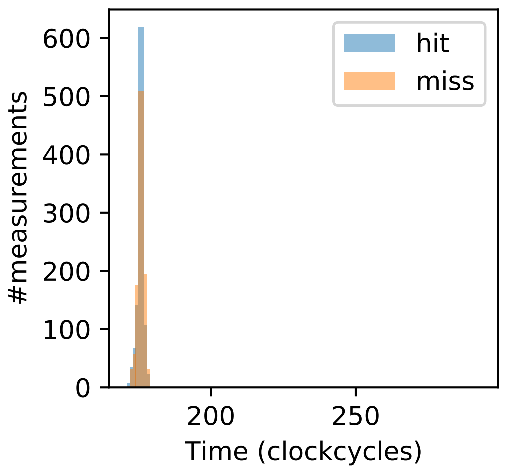
Before and after every access of the timing source, an ISB (instruction synchronisation barrier) and DSB (data synchronization barrier) are performed to make sure that the instructions are not reordered and that the memory accesses during the PROBE/RELOAD step are done during the measurement.

6.5.1. Flush+Reload

The results of the data-based and instruction-based tests for FLUSH+RELOAD are shown in fig. 4. In the data-based test there is a significant difference between the hit (address was cached) and the miss (address was not cached). However, for the instruction-based test, there is no difference.



(a) Data-based



(b) Instruction-based

Figure 4: Flush+Reload Test

6.5.2. Prime+Probe

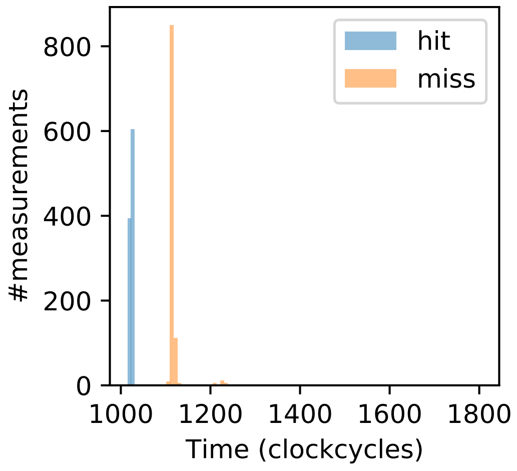
The results of the data-based and instruction-based tests for Prime+Probe are shown in fig. 5. Aside from the higher number of clock cycles, the results are very similar to those for Flush+Reload. Again, similar to Flush+Reload there is a difference for the data-based test, but no difference for the instruction-based test.

Test L2 instruction-inclusiveness According to X. Zhang et al. [14], the L2 cache of the Nexus 6 is inclusive to the L1-D cache as well as the L1-I cache. They tested this using the method described in section 4.4.1.

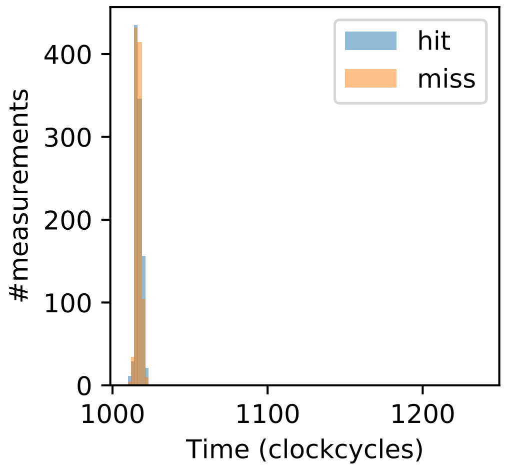
Our instruction-based experiments in sections 6.5.1 and 6.5.2 are similar to the experiment of X. Zhang et al., we also measure the difference in execution time of a dummy function with and without first evicting the L2 cache with data. However, in our experiments there is no difference between the execution time without eviction and the execution time after eviction, this would indicate that the L2 cache is *not* inclusive to instructions.

To further verify this conclusion we perform the instruction-based Flush+Reload test (section 6.5.1) but in addition to flushing the L2 cache, we also flush the L1-I cache. The results are shown in fig. 6 this does result in a significant difference, indicating that the L1-I cache was not evicted before.

A cause for the different results could be the difference in length between our dummy function which is a single NOP instruction and the dummy function of X. Zhang et al., which is 1 KB of `mov r0, r0` instructions. To verify



(a) Data-based



(b) Instruction-based

Figure 5: PRIME+PROBE Test

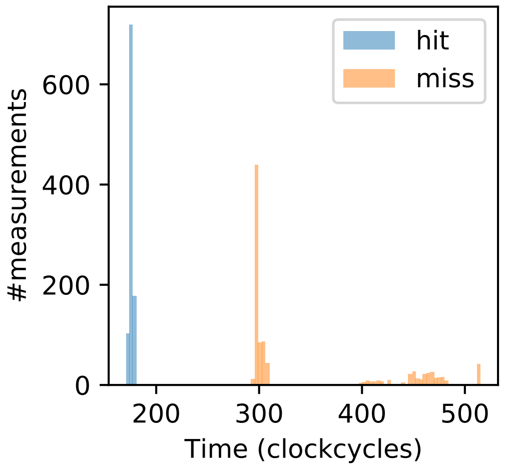
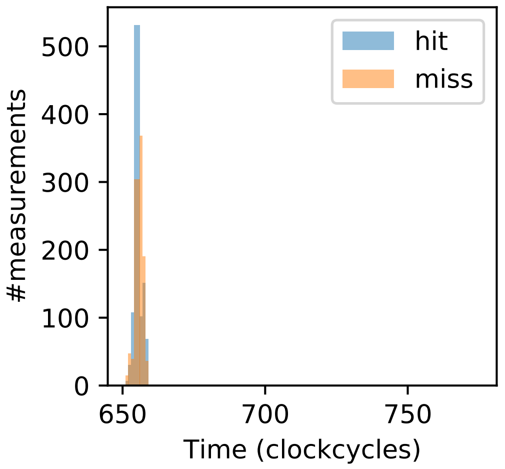
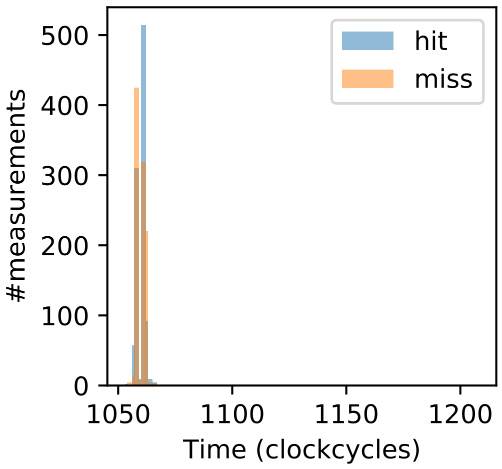


Figure 6: Instruction-based FLUSH+RELOAD when flushing the instruction cache



(a) Flush+Reload



(b) Prime+Probe

Figure 7: Instruction tests with 1000 KB `mov r0, r0` instructions

this, the dummy instruction was changed to have 1 KB of `mov r0, r0` instructions and the experiment was repeated. The results are shown in fig. 7. We conclude that this does not cause a difference in the results.

Another cause for the difference could be the difference in platform, we run on bare-metal, while X. Zhang et al. run on Android. Our initialization of the MMU and the caching behaviour may deviate from the Android kernel, however, we have not been able to find the differences.

