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

Ben Lapid and Avishai Wool(✉)

School of Electrical Engineering, Tel Aviv University, Tel Aviv, Israel
ben.lapid@gmail.com, yash@eng.tau.ac.il

Abstract. The ARM TrustZone is a security extension helping to move the "root of trust" further away from the attacker, which is used in recent Samsung flagship smartphones. These devices use the TrustZone to create a Trusted Execution Environment (TEE) called a Secure World, which runs secure processes called Trustlets. The Samsung TEE is based on the Kinibi OS and includes cryptographic key storage and functions inside the Keymaster trustlet.

Using static and dynamic reverse engineering techniques, we present a critical review of Samsung's proprietary TrustZone architecture. We describe the major components and their interconnections, focusing on their security aspects. During this review we identified some design weaknesses, including one actual vulnerability. Next, we identify that the ARM32 assembly-language AES implementation used by the Keymaster trustlet is vulnerable to cache side-channel attacks. Finally, we demonstrate realistic cache attack artifacts on the Keymaster cryptographic functions, despite the recently discovered Autolock feature on ARM CPUs.

1 Introduction

1.1 Motivation

The ARM TrustZone [3] is a security extension helping to move the "root of trust" further away from the attacker. TrustZone is a separate environment that can run security dedicated functionality, parallel to the OS and separated from it by a hardware barrier.

Recent Samsung flagship smartphones rely on Samsung's Exynos SoC architecture cf. [28]. This architecture incorporates an ARM CPU, as well as a GPU, memory and peripherals. The ARM cores in Exynos support the TrustZone security extension to create Trusted Execution Environments (TEEs). On their Exynos-based platforms, Samsung uses Trustonic's Kinibi OS as the Secure World kernel.

These TEEs are often used in scenarios which require a higher level of security or privacy guarantees, such as application of cryptographic functions, secure payments and more. Therefore, these environments present a high value target

© Springer Nature Switzerland AG 2018
J. Lopez et al. (Eds.): ESORICS 2018, LNCS 11098, pp. 175–196, 2018.
https://doi.org/10.1007/978-3-319-99073-6_9

for attackers. However, the security practices in these environments were not thoroughly studied by the research community yet.

In order to support cryptographic modules, the Android OS includes a mechanism for handling cryptographic keys and functions called the Keystore [11]. Keystore is used for several privacy related features such as full disk encryption and password storage. The Keystore depends on a hardware abstraction layer module to implement the underlying key handling and cryptographic functions; and many OEMs, including Samsung, choose to implement this module using the TrustZone.

1.2 Related Work

Lipp et al. [16] implemented cache attack techniques to recover secret keys from Java implementation of AES-128 on ARM processors, and exfiltrate additional execution information. In addition they were able to monitor cache activity in the TrustZone.

Zhang et al. [38] demonstrated a successful cache attack on a T-Table implementation of AES-128 that runs inside the TrustZone—however, their target was a development board lacking a real Secure World OS rather than a standard device. Ryan et al. [18] demonstrated reliable cache side-channel techniques that require loading a kernel module into the Normal World—which is disabled or restricted to OEM-verified modules on modern devices. To our knowledge no previous cache attacks on ARM TrustZone have been published on standard devices using publicly available vulnerabilities.

Recently, Green et al. [14] presented AutoLock, an undocumented feature in certain ARM CPUs which prevent eviction of cross-core cache sets. This feature severely reduces the effectiveness of cache side-channel attacks. The authors listed multiple CPUs that include AutoLock, and among them are the A53 and A57 used in the device we used (Samsung Galaxy S6).

Cache side-channel attacks on AES were first demonstrated by Bernstein [5] with the target being a remote encryption server with an x86 CPU. Osvik et al. [25] demonstrated the *Prime+Probe* technique to attack a T-Table implementation of AES which resides in the Linux kernel on an x86 CPU. Xinjie et al. [37] and Neve et al. [19] presented techniques which improve the effectiveness of cache side-channel attacks. Spreitzer et al. [31] demonstrated a specialization of these attacks on misaligned T-Table implementations. Neve et al. [20] discussed the effectiveness of these attacks on AES-256 and demonstrated a successful specialized attack for AES-256.

Little is publicly known about the design and implementation of the proprietary closed-source Kinibi OS [32] used as a Secure World by Samsung.

1.3 Contributions

Our first contribution is a critical review of Samsung's TrustZone architecture on the Exynos SoC platform, including the Kinibi OS. Through a combination of

firmware disassembly, open-source code review and dynamic instrumentation of system processes, we are able for the first time to provide a description of all the major subsystems, with their interconnections and communication paths, of this complex proprietary system. Our review focuses on the security aspects of the architecture, and in particular on the Keymaster trustlet, which is responsible for many critical cryptographic functions. During this review we identified some design weaknesses, including one actual vulnerability.

Our next contribution is identifying that the ARM32 assembly-language AES implementation used by the Keymaster trustlet is vulnerable to cache side-channel attacks. We also identify that the Keymaster uses AES-256 in GCM mode. In a separate paper [15] we show successful cache attacks against the implementation.

Our final contribution is demonstrating realistic cache attack artifacts on the Keymaster cryptographic functions embedded in the Secure World and protected by the ARM TrustZone architecture. Contrary to prior assumptions, we found that the cache is *not* flushed upon entry to the Secure World. On the other hand, the recently discovered "AutoLock" ARM feature is a serious limitation. Nonetheless, we are able to successfully demonstrate cache side-channel effects on "World Shared Memory" buffers, and we show compelling evidence that full-blown cache attacks against the AES implementation inside the Keymaster trustlet are plausible.

Organization: In the next section we introduce some background about the ARM TrustZone and its use in Android. Section 3 describes our discoveries about the Exynos secure boot and the Kinibi secure OS. Section 4 describes the Normal World components interfacing with the secure OS. Section 5 describes our achievements in mounting cache attacks against the Keymaster trustlet, and we conclude with Sect. 6.