7. AES attacks

This section describes attacks on several AES implementations.

7.1. Textbook AES

A 128-bit textbook AES implementation [40] is attacked using a Flush+Reload attack. The AES code is initialized by the attacker and then executed on the second CPU with the attacker controlled buffer containing the input, `roundkey` buffer and key as a parameter.

The attacker's goal is to profile the execution of the victim's AES implementation, i.e. to know at what stage the victim is in the execution at any point in time. Since the attacker shares the memory of the round key buffer with the victim, he can monitor that buffer in order to find when the victim executes the `AddRoundKey` method of AES, indicating that a new AES round has been started. Since the Krait 450 uses a 128 byte line size, the 172 bytes round key buffer will reside in two different cache sets. To this end two addresses within the round key are selected that will reside in different cache sets. The cache set that an address belongs to can be calculated based on the physical address. Concurrently with the execution the attacker performs a Flush+Reload on the two selected addresses. The is as shown in fig. 8.

The 11 accesses of the round key of the 128-bit AES implementation can be distinguished. One initial `AddRoundKey`, 9 rounds and the final round (see section 7). 8 Round keys are in the first cache line ($8 \cdot 16$ bytes = 128 bytes) and the other 3 round keys are in the second cache line.

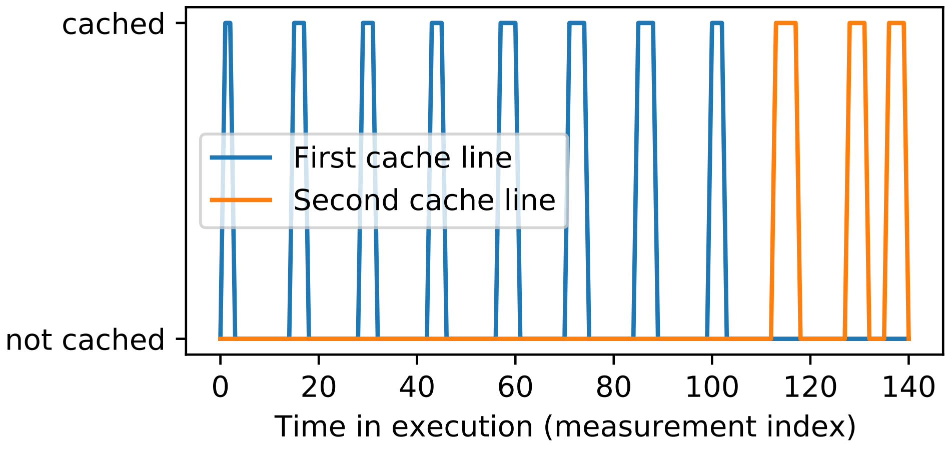


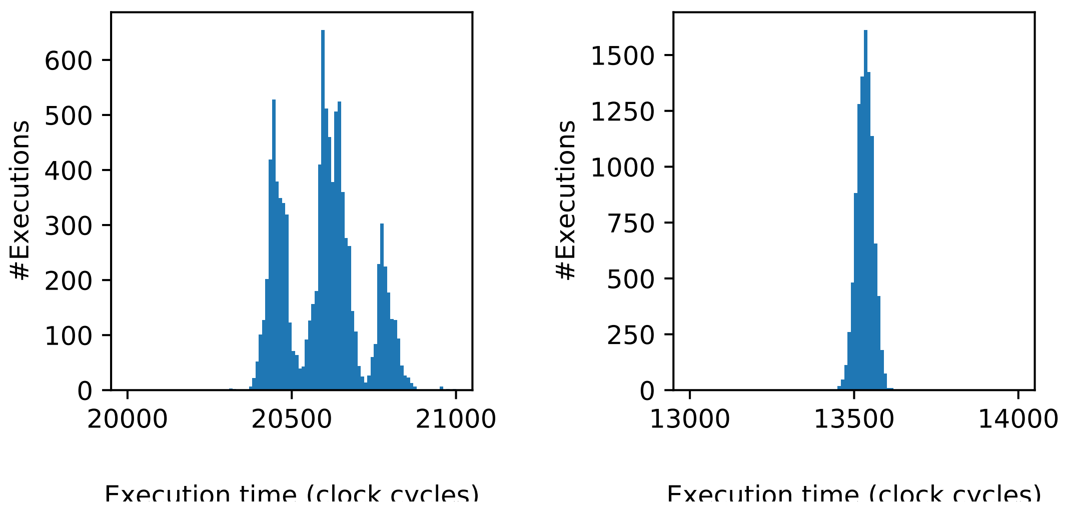
Figure 8: Flush+Reload attack on textbook AES implementation

7.2. OpenSSL AES

The OpenSSL AES implementation [33] uses T-Tables to speed up the operation. The faster code makes it harder to profile since fewer measurements can be done during a run of the algorithm. While a Flush+Reload attack on the textbook AES can do about 140 measurements, the same attack on the OpenSSL implementation can do only about 8 measurements.

7.2.1. Execution time

In order to have a reference of the time within the execution, the expected total execution time is measured. This is done by repeatedly executing an AES operation on a second CPU core and measuring the time it takes from the attacker core. The results can be seen in fig. 9. On the primary CPU we see that the results are spread around a single value, however for the second CPU, we see three peaks, with about 200 clock cycles difference from each other. Since the second CPU busy-waits for a function to execute, one more iteration of the waiting loop could explain the difference [22].



(a) Second CPU (b) Primary CPU

Figure 9: Execution time of AES over 10000 runs

7.2.2. Flush+Reload profiling

The experiment described in section 7.1 was repeated on the OpenSSL AES implementation. To improve the accuracy of the measurements the results of multiple AES executions are combined. Two arrays are created, for the two cache lines of the round key. The indexes of the arrays correspond to intervals within the AES execution time. Several AES runs are performed sequentially on a second core. During each run, a series of Flush+Reload measurements are performed on the two cache lines of the round key. We save the value of the PMCCNTR at the start of the AES execution. The number of clock cycles from the start of the AES execution to the measurement indicates the index in the array that the measurement belongs to. The corresponding value is incremented if the measurement indicated that the address was used by the AES execution. The start of the measurements (on the attacker core) is delayed a random period after the start of the AES operation on the victim core. This delay results in an even spread of the measurements over the time intervals. Accumulating the results over multiple measurements in this way results in a greater resolution than a single measurement would give.

The output of this experiment can be seen in fig. 10. Based on the location of the round key in the memory we know that 8 round keys are in the first cache line and 3 round keys are in the second. Since we observe 7 round key accesses in the first line and 3 in the second, we know that the first round key access was not measured. We presume that this may be caused by the time the attack core needs between starting AES on the second core and

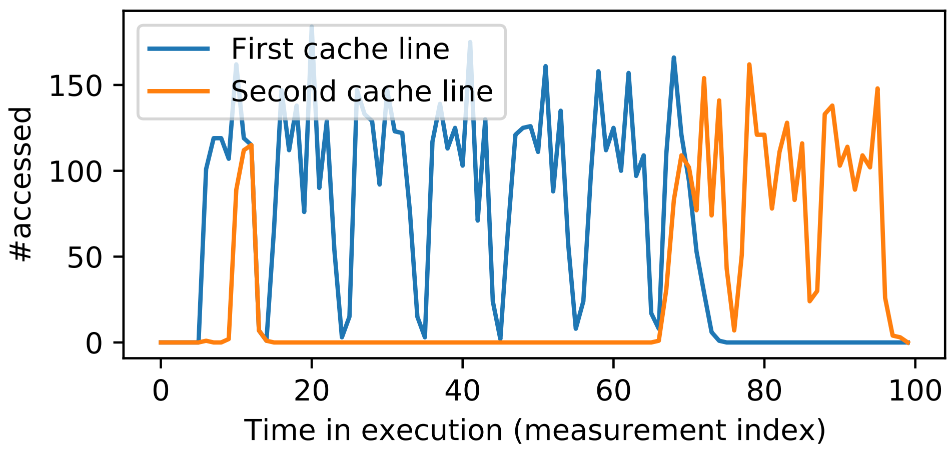


Figure 10: OpenSSL profiling, 100 sample points and 1000 AES executions

starting the first measurement. During that time the AES execution has already started.

A smaller spike on the second cache line can be seen at the start of the measurements, this is because the number of AES rounds to perform is stored after the round key, so this variable shares the cache line with the second part of the round key. The number of rounds to perform is accessed at the start of the AES execution. We verified this by hardcoding the number of rounds in the AES implementation, we observed that this eliminated the first spike on the second cache line.

7.2.3. Prime+Probe profiling

Since the round key is no longer in shared memory when the AES process is running in TrustZone, Flush+Reload cannot be used in that situation. Therefore, Prime+Probe could be used to perform the same attack. The same experiment as described in section 7.2.2 has been performed using Prime+Probe, however, due to the longer probing time, all timing measurements are above the threshold, so averaging cannot increase the resolution.

A possible solution to find the correct round even with this low resolution would be to attack a round key with a specific division between the cache lines such that round keys 1-6 are in the first cache line and 7-11 are in the second cache line. In that situation, only the second cache line is monitored, and the first measured access of that cache line is the start of the 7th round. This, however, depends on the attacker being able to obtain such an offset for the round key.

Another solution would be to change the relative clock speed of the cores so the attacker core runs faster than the victim core, resulting in more measurements during a single run. In this work we did not verify this in order to have enough time to perform our other attacks.

7.2.4. Locating the T-Tables and round key

In the previously described Flush+Reload attack, the round key is used to leak the state of the AES operation. This round key is available since the invocation used an attacker-supplied key. In practice however, the attacker does not have access to the key, so he has to find another way to find its cache location in order to profile it.

This can be done using a technique based on the *statistical matching* technique described by N. Zhang et al. [13]. Each cache set is primed before an AES invocation and probed afterwards, this indicates which cache lines were used during the AES invocation. The same Prime+Probe measurement is performed with a NOP loop instead of the AES invocation in order to have a baseline profile of cache usage. These results are subtracted to find the cache sets that are used by the AES implementation. This heat map is shown in fig. 11.

A line of 32 consecutive used cache lines can be seen. This amounts to $32 \cdot 128 = 4096$ Bytes = 4 KB of memory, which is exactly the amount of memory used by the encryption T-Tables. This was verified by dumping the memory locations of the T-Tables from the AES implementation and checking that the addresses matches the cache set indices in fig. 11.

In the same way we verified that the two consecutive used cache sets on row 8 correspond to the round key. We have not been able to find the cause of the used cache sets on row 9.

Since the Prime+Probe technique does not rely on shared memory, only on a shared cache, it can also be applied to an AES implementation running within TrustZone.

7.2.5. TrustZone AES

An interesting AES implementation to attack would be the one located in the Keymaster trustlet. As discussed in section 4.6 Lapid and Wool [16] perform cache attacks on the AES implementation running in Samsung's Keymaster trustlet. In this project we have decided not to attack the ARM implementation in Qualcomm's Keymaster trustlet running on the Nexus 6 due to the following difficulties. Firstly, we were unable to determine if Qualcomm's Keymaster trustlet uses a software implementation of AES, like Samsung's Keymaster trustlet does. If the AES implementation uses dedicated hardware (i.e. the crypto engine) a cache-based attack would not work. Secondly, due to the low resolution of the Prime+Probe profiling attack it is unlikely that it would succeed on the Keymaster implementation. Due to these reasons we deemed it more useful to look at profiling RSA in

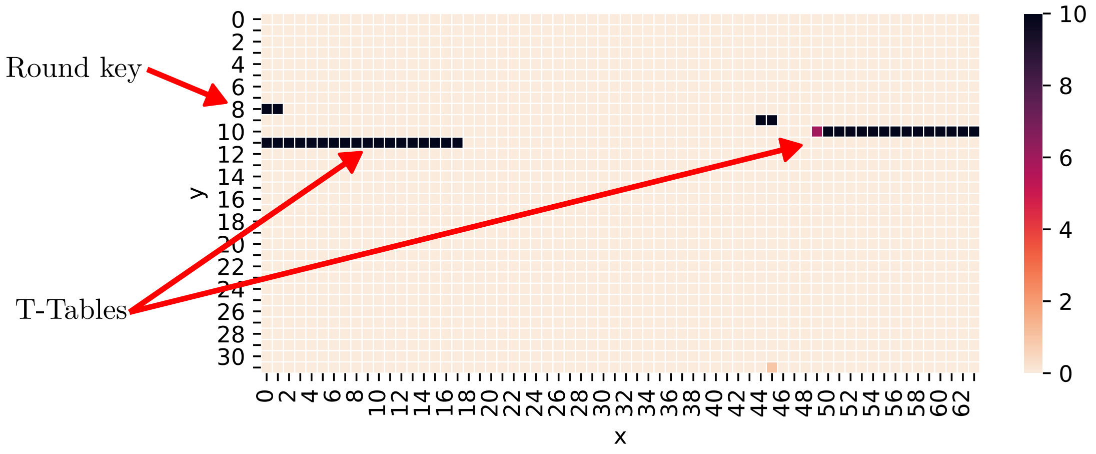


Figure 11: Amount of times each cache set was used out of 10 evocations of AES, obtained using PRIME+PROBE. Cache set $= x + y \cdot 64$

TrustZone.

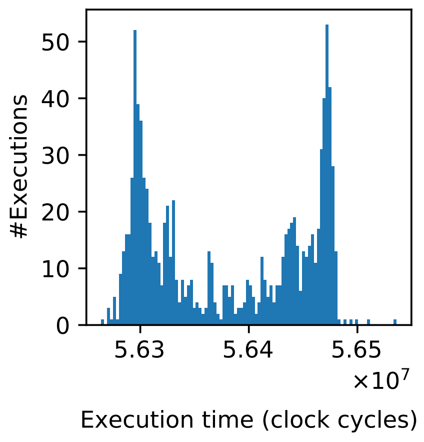
8. RSA attacks

As described in sections 4.9.3 and 4.10.1, Tang et al. [2] and Qiu et al. [18] have profiled the RSA implementation used for verifying trustlets using instruction-based PRIME+PROBE. We want to reproduce their research, and provide a practical implementation, since no public implementation of their profiling attack is available.