2 Preliminaries

2.1 ARM TrustZone Overview

ARM TrustZone security extensions [4] enable a processor to run in two states, called Normal World and Secure World. This architecture extends the concept of "privilege rings" and adds another dimension to it. In the ARMv8 ISA, these rings are called "Exception Levels" (ELs). The most privileged mode is the "Secure Monitor" which runs in EL3 and sits "above" the Secure and Normal Worlds. In the Secure World, the Secure OS kernel runs in EL1 and the Secure userspace runs in EL0. In the Normal World, an optional hypervisor may be run in EL2, the Normal OS kernel runs in EL1 and the Normal userspace runs in EL0. On the Galaxy S6 there is no hypervisor, and the Normal World OS is Android.

The separation of Secure and Normal World means that certain RAM ranges and bus peripherals may be indicated as "secure" and only be accessed by the Secure World. This means that compromised Normal World code (in userspace,

kernel or hypervisor) will not be able to access these memory ranges or devices and thus pose a threat to them as well.

To allow a controlled method of passing information between the worlds, a mechanism called "World Shared Memory" allows memory pages to be accessible by both worlds. These physical memory pages reside in the Normal World, and the Secure World maps them into its processes' virtual memory as needed.

Additionally, communication may be initiated between worlds by means of SMC calls. SMC calls are basically "system calls" made by a kernel in EL1 or EL2 (either Secure or Normal) to the EL3 "Secure Monitor". These SMCs, use the "Secure Monitor" to pass information between the worlds. In particular, a common SMC is used by one world to notify the other of pending work; such SMC is implemented in the "Secure Monitor" by triggering a software interrupt in the other world. Note that ARM CPUs also have SVC calls: regular system calls from EL0 to EL1 within the same world.

It is important to note that the world separation is completely "virtual". The same cores are used to run both Secure and Normal Worlds and they use the same RAM. Therefore, they use the same cache used by the core to improve memory access times; as we shall see in Sect. 5.3, this design decision may be used to mount cache side-channel attacks.

2.2 TrustZone Usage in Android

In the Samsung/Android ecosystem, there are two major players in field of TrustZone implementations. One is Qualcomm, with the QSEE operating system [27] which is compatible with the Snapdragon SoC architecture used on many Samsung devices. The other is Trustonic, with the Kinibi operating system [32] which is used by Samsung in their popular Exynos SoC architecture as a part of the KNOX security system [29].

These Trusted Execution Environments (TEEs) are used for various activities within the smart device: Secure boot (see Sect. 3), Keymaster implementation (see Sect. 4.4), secure UI, kernel protections, secure payments, digital rights management (DRM) and more. Because their usage is often linked to security of privacy-critical applications, they are a high-value target. In our research we focused on the Trusted Execution Environment present in Samsung's Exynos SoC (in particular in Samsung's Galaxy S6): Secure Boot, Trustonic's Kinibi OS, Trusted Drivers and Trustlets.

2.3 Attack Model

The fundamental reason for the existence of the TrustZone is to provide a hardware-based root of trust for a trusted execution environment (TEE)—that is designed to resist even a compromised Normal World kernel.

Since the Normal World kernel, and all the kernel modules on Samsung's smartphones are signed by Samsung and verified before being loaded, injecting code into the kernel is challenging for the attacker. Our goal in this work is

to demonstrate that weaker attacks, that do not require a compromised kernel, are sufficient to exfiltrate Secure World information—in particular secret key material.

In our attack mode we assume an attacker is able to execute code on a Samsung Galaxy S6 device, under **root privileges** and relevant **SELinux permissions**. Note that these privileges are significantly less than kernel privileges, since the attack code runs in EL0.

Root privileges are needed to access the */proc/self/pagemap* to identify cache sets, as described by Lipp et al. [16]. Our attack can theoretically be mounted without access to this file, but it will be substantially more difficult. SELinux permissions are needed to connect to the *mcDriverDaemon* process (see Sect. 4.2) through the Unix domain socket, and to access the */dev/mobicore* device (see Sect. 4.1), as Samsung's Keymaster HAL module uses these interfaces to load and communicate with the trustlet (see Sect. 4.4).

To achieve root privileges and the necessary SELinux permissions in our investigation we used the publicly known vulnerability called *dirtycow*. The rooting process is based on Trident [6], which uses *dirtycow*.

3 The Exynos Secure World Components

In our research we explored the inner workings of the trusted execution environment implemented in Samsung's Exynos SoC platform [28]. This platform is present in many of its flagship phones; of which we focused on the Galaxy S6. Several security researchers have previously presented different pieces of information about the TEE in this environment, but to our knowledge there is no publication which covers the TEE in a systematic manner. This section describes our findings regarding the platform's *Secure Boot* mechanism (which includes a series of bootloaders, the trusted OS and several trustlets). In Sect. 4 we describe how the Normal World OS (Android Linux) communicates with the secure OS.

Secure Boot (sboot). We started our exploration by reverse-engineering firmware images for the Galaxy S6 smartphone. We observed that these images contain several distinct files, including the Android Linux image, the *system* partition, the *Secure Boot* partition and more. Samsung does not provide much information about the *Secure Boot* apart from one short page [29]. According to that page, the boot process consists of a chain of bootloaders, starting with a primary bootloader which resides in read only memory, and each link of the chain verifies the next bootloader. Hence the remainder of this section is based on our own discoveries.

The *Secure Boot* partition lies within the *sboot.bin* file, of size 1.6 MB. Opening the file with a disassembler reveals several distinct parts. All of the parts seem to include a code segment and data segment, some are in ARM64 and some are in ARM Thumb mode. In our research we identified them as follows:

- EL3 bootloader and Monitor Code (SMC handler) (ARM64).
- Normal World bootloader (ARM64).
- The Kinibi Secure World operating system (ARM Thumb), which contains: the OS itself, Trustlet and Driver API library and what appears to be an *init*-like first user-land process.
- Three Secure World Drivers: *SecDrv*, *Crypto Driver* and *STH Driver* (ARM Thumb).

The EL3 Monitor. The first part in *sboot.bin* contains instructions which are reserved for EL3 execution only, such as setting the interrupt vector base and several other ARM special registers. While reverse-engineering this part, we found many similarities with ARM's reference implementation of TrustZone boot sequence. This lead us to conclude that the responsibilities of this part are: Architectural initialization, Platform initialization, Runtime services initialization and Normal World bootloader execution (See the ARM reference documents [1]).

Based on [21], we found that the registered runtime services (*rt_svc_desc_t* array [2]) gives us insight into what functionality is made available by the monitor code which runs in EL3.