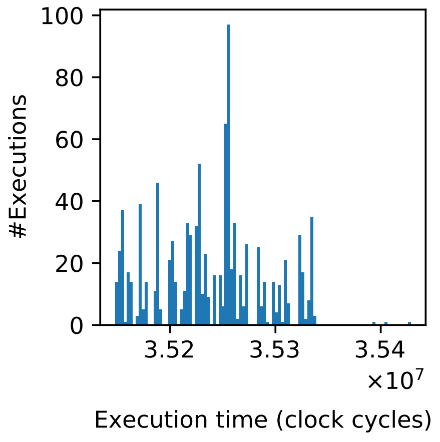
8.1. Execution time

To have a reference for how long an RSA encryption takes, we measure the execution time of the OpenSSL RSA implementation [41]. For the primary CPU the times range from $3.515 \cdot 10^7$ to $3.535 \cdot 10^7$ (not considering the outliers) which is a variation of $2 \cdot 10^5$ clock cycles. While for the second CPU, it ranges from $5.626 \cdot 10^7$ to $5.647 \cdot 10^7$ (not considering the outliers) which is a variation of $2.1 \cdot 10^5$ clock cycles. Even though the variation in clock cycles is not much different, we see that again, similar to AES, the execution time on the second CPU has more than one peak. For RSA, however, there is also no single peak for the primary CPU, there is considerable spread besides the main peak. A reason for this might be that due to the longer execution time, there is more room for variation to stack up. Variation can occur for several reasons, among which are the data cache, instruction cache and BTB.

RSA is several orders of magnitude slower than AES, ($3.5 \cdot 10^7$ vs $1.3 \cdot 10^4$), therefore profiling it with PRIME+PROBE does not suffer from resolution



(a) Second CPU



(b) Primary CPU

Figure 12: Execution time of OpenSSL RSA over 1000 runs

problems like for AES.

8.2. RSA TrustZone attack

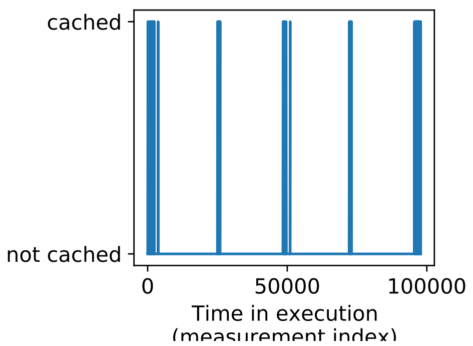
Tang et al. [2] provided pseudo-code and memory locations for the RSA function (they call it DecryptSig). The pseudo code is shown in algorithm 5.

The attack setup is as follows. On one core, the Keymaster trustlet is loaded into TrustZone. During the loading of this trustlet, on another core, a Prime+Probe attack is executed on a single cache set. The attack primes the cache set, then measures the time taken to probe the cache set and stores in an array whether the time difference was greater than the threshold, indicating that the address was used by the victim.

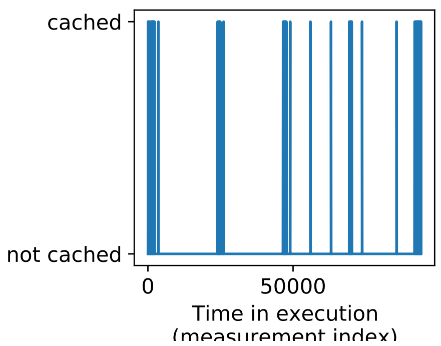
This attack is performed for each of the 2048 cache sets in the L2 cache. The results of some cache sets are shown in fig. 13.

A relevant cache set to look at is the one belonging to an address of the FlipEndianness function, which is located at `0xfe868494` in firmware version ngi77b [21]. We found this using the pointers Tang et al. [2] provided for another firmware version. See algorithm 6 for the pseudo code of this function. The address `0xfe8684a6` is located inside the inner loop in FlipEndianness and is thus accessed multiple times during the execution of FlipEndianness. The L2 cache set that this address belongs to is 1289 (calculated using algorithm 4). The result for this set is shown in fig. 13a.

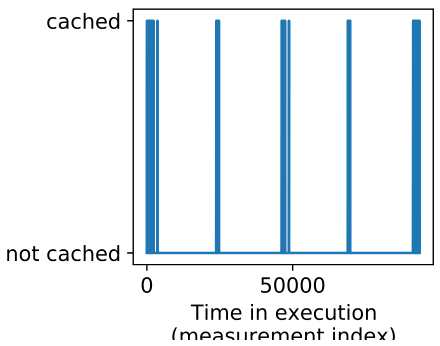
The traces for each of the cache sets have at least the same peaks that are shown in fig. 13a. Some cache sets are used more, such as fig. 13b. The cache sets that were used significantly more are 2, 102, 493, 968, 969, 1280, 1312, 2046. We show the results for 1280 as an example of the differences with other cache



(a) Cache set 1289



(b) Cache set 1280



(c) Cache set 298

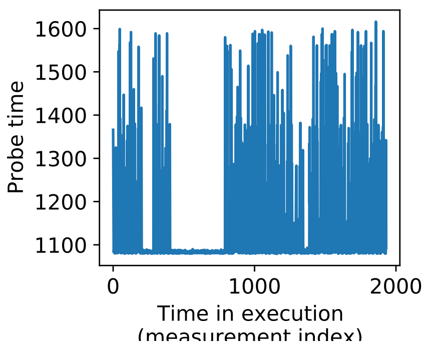
Figure 13: Prime+Probe attack on a single cache set during loading of Keymaster

sets fig. 13b. We were not able to correlate the cache sets that were used more to relevant places in the RSA code.

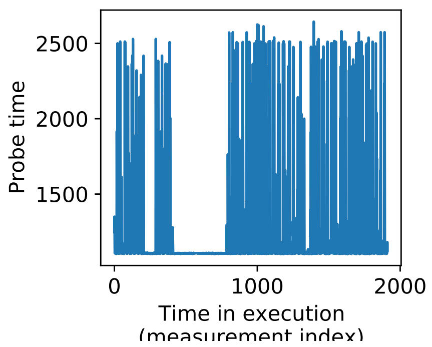
Another relevant cache set to look at is the one which is used by the memory location where the RSA modulus N is stored. N is stored at `0xfc8952c`, which belongs to cache set 298. We found this address, again, using pointers from Tang et al. [2]. The trace for this cache line is shown in fig. 13c. This trace again has the same features as the other traces, so we cannot deduce much information from it.

A flush operation could explain that all cache sets are evicted at the same time. By examining the TrustZone binary, we see at least some cache flushes (using DCCISW) are executed. However, it is unlikely that these flush operations are executed this often (about 1400 times).

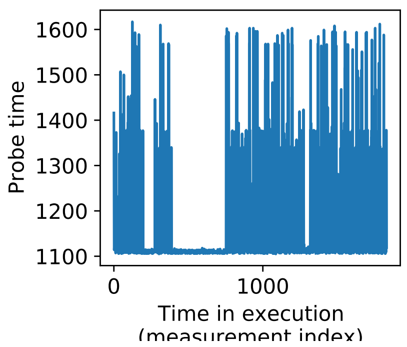
Since during a single trustlet load, the RSA function is executed four times, for each of the certificates [2], we can corrupt the first signature that is stored in the Keymaster binary, to make sure that RSA is only executed once. Instead of hardcoding the eviction threshold, we plot the actual timing values in order to have more information available. The results are shown in



(a) Cache set 1289



(b) Cache set 1280



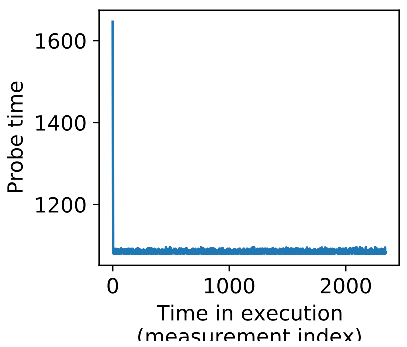
(c) Cache set 298

Figure 14: Prime+Probe attack on a single cache set during loading of corrupted Keymaster

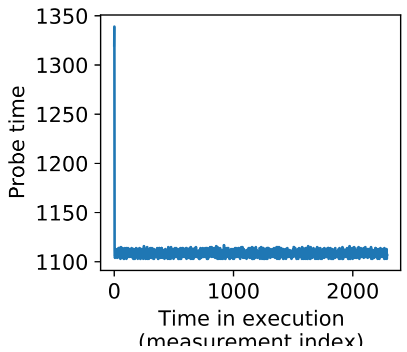
fig. 14. Again, we see the same pattern occurring in all cache sets.

To verify that we are actually measuring a side channel, we replace the SMC to load the trustlet by a call to a dummy function that performs NOPs for a similar amount of time. These results are shown in fig. 15. The peak at the start is probably due to the fact that the attacking code has not been cached yet. Corrupting the Keymaster certificates resulted in a much shorter trace. Therefore we know that the majority of the trace of fig. 13 was measured during the TrustZone execution. Since figs. 15a and 15c are consistently low, we know that it makes a difference whether we execute the code to load the trustlet or not.

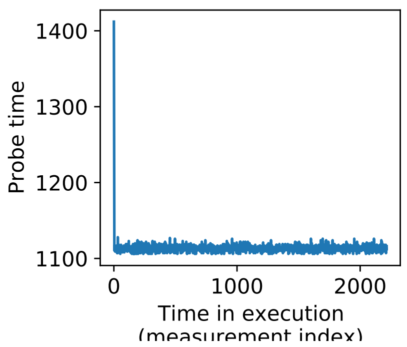
Tang et al. [2] use the time since the previous value above the probe threshold (gap values) as the metric to determine the time at which FlipEndianness is called on N, their plot is shown in fig. 18. They count the time using a counter that is implemented in a loop. We also measure gap values, but instead of using a counter, we measure the amount of clock cycles since the last measurement above the probe threshold (1200), this is shown in fig. 16.



(a) Cache set 1289



(b) Cache set 1280



(c) Cache set 298

Figure 15: Prime+Probe attack on a single cache set during execution of a dummy function

Tang et al. [2] describe that each certificate verification is preceded by a large spike into the 75000 on their counter, followed by a smaller spike of around 180, see fig. 18. We observe the same behaviour, each large spike (around $7 \cdot 10^7$ clock cycles) is followed by a smaller spike, see fig. 17.

There are some differences between our plots and those of Tang et al. [2], namely, our plots contain more small spikes (< 100000), this indicates that we probably measure more often than Tang et al.. Furthermore, our y-scale is different, this is due to the fact that we measure clock cycles, while Tang et al. measure counter increments, which is much slower. Besides these differences, the behaviour is much the same, so it seems that we measure the same side channel. To verify this completely, we need more information on specifics of the of Tang et al. [2]. Tang et al. experimentally determine that FlipEndianness runs after the last spike, however, it is unclear to us, what experiments they performed. Furthermore, we do not know whether they also observed that the results for all cache sets are very similar.

We have had email contact with Tang et al. [2]. In response to our questions they provided more information about the AES implementation that was used. However, we did not receive a response to our later questions regarding the details about their RSA profiling.

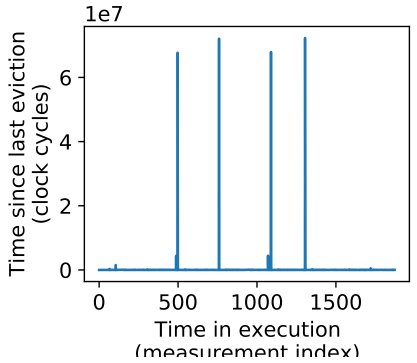
8.2.1. Instrumentation

To obtain more information on what the Prime+Probe measurements indicate, we attempted to instrument the RSA function. Beniamini [42] has written an exploit which could be used to gain code execution in TrustZone. This exploit works for firmware version lmy48m [43]. We planned to overwrite parts of the RSA function with a jump to our shellcode, then let that shellcode write timing information to a shared buffer before jumping back. This shared buffer could then be read from the normal world to obtain a trace of the RSA execution.

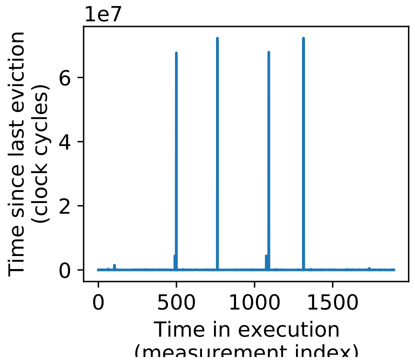
Before writing the full exploit, a test shellcode was written, which aims to overwrite the first 8 bytes of the FlipEndianness in the TrustZone kernel. After overwriting the function, the first 20 bytes are copied into the shared buffer. This shared buffer is then printed in user space.

We observe that the first 8 bytes that are read exactly correspond to what we had overwritten and the remaining bytes exactly match the function. So it seems that the function is overwritten. However, when we execute a second shellcode which only reads the bytes of the function, we find that the original data is there.

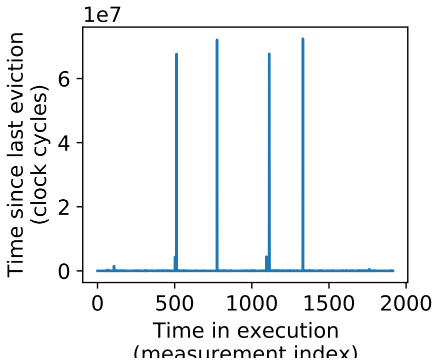
Beniamini [42] describes that due to XPU (external protection unit) memory protection the majority of the TrustZone kernel is not writable. To test whether this is the cause of the behaviour we observe, we try to overwrite an address in a region of the TrustZone kernel that is unprotected according to Beniamini, namely the range from `0xfe806000` to `0xfe810000`. Values



(a) Cache set 1289

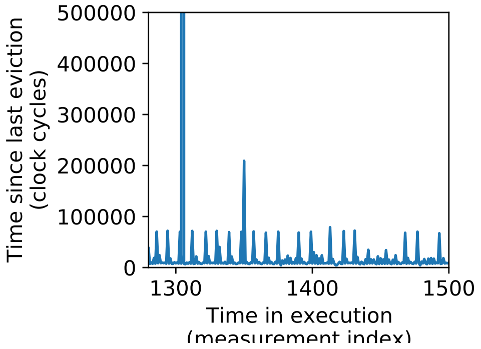


(b) Cache set 1280

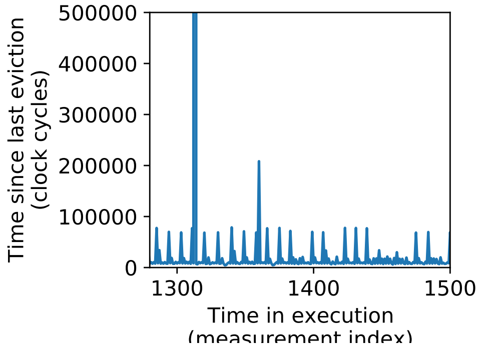


(c) Cache set 298

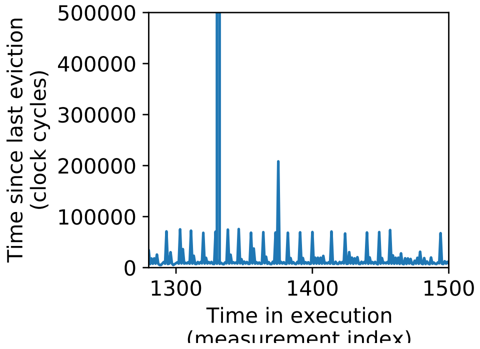
Figure 16: Gap values for Prime+Probe attack on a single cache set during loading of Keymaster