It is important to note that the EL3 monitor binary is verified by an earlier bootloader and is responsible for verifying the binaries of the parts it loads: the Normal World bootloader and the secure OS.

The Normal World Bootloader. The second part we found in *sboot.bin* is the Normal World bootloader. This part runs in Normal World EL1 and has several responsibilities: booting the Android Linux kernel (after verifying its binary), requesting secure OS initialization from the monitor code, handling firmware flash requests ("Download mode"), handling "Recovery mode" requests and presenting relevant user interfaces for these modes. This part executes only on device start-up and therefore was less interesting to us. Others [8,21] have presented their research on this part.

The Kinibi Secure Operating System. The third part we found in *sboot.bin* is the Kinibi secure operating system which includes the OS, a user-space API library and an *init*-like user-space process. For the Exynos platform, Samsung has chosen to use Trustonic's Kinibi [32] as the base of their trusted execution environment. Note that Kinibi was previously called t-base or MobiCore; much of the internal naming still uses the "mobicore" name: e.g., the device */dev/mobicore* etc. Hence when we discuss the Kinibi internals we often use the name mobicore.

Surprisingly, we found that the binary code for the operating system runs in Thumb (32bit) mode even though the platform has a 64bit processor. Furthermore, we found that while the Kinibi OS is protected by the TrustZone architecture, internally it does not protect itself very well. Lacking were defenses

such as Address Space Randomization (ASLR), non-executable (NX) stack, or stack canaries, which are all present in stock Android since version 4.0. Our observations about the Kinibi OS are as follows:

- Privileges are separated to: OS code—which runs in Secure World EL1; Trusted Applications (or Trustlets)—which run in Secure World EL0 as processes and have access to a limited set of system calls; and Drivers—which run in Secure World EL0 and have access to a broader set of system calls.
- Kinibi supports processes and virtual memory isolation. In addition, Drivers may spawn additional threads.
- Kinibi uses a priority based scheduler. Time quanta are made available by having the Normal World issue specific SMCs which are transfered to the Secure World OS. Without them, the secure OS would not run at all. Two methods of entry are available after initialization: SIQ - which signals the Kinibi OS that an interrupt (or an asynchronous notification) was issued by the Normal World and needs to be handled; and Yield - which means the secure OS may continue any work it chooses.
- Processes may request memory allocation. Furthermore, Drivers may request memory mapping to physical memory for integration with platform devices.
- Kinibi supports *World Shared Memory* for communication between Normal World and the Secure World—recall Sect. 2.1. In particular, Kinibi uses World Shared Memory to define the TCI (Trustlet Connector Interface) memory, which plays an important part of our research, see Sect. 5.3.
- Kinibi supports inter (secure)-process RPC-like communication. Trustlets may send requests to Drivers and receive responses via a message queue. Requests and responses are routed by an IPCH (covered below) which receives the requests from Trustlets and routes them to Drivers and vice versa. Furthermore, a notification system is supported which allows Drivers and Trustlets to wait until the Normal World has issued them a notification.
- Kinibi supports a circular buffer logging mechanism which can be read by the Normal World.

It is important to note that Kinibi OS is bound to a specific CPU core (which can be changed at runtime), and discards interrupts issued on other cores: On our device, Kinibi boots on core 0 and is later switched by default to core 1.

Analyzing the Kinibi OS reveals several distinct segments: (i) the interrupt vector base, interrupt handlers and the OS kernel initialization code; (ii) a user-space code which appears to be a shared library that is injected into Trustlets and Drivers and presents an interface to the OS. (iii) the rest of the OS kernel code; and (iv) an *init*-like secure-world user-land process which is spawned at OS kernel initialization. We omit the details.

Kinibi Drivers. The fourth part of *sboot.bin* consists of three Secure World Drivers: *SecDrv*, *Crypto Driver* and *STH Driver*. We note that the crypto driver implements various cryptographic functions over an IPC mechanism—however the Keymaster trustlet we discuss in Sect. 4.4 includes its own cryptographic implementations. We omit the details.

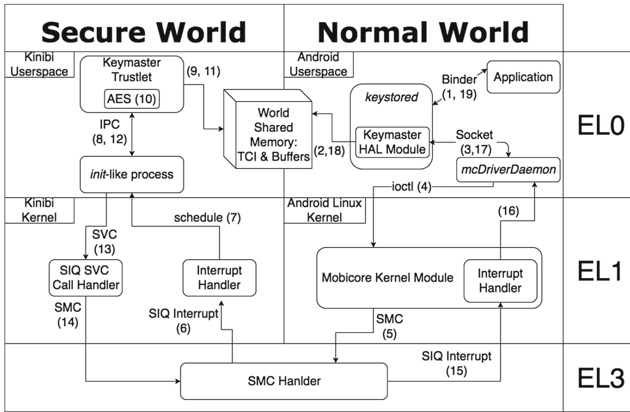

Fig. 1. Secure World/Normal World layering around the Keymaster trustlet. TCI stands for Trustlet Connector Interface, SIQ for Software Interrupt Queue. The numbers in parenthesis mark the actions illustrated in Appendix A.

4 The Exynos Normal World Components

In this section we explore the way the Normal World communicates with the Secure World and what APIs are made available to Android applications. We start by describing the MobiCore kernel module which implements the interface between the Secure World and Normal World users (other kernel modules and user-land processes). We then present our findings on the user-land process *mcDriverDaemon* and Samsung's implementation of the Keymaster HAL interface (see Fig. 1 as reference). In Appendix A we present an example of communication between the Normal World and the Secure World and trace the execution path between them.