(a) Cache set 1289



(b) Cache set 1280



(c) Cache set 298

Figure 17: Gap values for Prime+Probe attack on a single cache set during loading of Keymaster (Validation of the 4th certificate)

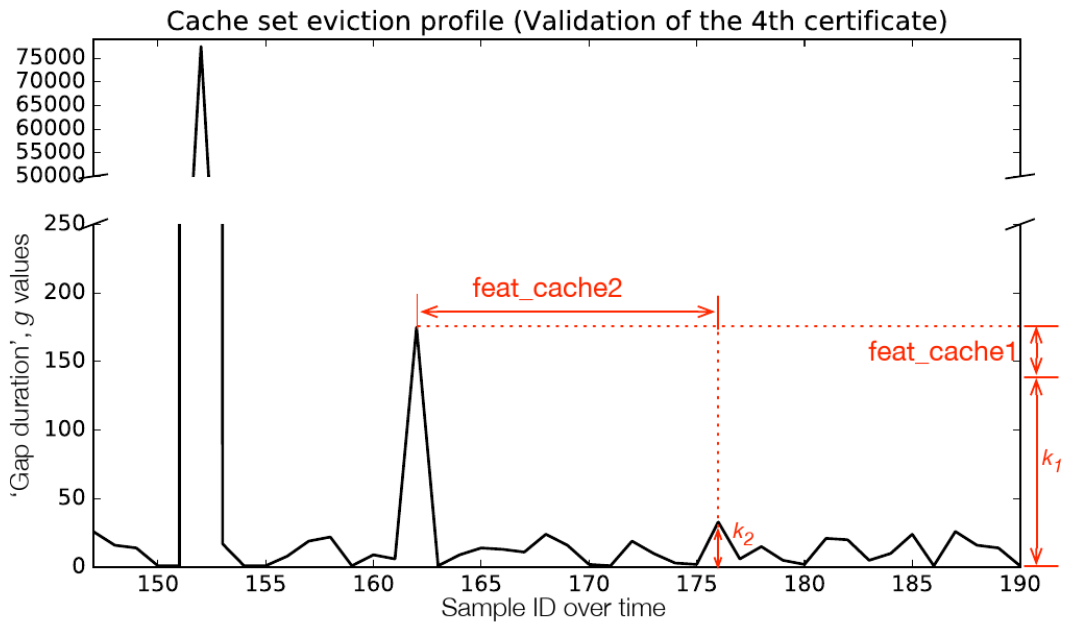


Figure 18: Profiling by Tang et al. [2] during loading of Keymaster

written in that range actually persist to the second shellcode, so we conclude that the XPU protection is indeed the cause of the behaviour we observe.

The reason that we could actually read the written data even though the area is write-protected could be because of caching or reading from micro-architectural buffers.

Since we cannot write to the relevant part of the TrustZone Kernel, we are unable to use this method of instrumentation.

9. Discussion

9.1. Bare metal

In this work we have used a bare-metal environment. As stated before, this eliminates noise sources like scheduling and the operating system. The advantages of this noise reduction can be seen in table 1. The measurements on Android have a lower eviction rate, longer eviction time and need more addresses and iterations in the eviction strategy. The drawbacks of using our bare-metal environment is that several initializations, which are normally handled by the Linux kernel, now have to be implemented in bare-metal. Since little is document about the drivers for the APQ8084, most of the implementation details have to be obtained by studying the Linux kernel. The driver implementations in Linux use many frameworks, which makes it hard to extract the relevant code. As a result, the initializations in the bare-metal environment may behave differently from the initializations in Linux, which interferes with reproducing existing research. Also, since most research is done on Android, using existing implementations is harder, because those implementations are not necessarily compatible with an environment without `libc` or system calls.

9.2. L2 cache inclusiveness to instructions

We tested the inclusiveness of the L2 cache with respect to the L1-D and L1-I caches. We found, contrary to the work of X. Zhang et al. [14], that the L2 cache is not inclusive to the L1-I cache. The main difference between our work and the work by X. Zhang et al. [14] is that we use a bare-metal environment in which we configured the MMU, while X. Zhang et al. perform the experiments on Android. Other works [2, 18] also successfully perform instruction-based side-channel attacks on the Nexus 6. This suggests that our work is the one deviating. So, either there is a problem in our testing methodology or implementation, or the kernel configuration may have an influence on the L2 inclusiveness.

9.3. AES attacks

Flush+Reload profiling attacks were performed on two normal-world AES implementations [40, 33]. The boundaries between the different rounds could be observed. In a real-world scenario, Flush+Reload cannot be used to profile based on the round key, since that requires shared memory on the round keys. However, these attacks demonstrate a profiling method that can be used with other side-channels that do not have this requirement.

The resolution of our Prime+Probe attack on OpenSSL is too low to be useful for profiling, however there are some methods to increase this resolution, such as different core speeds for the attacker and victim, or attacking a specific alignment of the round keys. In future work these methods could be attempted. A Prime+Probe attack could be used to attack a software AES implementation in TrustZone.

In addition to the profiling attacks, a method to detect the location of AES T-Tables using Prime+Probe was shown (section 7.2.4). This method is similar to how Liu et al. [12] find the relevant cache sets for their attack on ElGamal. The difference is that they measure a complete trace of each cache set while we only check after the AES execution whether the line was used. This results in an attack of lower granularity, since we cannot see temporal patterns. However, we have shown that it is clear enough to detect the T-Tables and also the round keys of the AES implementation.

9.4. RSA attacks

Since RSA is much slower than AES, using PRIME+PROBE for profiling RSA results in a resolution that is high enough to get a relevant trace. We were able to perform over 100000 measurements during a single RSA execution, while for OpenSSL AES we could perform only two measurements in a single execution. For our attack on the RSA implementation used to verify trustlet certificates, we created traces for each cache set. All traces show usage in the same five locations, while the parts between those five locations usually have no cache usage by the victim. We have discussed the possibility for flushes to be the cause of those slower probe times, but we deem that unlikely considering the vast number of flushes that would be required. Some traces, like for cache set 1280, show usage outside of the five hot spots. By examining the TrustZone kernel, we determined that the deviating cache sets do not correspond to locations within the TrustZone RSA implementation.

Another possible explanation for the similarity in traces for all cache sets is that we might be measuring the effects of another side channel. Note that we measure the length of performing the probe step of PRIME+PROBE. It might be the case that the probing time is longer for reasons other than cache misses, such as scheduling of the secure world. This would explain that all traces have higher probe timings at the same temporal location.

We verified that the PRIME+PROBE measurements are a side channel into TrustZone, by observing a difference in execution length with a corrupted Keymaster binary and by checking that we do not measure cache usage when the victim executes NOP instructions. So, we are sure that the higher probe timings are caused by the TrustZone execution.

Even though we do not know the exact cause for the higher probe timings, we can use it as a side channel to profile TrustZone. Our gap value plots match the descriptions given by Tang et al. [2] and our plots of the validation of the 4th certificate show the same pattern as the plot by Tang et al. [2] (fig. 18). To know for sure that the results match, experiments have to be conducted to see if FLIPENDIANNESS indeed runs right after the smaller spike that we observe in fig. 17.

9.5. Cross-core vs. scheduling

The AES and RSA attacks mentioned above were executed in a cross-core scenario. Either scheduling or execution on a second core had to be implemented in the bare-metal environment to have the attacker and victim run simultaneously. The drawback of using scheduling is that it does not work when profiling a TrustZone execution, since after executing an SMC call, the normal world has to wait for the secure world to be done executing. Another drawback is that a self-implemented scheduler would work different from the Linux scheduler, which results in more differences with previous research.

Executing code on a second core is supported by the hardware, therefore the differences with Linux are minimized. In addition, while an SMC call is running on a second core, the first core can continue executing. So, a profiling attack on TrustZone requires executing code on a second core.

10. Conclusion

We have provided an overview of the state-of-the art of ARM cache attacks. Several attacks have been described in detail to enable the reader to obtain a good understanding of the subject before reading the individual papers. In addition to the attacks on the ARM platform, several speculative execution attacks have been described [6, 7, 8, 10, 37, 9]. These attacks are not directly applicable to ARM, but similar attacks could be possible, either currently or in the future. For example, Spectre ([6]) has already successfully been applied to ARM devices and Meltdown ([7]) could theoretically work on ARM.

We have shown FLUSH+RELOAD profiling attacks on two AES implementations [40, 33]. The boundaries between the different rounds can be observed for both of the implementations.

We have shown a PRIME+PROBE side channel on an RSA implementation running in TrustZone. Our results seem to match the results by Tang et al. [2], however, more research is necessary to verify this.

All our attacks were executed in a bare-metal environment, which reduces noise since no operating system is running in the background. Our work on this environment leads to more public insight in how the Nexus 6 hardware works, notably, symmetric multiprocessing (SMP), MMU initialization and accessing the hardware PRNG.

The attacks that have been shown could be used as a method to determine a timing anchor for fault-channel attacks, similar to work by Tang et al. [2] and Qiu et al. [18]. Or the attacks could be used to obtain more information about execution flow in TrustZone.

In a forensic setting, practical and reproducible attacks are important, this research can be a starting point for fault-injection attacks with runtime profiling in a bare-metal environment. This hopefully makes such attacks more reliable and reproducible.

11. Future work

This section describes steps that could be taken following this work.

Perform Prime+Probe profiling on AES Our FLUSH+RELOAD attack on AES could be used as the basis for a PRIME+PROBE profiling attack. The resolution could be increased by using a different core speed for the attacker and the victim so more measurements can be done during a single execution.

If only a specific round has to be found, i.e. the 7th round, a specific division of the round keys over the cache lines could make it possible to detect the 7th round as the first access to the second cache line as described in section 7.2.3.

Verify that our Prime+Probe measurements match the results of Tang et al. [2] Experiments could be performed to detect if FLIPENDIANNESS runs right after the smaller spike in fig. 17. Tang et al. [2] performed such experiments but we could not find a description of them.

Find the exact cause of the difference in L2 instruction-inclusiveness compared to previous research by X. Zhang et al. [14] We suspect that a different MMU initialization results in our deviating results, however it could also be a methodology or implementation error.

Perform a fault-injection attack based on this runtime profiling information, possibly from bare-metal Since we provide an implementation of runtime profiling, this can be used in subsequent work to perform attacks to determine secret keys or to load arbitrary trustlets.

Determine whether there is a software AES implementation in Qualcomm's TrustZone We were unable to find a software AES implementation in Qualcomm's TrustZone, however, we have not done an exhaustive review. Future work could examine the Keymaster trustlet and the TrustZone kernel to find exactly how the used AES implementations work, i.e. if they use the crypto engine or are implemented in software. The work by Lapid and Wool [16] could be used as a starting point.

Extend the bare-metal environment to work on other devices than the Nexus 6 The current implementation is tailored towards the Nexus 6 with respect to the cache sizes and hardware initialization. This could be made more general in order to run it on other devices.

12. References

References

[1] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. Armageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 549–564, 2016.

[2] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW: exposing the perils of security-oblivious energy management. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1057–1074, 2017.

[3] CLKscrew: Exposing the perils of security-oblivious energy management. `https://github.com/0x0atang/clkscrew`. Commit: 712f75dab260da9b81f9d95f3debf3460663d183.

[4] Ben Lapid and Avishai Wool. Cache-attacks on the arm trustzone implementations of aes-256 and aes-256-gcm via gpu-based analysis. In *International Conference on Selected Areas in Cryptography*, pages 235–256. Springer, 2018.

[5] Android full-disk encryption. `https://source.android.com/security/encryption/full-disk`. Accessed: June 2020.

[6] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[7] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, 2018.

[8] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, August 2018. See also technical report Foreshadow-NG [44].

[9] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 753–768, 2019.

[10] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *S&P*, May 2019.

[11] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. CacheOut: Leaking data on Intel CPUs via cache evictions. `https://cacheoutattack.com/`, 2020.

[12] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE symposium on security and privacy*, pages 605–622. IEEE, 2015.

[13] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y Thomas Hou. Truspy: Cache side-channel information leakage from the secure world on arm devices. *IACR Cryptology ePrint Archive*, 2016:980, 2016.

[14] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. Return-oriented flush-reload side channels on arm and their implications for android devices. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 858–870, 2016.

[15] Marc Green, Leandro Rodrigues-Lima, Andreas Zankl, Gorka Irazoqui, Johann Heyszl, and Thomas Eisenbarth. Autolock: Why cache attacks on ARM are harder than you think. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1075–1091, 2017.

[16] Ben Lapid and Avishai Wool. Navigating the samsung trustzone and cache-attacks on the keymaster trustlet. In *European Symposium on Research in Computer Security*, pages 175–196. Springer, 2018.

[17] Haehyun Cho, Penghui Zhang, Donguk Kim, Jinbum Park, Choong-Hoon Lee, Ziming Zhao, Adam Doupé, and Gail-Joon Ahn. Prime+count: Novel cross-world covert channels on arm trustzone. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 441–452, 2018.

[18] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. Voltjockey: Breaching trustzone by software-controlled voltage manipulation over multi-core frequencies. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 195–209, 2019.

[19] Yangdi Lyu and Prabhat Mishra. A survey of side-channel attacks on caches and countermeasures. *Journal of Hardware and Systems Security*, 2(1):33–50, 2018.

[20] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 8(1):1–27, 2018.

[21] Nexus 6 firmware version ngi77b. `https://dl.google.com/dl/android/aosp/shamu-ngi77b-factory-5cd75e2a.zip`. Accessed: June 2020.

[22] Nexus 6 bare metal. `https://github.com/JJK96/nexus6-baremetal-cache-attacks`.

[23] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross processor cache attacks. In *Proceedings of the 11th ACM on Asia conference on computer and communications security*, pages 353–364, 2016.