4.1 The MobiCore Kernel Module

The MobiCore kernel module is statically linked into the Android Linux kernel image and is initialized on kernel startup. The module is licensed under "GPL V2" and therefore is open-source (source code can be found under many Android Kernel tree publications such as [9]). By reading the source code one can see that the module's responsibilities are:

– Register device files (*/dev/mobicore* and */dev/mobicore-user*) which allow user-space programs to interact with the driver (through *ioctl*, *read* and *mmap* syscalls). The *mobicore-user* device is used by user-land processes that wish to interact with the kernel module, and exposes a limited set of APIs (only mapping and registration of World Shared Memory). The *mobicore* device is used only by the *mcDriverDaemon*, is considered the *admin device* and allows for broader functionality such as: Initializing the MCI shared memory

(discussed in Sect. 4.2), issuing Yield or SIQ SMC calls, locking shared memory mappings and receiving notifications of interrupts from the Secure World OS. It is important to note that only **one** process may open the *mobicore* device at any point in time: if another process tries to open it, an error will be returned. Usually, the *mcDriverDaemon* opens this device first; however, if the *mcDriverDaemon* process dies for any reason, the next process to open the *mobicore* device will receive *admin* status as far as the kernel module is concerned. This means that an attacker within our attack model (recall Sect. 2.3) can hijack the *mobicore* device and act as the *admin*.
- Register an interrupt handler which receives completion notifications from the Secure World OS. These notifications are forwarded to the active daemon.
- In order to trigger interrupts to the right core (so that Kinibi OS will not discard them), the kernel module starts a dedicated thread which is bound to the core on which the Kinibi OS is running. This thread issues SMC calls requested by other processes.
- Perform additional tasks such as initializing and periodically reading log messages from the Secure World (via a work queue and a dedicated kernel thread), migrating the Secure OS to different CPU cores if needed, managing the World Shared Memory buffers that were registered by the Normal World, handling power management notifications, and suspending/resuming the Secure OS as needed.

4.2 The mcDriverDaemon Process

The *mcDriverDaemon* binary is located within the *system* partition of the device's firmware under */system/bin/mcDriverDaemon*. A version of the daemon source code is available online [36], however we noticed some discrepancies between the online version and the binary on our device (the device probably has a newer version). The binary is executed by *init* at system startup; it immediately opens the */dev/mobicore* device and receives *admin* status. We analyzed this daemon by conducting both static analysis (reading the source code) and dynamic analysis: We killed the original daemon and quickly executed it from a root shell with a LD_PRELOAD directive. This directive injected our library (which is based on *ldpreloadhook* [26]) into the process and allowed us to hook *libc* functions which the daemon is using. These hooks gave us execution traces and raw parameters used by the running daemon, and helped us understand its inner workings. By this method, we identified the following responsibilities:

- Initialize the MobiCore Communication Interface (MCI) through the MobiCore kernel module. This maps a virtual address range in the daemon's memory to a World Shared Memory which is accessible to the Secure OS (in particular to the secure *init*-like process). As mentioned above, this allows the daemon to access the Secure OS API: Opening/Closing Trustlets, Map and Unmap World Shared Memory, Suspend and Resume the Secure OS and more.
- Periodically allow the Secure OS time quanta by calling the Yield or SIQ *ioctl* which the kernel module implements as SMC calls.

– Create and listen on *netlink* and *abstract unix domain* (*"#mcdaemon"*) sockets as servers which act as an interface for other user-land processes. This interface has a defined protocol [34] for serializing requests and responses and implements the following API: General information requests, Open/Close TrustZone device, Open/Close Trustlets (via UUID or sent data), send a Notification to trustlets and register World Shared Memory with Trustlets. A client library is available [33] for other processes to easily use.
– The *mcDriverDaemon* creates an instance of the *File System Daemon* [35] (we omit the details).

In particular, when handling *openSession* commands from Normal World clients the command receives the Trustlet UUID as an argument. The *mcDriverDaemon* then looks for the correct Trustlet to load in the Normal World file system. The daemon has two locations it looks in: */system/app/mcRegistry* (which is a read-only partition and verified at boot by *dm-verity*) and */data/app/mcRegistry* (which is a read-write partition). This request is then passed to the Secure OS which (as mentioned in Sect. 3) verifies the Trustlet's binary structure and signature before loading it into the Secure World.

The ability to load files from the read-write partition was previously exploited [7] to load old versions of trustlets which had vulnerabilities in them; thereby "bringing the attack surface to the device".

4.3 Keystore and Keymaster Hardware Abstraction Layer (HAL)

The Android Keystore system [11], which was introduced in Android 4.3, allows applications to create, store and use cryptographic keys while attempting to make the keys themselves hard to extract from the device. The documentation advertises the following security features:

– Extraction Prevention: The keys themselves are never present in the application's memory space. The applications only know of *key-blobs* which cannot be used by themselves. The *key-blobs* are usually the keys packed with extra meta-data and encrypted with a secret key by the Keymaster HAL. In the Samsung implementation we explored, the keys are bound to the secure hardware controlled by the Kinibi OS, which makes them even harder to extract: the keys themselves never leave the secure hardware unencrypted.
– Key Use Authorizations: The Keystore system allows the application to place restrictions on the generated keys to mitigate the possibility of unauthorized use. Restrictions include the choice of algorithms, padding schemes, and block modes, the temporal validity of the key, or requiring the user to be authenticated for the key to be used.

The Keystore system is implemented in the *keystored* daemon [12], which exposes a binder interface that consists of many key management and cryptographic functions. Under the hood, the *keystored* holds the following responsibilities:

– Expose the binder interface, listen and respond to requests made by applications.

- Manage the application keys. The daemon creates a directory on the filesystem for each application; the key-blobs are stored in files in the application's directory. Each key-blob file is encrypted with a key-blob encryption key (different per application) which is saved as the *masterkey* in the application's directory. The *masterkey* file itself is encrypted when the device is locked, and the encryption employs the user's password and a randomly generated salt to derive the *masterkey* encryption key.
- Relay cryptographic function calls to the Keymaster HAL device (covered below).

The Keymaster hardware abstraction layer (HAL) [10] is an interface between Android's *keystored* and the OEM implementation of a secure-hardware-backed cryptographic module. It requires the OEM to implement several cryptographic functions such as: key generation, init/update/final methods for various cryptographic primitives (public key encryption, symmetric key encryption, and HMAC), key import, public key export and general information requests. The implementation is a library that exports these functions and is implemented by relaying the request to the secure hardware runtime. The secure runtime usually encrypts generated keys with some key-encryption key (which is usually derived by a hardware-backed mechanism). Therefore, the non-secure runtime does not know the actual key that is used, but may still save it in the filesystem and subsequently use it through the Keymaster to invoke cryptographic functions with the key. In practice - this is exactly how the *keystored* daemon uses the Keymaster HAL (with the aforementioned addition of an additional encryption of the key blobs).