[24] Gal Beniamini. QSEE privilege escalation vulnerability and exploit (cve-2015-6639). `http://bits-please.blogspot.com/2016/05/qsee-privilege-escalation-vulnerability.html`, 2016.

[25] Sandro Pinto and Nuno Santos. Demystifying arm trustzone: A comprehensive survey. *ACM Computing Surveys (CSUR)*, 51(6):1–36, 2019.

[26] ARM cortex-a series programmer's guide for ARMv8-a. `https://developer.arm.com/documentation/den0024/latest/`. Accessed: April 2020.

[27] Onur Acıiçmez and Çetin Kaya Koç. Trace-driven cache attacks on aes (short paper). In *International Conference on Information and Communications Security*, pages 112–121. Springer, 2006.

[28] Raphael Spreitzer and Benoît Gérard. Towards more practical time-driven cache attacks. In *IFIP International Workshop on Information Security Theory and Practice*, pages 24–39. Springer, 2014.

[29] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+ flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.

[30] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers track at the RSA conference*, pages 1–20. Springer, 2006.

[31] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer. js: A remote software-induced fault attack in javascript. In *International conference on detection of intrusions and malware, and vulnerability assessment*, pages 300–321. Springer, 2016.

[32] Joan Daemen and Vincent Rijmen. The rijndael block cipher: Aes proposal. In *First candidate conference (AeS1)*, pages 343–348, 1999.

[33] OpenSSL AES core. `https://github.com/openssl/openssl/blob/master/crypto/aes/aes_core.c`, . Commit: c72fa2554f5adc03bcc3c6e4ebcd1929e70efed4.

[34] Michael Tunstall, Debdeep Mukhopadhyay, and Subidh Ali. Differential fault analysis of the advanced encryption standard using a single fault. In *IFIP international workshop on information security theory and practices*, pages 224–233. Springer, 2011.

[35] `https://www.qualcomm.com/products/snapdragon-processors-805`. Accessed: August 2020.

[36] Sean Beaupre. Trustnone: Signed comparison on unsigned user input leading to arbitrary read/write capabilities of secure memory/registers in the apq8084/snapdragon 805 trustzone kernel. `http://theroot.ninja/disclosures/TRUSTNONE_1.0-11282015.pdf`, 2015.

[37] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, et al. Fallout: Leaking data on meltdown-resistant cpus. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 769–784, 2019.

[38] ARM architecture reference manual. *ARMv7-A and ARMv7-R edition*, 2012.

[39] Eviction strategy evaluator. `https://github.com/IAIK/armageddon/tree/master/eviction_strategy_evaluator`. Commit: f9a96c388225e22f19f446be99f892b2f17619b6.

[40] Tiny AES in c. `https://github.com/kokke/tiny-AES-c`. Commit: 0677e48a4980cc3695bc0f4dab89bad8708d16ea.

[41] OpenSSL RSA. `https://github.com/openssl/openssl/blob/master/crypto/rsa/rsa_ossl.c`, . Commit: 8a5cb59601fa9892e22e26337917cf513d57c473.

[42] Gal Beniamini. Extracting qualcomm's keymaster keys - breaking android full disk encryption. `http://bits-please.blogspot.com/2016/06/extracting-qualcomms-keymaster-keys.html`, 2016.

[43] Nexus 6 firmware version lmy48m. https://dl.google.com/dl/android/aosp/shamu-lmy48m-factory-25210911.zip. Accessed: June 2020.

[44] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. *Technical report*, 2018. See also USENIX Security paper Foreshadow [8].

13. Acronyms

ABoot	Android bootloader
ACL	access control lists
ASLR	address space layout randomization
BTB	branch target buffer
BTB	branch-target buffer
CPSR	current program status register
DCCISW	data cache clean and invalidate by set/way
DFA	differential fault analysis
DSB	data synchronization barrier
DVFS	dynamic voltage and frequency scaling
EL	exception layer
FBE	file-based encryption
FDE	full-disk encryption
GCM	Galois counter mode
I/O	input/output
ISA	instruction set architecture
ISB	instruction synchronisation barrier
L1-D	level 1 data
L1-I	level 1 instruction
L1	level 1
L2	level 2
LDR	load register
LFB	line fill buffer
LLC	last-level cache
LRU	least recently used
MDS	microarchitectural data sampling
MMU	memory management unit
MRC	move to ARM register from coprocessor
NOP	no operation
PBL	primary bootloader
PC	program counter
PIPT	physically indexed, physically tagged
PMCCNTR	cycle count register

PMCNTENSET	performance monitors count enable set register
PMCR	performance monitors control register
PMSELR	performance monitors event counter selection register
PMU	performance monitoring unit
PMXEVTYPER	performance monitors event type select register
PRNG	pseudo-random number generator
ROP	return oriented programming
SBL	secondary bootloader
SB	store buffer
SCR	secure configuration register
SCTLR	system control register
SGX	secure guard extensions
SMC	secure monitor call
SMP	symmetric multiprocessing
SoC	system on chip
SP	stack pointer
TCI	trustlet connector interface
TLB	translation lookaside buffer
TSX	transactional memory extension
TTBR	translation table base register
TZASC	TrustZone address space controller
TZMA	TrustZone memory adapter
TZPC	TrustZone protection controller
WTF	write transient forwarding
XPU	external protection unit

A. Code snippets

Algorithm 4: Address to cache set

```
input: Address
(Address >> $\log_2$(CACHE_LINE_LENGTH)) %
  NUM_CACHE_SETS
```

Algorithm 5: RSA decryption in TrustZone [2]

```
procedure DecryptSig($S$, $e$, $N$)
    $r \leftarrow 2^{2048}$;
    $R \leftarrow r^2 mod N$;
    $r^{-1} \leftarrow$ ModInverse($r, N_{rev}$);
    $N_{rev} \leftarrow$ FlipEndianness($N$);
    found_first_one_bit ← false;
    for $i \in \{bitlen(e) - 1..0\}$ do
        if found_first_one_bit then
            $x \leftarrow$ MontMult($x, x, N_{rev}, r^{-1}$);
            if e[i] == 1 then
                $x \leftarrow$ MontMult($x, a, N_{rev}, r^{-1}$);
            end
        end
        else if e[i] == 1 then
            $S_{rev} \leftarrow$ FlipEndianness($S$);
            $x \leftarrow$ MontMult($S_{rev}, R, N_{rev}, r^{-1}$);
            $a \leftarrow x$;
            found_first_one_bit ← true;
        end
    end
    $x \leftarrow$ MontMult($x, 1, N_{rev}, r^{-1}$);
    $H \leftarrow$ FlipEndianness($x$);
    return $H$;
end
```

Algorithm 6: Reverse the endianness of a memory buffer [2]

```
procedure FlipEndianness($src$)
    $d \leftarrow 0$
    $dst \leftarrow \{0\}$
    for $i \in \{0 .. len(src)/4 - 1\}$ do
        for $j \in \{0 .. 2\}$ do
            $d \leftarrow (src[\mathrm{i} * 4 + \mathrm{j}] \mid d) \ll 8$
        end
        $d \leftarrow src[\mathrm{i} * 4 + 3] \mid d$
        $k \leftarrow \mathrm{len(src) - i * 4 - 4}$
        $dst[\mathrm{k} .. \mathrm{k} + 3] \leftarrow d$
    end
    return $dst$
end
```