An example of the usage of the Keymaster HAL is the Android Full Disk Encryption feature, implemented by the userspace daemon *vold* [13], which uses the Keymaster HAL as part of the key derivation.

4.4 Samsung's Keymaster HAL and Trustlet

Samsung's Keymaster HAL library exposes the aforementioned Keymaster interface and implements its functions by making calls to the Keymaster trustlet (through *mcDriverDaemon*). The trustlet itself has UUID: *ffffffff00000000000000000000003e*, and is located in the system partition (*/system/app/mcRegistry/<UUID>.tlbin*). The Trustlet code handles the following tasks:

- Listen to various requests that are sent over the World Shared Memory and handle them.
- Key generation of RSA/EC, AES and HMAC keys. Keys are generated using random bytes from the OpenSSL FIPS DRBG module, which seeds its entropy either from *keymaster_add_rng_entropy* calls from the Normal World or from a secure PRNG made available by the Secure World Crypto Driver. Key generation requests receive a list of key characteristics (as defined by the Keymaster HAL), which describe the algorithm, padding, block mode

and other restrictions on the key. The generated keys (concatenated with their characteristics) are encrypted by a key-encryption key (**KEK**) which is unique to the Keymaster trustlet. The trustlet receives this key by making an IPC request along with a *constant* salt to a driver which uses a hardware-based cryptographic function to drive the key. The encryption used for key encryption is AES256-GCM128. The GCM IV and authentication tag are concatenated to the encrypted key before being returned to the user as a key blob. Therefore, an attacker that is able to obtain this KEK is able to decrypt all the key blobs stored in the file system—i.e., the KEK can be viewed as the "key to the kingdom", and is our target in the attacks in Sect. 5.

– Execution of cryptographic functions. The trustlet can handle begin/update/final requests for given keys created by the trustlet. It first decrypts the key-blobs and verifies the authentication tag, then verifies that the key (and the trustlet) supports the requested operation, and then executes it. The cryptographic functions are implemented using the OpenSSL FIPS Object Module [24]. In particular, we discovered that the AES code is a pure ARMv4 assembly implementation that uses a single 1KB T-Table. In general, AES implementations based on T-Tables are vulnerable to cache attacks [25]. However, as we shall see in Sect. 5, mounting the attack in practice is not trivial.
– The trustlet handles requests for key characteristics and requests for information on supported algorithms, block modes, padding schemes, digest modes and import/export formats.

Leaking the KEK Through Vulnerabilities in Other Trustlets. One of the many trustlets created by Samsung to provide secure computations to devices is the OTP trustlet. This trustlet implements a mechanism which creates One Time Passwords on the device. Exploiting a vulnerability in the OTP trustlet discovered by Beniamini [7], we were able to recover the Keymaster KEK. The OTP vulnerability gives us the ability to read and write 4-byte words into arbitrary OTP trustlet memory and branch execution to arbitrary OTP trustlet code. We used these primitives to imitate the way the Keymaster trustlet makes a request to derive the KEK: use the write primitive to fill the request struct and the fixed Keymaster salt (which we discovered via disassembly) into the OTP trustlet memory, then used the branch primitive to call a specific trustlet API function which is available on both the OTP and Keymaster code, and finally we used the read primitive to read the result—the KEK.

We argue that another trustlet's ability to imitate the Keymaster request and receive its KEK is a vulnerability in the API design and, in particular, in the driver that implements this request. Due to the lack of even basic mitigation techniques (ASLR, stack canaries, etc.) in the Kinibi OS and userspace, we believe more vulnerabilities may well be discovered in trustlets in the future. Therefore, critical keys, such as the Keymaster KEK, should be more protected. We propose a simple countermeasure: Have the handler of the key derivation IPC request concatenate the client UUID to the salt; this will prevent different

trustlets from deriving the same keys, and then a compromised trustlet will not immediately compromise the Keymaster KEK.

This vulnerability was reported to Samsung (CVE-2018-8774, SVE-2018-11792) on February 2018 and was labeled by Samsung as a "critical vulnerability." It was patched in Samsung's Android security update [30] in June 2018. In Sect. 5 we discuss an attack which aims at recovering the Keymaster key via a cache side-channel without relying on other trustlets being compromised.

5 Attacking the Keymaster Trustlet

Since Secure World computations (such as the AES implementation in the Keymaster trustlet) use the same cache as the Normal World, it is theoretically possible to mount cache attacks against the Secure World. Lipp et al. [16] suggested that the Samsung Galaxy S6 (which is built on the Exynos platform) flushes the cache when entering the TrustZone, thereby making the attack much more difficult. In contrast, we did not see any cache flushing operations when entering the TrustZone: none were present in the sources we reviewed or binaries we disassembled. Moreover, as we shall see, we were able to reliably infer execution information of Trustlets through cache side-channel artifacts. However, we encountered other hurdles. In this section we will discuss our proposed attack model, method and results.

5.1 The Target of the Attack

In our research we focused on recovery of the Keymaster KEK. Recovering this key would lead to compromise of all past, present and future Keystore keys and data encrypted by these keys on the device on which the attack was mounted on. The trustlet uses this key in several request handlers, which include: key generation, *begin* operation on keys and *get_key_characterstics*. Of these three, *get_key_characterstics* does the least amount work that's not related to key encryption; therefore we focused on this request. The request receives a buffer which should hold a key blob that consists of the encrypted key bytes and key characteristics followed by an IV and GCM authentication tag; the trustlet returns the key characteristics serialized in a buffer. Valid key blobs often include over 100 bytes of encrypted data (e.g., 32 key bytes of a stored AES-256 plus many required key characteristics), therefore the request uses the AES-256 block function at least 9 times (2 for initialization and at least 7 subsequent blocks). If we measure cache access effects only after the trustlet completes its work, the 9 block function invocations will induce too much noise and render our attacks infeasible. Therefore, instead we send *invalid* requests: having the key blob hold just one byte, a random IV, and zeros for the authentication tag. Such requests induce the two block function calls for initialization, and a single additional call to decrypt the single byte. The request then fails, therefore we do not have access to any ciphertext; but possibly, side-channel information may leak.

5.2 Challenges in Mounting the Attack

In our attempts at mounting the attack we encountered three major difficulties: (i) finding the cache sets which correspond to the trustlet's T-Table memory, (ii) Keymaster request execution times, (iii) facing AutoLock [14] behavior.

Searching for the T-Table. Before a cache attack can be mounted, the cache sets which correspond to the T-Table need to be identified. Our research suggests that the secure OS usually resides in either core 0 or 1 - both of them in the A53 CPU. The A53 CPU in the Galaxy S6 has a 256 KB L2 cache, with 64 byte cache lines and 16-way associativity; this means it has 256 different cache sets (8 bits used in set addressing). The index of a cache set is determined by the physical address of the memory which is being accessed. Because the cache lines are 64 bytes long, the 6 least significant bits are not used in the index calculation. Therefore the index calculation uses bits 6 through 13 of the address.

The T-Table used in the AES implementation inside the Keymaster trustlet is 256 4-byte entries long. We also know (through analysis of the trustlet binary) that the T-Table resides at virtual address *0x364c8*, so it is misaligned by 8 bytes, which means the T-Table spans 17 cache sets. We learn two things from this information: (a) the entire T-Table resides in a single page of memory and (b) that it starts at an offset of *0x4c8* inside the page. Knowing that the entire table resides in a single page ensures that its cache set indexes are **contiguous** (if it had spanned two pages, those pages could have been mapped to different physical pages, resulting in a potential discontinuity).

These points allow us to narrow down the possible cache set containing the *beginning* of the T-Table down to 4 options: Recall that the cache set index calculations use bits 6 through 13 of the physical address. The in-page offset (bits 0 through 11) of the physical address are equal to those in the virtual address, which we have. Therefore, only bits 12 and 13 remain unknown and the only candidates for the cache set index are: $\{19, 83, 147, 211\}$. Because we know the T-Table cache sets are contiguous, knowing the *beginning* cache set should give us complete information about the indexes of all the other sets.

A Synchronous Attack. Our initial attempts at discovering the T-Table location in the cache followed the *synchronous* attack model described by Osvik et al. [25]: *prime* the cache set candidates, call the AES encryption operation and then *probe* these cache sets and take measurements of the time it took to access them. Unfortunately, these measurements were too noisy. We noticed that the time it takes for the requests to complete is very long: 5–10 ms; this is enough time for many other processes to cause cache activity which taints our measurements.

An Asynchronous Attack. We then attempted to implement an *asynchronous* attack model. This technique *primes* and *probes* the cache sets in a loop on a

different core than the one which runs the secure OS. However, these measurements were not helpful either: the 17 contiguous cache sets following the result of the measurements did not present activity as expected of a T-Table. We believe the AutoLock feature described by Green et al. [14], is preventing us from making correct measurements with this approach since it blocks evictions that are induced by cache activity on a different core. Therefore, both attacks we described in this section failed to detect cache access effects that reveal the true cache set index of the T-Table.

5.3 Tracing Trustlet Execution Using Flush+Reload

Lipp et al. [16] also suggested using a *Flush+Reload* attack on ARM CPUs [16], which allows cache side-channel leakage of accesses of other processes to shared memory. While this attack is less relevant to leak information on the trustlet's T-Table, it is relevant to the "TCI memory". TCI memory is World Shared Memory which is accessible by both the Secure World and the Normal World. It is, in fact, a physical memory range which is mapped to virtual addresses in both the Normal World and the Secure World. Because the same underlying physical memory is shared, the *Flush+Reload* attack is relevant in leaking information about accesses to this memory by the Secure World.

Our disassembly of the Keymaster trustlet binary code points to three distinct World Shared Memory regions which are used by the trustlet. The first is the TCI memory itself, which contains the request identifier and pointers to two additional World Shared Memory buffers; the other two are the *input* buffer (filled by the Normal World) and the *output* buffer (filled by the Secure World). Upon receiving notifications of a pending request, the trustlet accesses the TCI memory, copies the relevant information from the input buffer to private memory, executes the request, if the request was successful it fills the output buffer, and finally fills the return code in the TCI memory. Therefore, by monitoring these three addresses with the *Flush+Reload* technique, we expect to see the following hit pattern: TCI → Input → Output(if successful) → TCI. Note that this pattern leaks fairly precise timing information about when the cryptographic operations take place within the 5–10 ms the request takes to complete: AES invocations occur after the input buffer is accessed and before the output buffer is accessed (or before the second TCI access on error).

Indeed, using this method we were able to recover timestamps of these events. Figure 2 shows multiple sets of timestamps recovered through this method. In the scenario illustrated by the figure we sent malformed requests and detected three events: 1st TCI access, Input access and finally a 2nd TCI access. Figure 2 shows the 1st TCI accesses (blue asterisks) happen around 2.5 ms into the measurement. This is followed by the Input access (red dots) about 1.5 ms later—we believe the delay is caused by the IPC requests the trustlet makes before handling the incoming request. Finally, about 30 µs after the Input access, we see the 2nd TCI access (black crosses). During this 30 µs period the encryption, along with the rest of the handler logic, takes place.

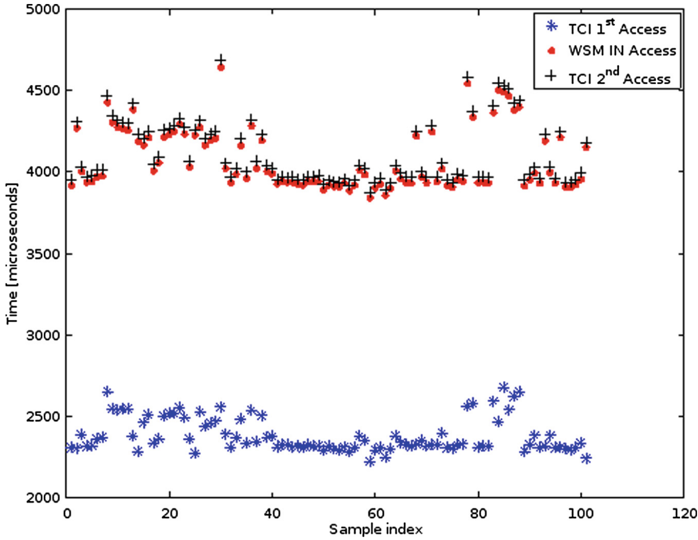

Fig. 2. Keymaster trustlet *world shared memory* (WSM) access timings (Color figure online)

These results strengthen our belief that leaking information from the Secure World is indeed possible through cache side-channel attacks.

5.4 Designing an Improved Attack

Moghimi et al. [17] demonstrated CacheZoom, an attack on Intel's secure execution environment - SGX. They use kernel mode privileges to trigger multiple clock interrupts while a secured computation is executed; these interrupts pause the secure execution and pass control to their kernel code which performs cache measurements with high temporal resolution - resulting in overall high resolution for the attack.

A similar attack is theoretically possible on ARM CPUs, since it would not be susceptible to AutoLock restrictions if it runs on the same core as the secure code. However, the attack as described requires running kernel code, which is outside our attack model (Sect. 2.3). As stated before, running kernel code is extremely difficult on modern devices since loading kernel modules is either disabled or requires OEM signatures. Therefore, we attempted to create an attack that tries to imitate CacheZoom without running kernel code.

We began by binding a single thread to the core which runs the Kinibi OS and let the thread run in a loop that measures time differences between iterations. As long as there is no work pending for the TrustZone, the Kinibi OS does not receive many execution time slices, and so our thread measures small time differences between iterations (under a microsecond). However, when requests are made to the secure OS, we notice considerably higher measurements. Usually these measurements are single gaps of hundreds of microseconds to several

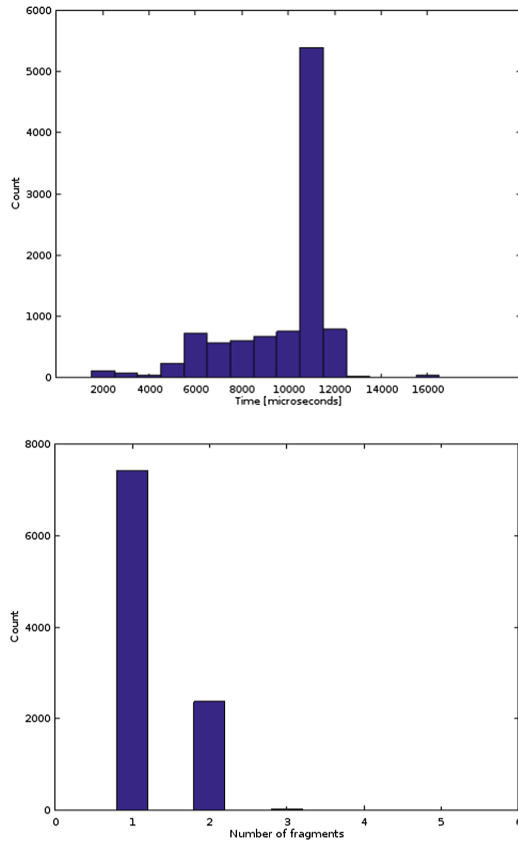

Fig. 3. Kinibi interrupted - measurements from the Normal World. Top: histogram of time difference between successive loop iterations where the difference exceeds 50 µs. Bottom: histogram of number of fragments per TrustZone call.

milliseconds—see Fig. 3 (top). This means that our thread is interrupted and the Secure World is scheduled.

Interestingly, on some occasions we observed more than one “gap fragment” per request; we believe this means that while the Secure World was running, a Normal World interrupt switched to the Linux kernel for handling that interrupt. After handling that interrupt, regular Linux scheduling took place, which first gave our iterating thread a time slice. Some time later, our thread was preempted by the kernel and execution was passed to the kernel thread which is responsible for translating Yield or SIQ requests from the *mcDriverDaemon* (which are periodically queued) to SMC calls. This kernel thread runs on the same core that the secure OS runs on (the secure OS rejects running interrupt handlers on other cores) and therefore our looping thread only resumes after the Secure World work was done or another interrupt is triggered on our core.

In Fig. 3 (bottom) we present the results mentioned above. The figure shows a histogram of the number of "gap fragments" we measured during a single call to the Keymaster. Most of the calls resulted in a single fragment, which means the Secure World was not interrupted; however, about 25% of calls resulted in two or more fragments, which implies that the Secure World was indeed interrupted. We grouped the measurements by those fragments and calculated their **sum**, as shown in the top graph. We see a clear peak around 10 ms—the total time it takes for the TrustZone to complete a request—even if the execution was interrupted and fragmented into two sessions or more. Crucially, we see that our looping thread gets control *while* the Keymaster work is paused, *on the same core*.

This evidence leads us to believe that this phenomenon can be leveraged to mount an attack on TrustZone. Our proposed attack consists of 4 Normal World user-land (EL0) threads:

1. A thread which makes Keymaster requests in a loop from one of the cores that the Kinibi OS is not bound to.
2. A looping thread running on the same core as the Secure World, which *primes* the cache sets and measures time differences between iterations. When a significant time difference is measured, it *probes* the cache sets and saves this measurement.
3. A thread running the *Flush+Reload* attack on the TCI memory, as described in Sect. 5.3 to trace the execution of the Keymaster trustlet as it handles the requests of thread #1. This allows us to select relevant measurements made by thread #2 by discarding *Prime+Probe* measurements made before the input buffer was accessed or after the output buffer (or the second TCI memory) was accessed. Thread #3 must run on a different core than thread #2.
4. A thread responsible for creating as many Normal World interrupts as possible, to increase the likelihood of interrupting the secure execution. Possible methods of doing this include creating network requests, in hope that the network card interrupts will be handled on our target core, or playing a video sequence causing graphic or sound card interrupts.

6 Conclusions

In this paper we provided, for the first time, a critical review of Samsung's proprietary TrustZone architecture. We described the major components and their interconnections, focusing on their security aspects. We discovered that the binary code for the Kinibi operating system runs in ARM32/Thumb mode even though the platform has a 64bit processor, and common OS defenses such as Address Space Randomization (ASLR), non-executable (NX) stack, or stack canaries are lacking. During this review we identified some design weaknesses, including one actual vulnerability.

We also found that the ARM32 assembly-language AES implementation used by the Keymaster trustlet is vulnerable to cache side-channel attacks. In

a separate paper we demonstrated successful cache attacks on a real device, against AES-256, on the Keymaster implementation, and presented a technique for mounting side-channel attacks against AES-256 in GCM mode.

Finally, we demonstrated realistic cache attack artifacts on the Keymaster cryptographic functions, despite the recently discovered "AutoLock" ARM feature. We successfully demonstrated cache side-channel effects on "World Shared Memory" buffers, and showed compelling evidence that full-blown cache attacks against the AES implementation inside the Keymaster trustlet are plausible.

We conclude that despite the architectural protections offered by the TrustZone, cache side-channel effects are a serious threat to the current AES implementation. However, side-channel-resistant implementations, that do not use memory accesses for round calculations, do exist for the ARM platform, such as a bit-sliced implementation [23] or one using ARMv8 cryptographic extensions [22]. Using such an implementation would render most cache attacks, including ours, ineffective.

A End-to-End Keymaster Communication Example

In the following section we describe an example of end-to-end communication between the normal and Secure World, that demonstrates how the entities mentioned above are chained together. In this section, numbers in parenthesis refer to their respective markers in Fig. 1:

1. In the Normal World user-space (NWd EL0), an application issues an encryption request to *keystored* through the *binder* interface (1). The kernel *binder* subsystem relays this request to *keystored*, which receives the request, loads the requested key file (and decrypts it with the relevant *masterkey*, recall Sect. 4.3) and calls the relevant function in the Keymaster HAL interface. Samsung's Keymaster HAL module writes a Keymaster trustlet request to TCI memory (2) and requests a trustlet notification from the *mcDriverDaemon* through the unix domain socket subsystem (3). The *mcDriverDaemon* calls the SIQ ioctl on the *mobicore* device (4).
2. In the Normal World kernel (NWd EL1), the Mobicore Kernel Module handles the ioctl by issuing a SIQ SMC (5).
3. Monitor code (EL3) is triggered to handle the SMC, it is deferred to the Mobicore SMC handler which issues an interrupt to the Kinibi OS and passes execution to it (6).
4. In the Secure World kernel (SWd EL1), the Kinibi OS interrupt handler schedules the *init*-like process and informs it of the interrupt (7).
5. In the Secure World userspace (SWd EL0), the *init*-like process handles the interrupt by sending an IPC message to the Keymaster trustlet (8). The Keymaster trustlet receives the IPC message, reads the TCI memory (9), parses and executes the request (e.g., encryption of data) (10). It then writes the output of the request to the TCI memory (11) and issues an IPC request to the *init*-like process to notify the Normal World (12). The *init*-like process then calls the SIQ SVC system call (13).

6. The Kinibi OS (SWd EL1) handles the SVC call by issuing a Normal World interrupt SMC call (14).
7. Monitor code (EL3) is triggered to handle the SMC, it is deferred to the Mobicore SMC handler which issues an interrupt to the Android Linux kernel and passes execution to it (15).
8. The Android Linux kernel (NWd EL1) interrupt handler is triggered, it calls the interrupt handler that the Mobicore kernel module registered. The Mobicore handler wakes up the *mcDriverDaemon* due to the interrupt (16).
9. Back in the Normal World userspace (NWd EL0), the *mcDriverDaemon* notifies its client of the interrupt through the unix domain socket subsystem (17). The Samsung's Keymaster HAL module receives the interrupt notification, reads and parses the response from TCI memory (18) and resumes the *keystored* function. *keystored* sends a response to the requesting application through the *binder* (19). Finally, the application execution resumes with the result.

References

1. ARM. ARM trusted firmware - firmware design documentation. https://github.com/ARM-software/arm-trusted-firmware/blob/v1.4/docs/firmware-design.rst#aarch64-bl31
2. ARM. ARM trusted firmware - runtime SVC code. https://github.com/ARM-software/arm-trusted-firmware/blob/v1.4/include/common/runtime_svc.h#L60
3. ARM. Building a secure System using TrustZone Technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf
4. ARM. ARM trustzone (2018). https://www.arm.com/products/security-on-arm/trustzone
5. Bernstein, D.J.: Cache-timing attacks on AES (2005). https://cr.yp.to/antiforgery/cachetiming-20050414.pdf
6. freddierice. Trident - temporary root for the Galaxy S7 active. https://github.com/freddierice/trident
7. Beniamini, G.: Trust issues: exploiting TrustZone TEEs (2017). https://googleprojectzero.blogspot.co.il/2017/07/trust-issues-exploiting-trustzone-tees.html
8. Ge0n0sis. How to lock the Samsung download mode using an undocumented feature of aboot (2016). https://ge0n0sis.github.io/posts/2016/05/how-to-lock-the-samsung-download-mode-using-an-undocumented-feature-of-aboot/
9. Giesecke & Devrient. Android kernel tree - mobicore kernel module. https://android.googlesource.com/kernel/msm/+/android-msm-shamu-3.10-marshmallow-mr2/drivers/gud/MobiCoreDriver/
10. Google. Android keymaster HAL. https://source.android.com/security/keystore/implementer-ref
11. Google. Android keystore. https://developer.android.com/training/articles/keystore.html
12. Google. Android keystore - source code. http://androidxref.com/6.0.0_r1/xref/system/security/keystore/keystore.cpp

13. Google. Android vold cryptfs. http://androidxref.com/6.0.0_r1/xref/system/vold/cryptfs.c
14. Green, M., Rodrigues-Lima, L., Zankl, A., Irazoqui, G., Heyszl, J., Eisenbarth, T: Autolock: why cache attacks on ARM are harder than you think. In: 26th USENIX Security Symposium (2017)
15. Lapid, B., Wool, A.: Cache-attacks on the ARM TrustZone implementations of AES-256 and AES-256-GCM via GPU-based analysis. Cryptology ePrint Archive, Report 2018/621 (2018). http://eprint.iacr.org/2018/621
16. Lipp, M., Gruss, D., Spreitzer, R., Maurice, C., Mangard, S.: ARMageddon: cache attacks on mobile devices. In: USENIX Security Conference (2016). https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_lipp.pdf
17. Moghimi, A., Irazoqui, G., Eisenbarth, T.: CacheZoom: how SGX amplifies the power of cache attacks. In: Fischer, W., Homma, N. (eds.) CHES 2017. LNCS, vol. 10529, pp. 69–90. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66787-4_4
18. nccgroup. Cachegrab. https://github.com/nccgroup/cachegrab
19. Neve, M., Seifert, J.-P.: Advances on access-driven cache attacks on AES. In: Biham, E., Youssef, A.M. (eds.) SAC 2006. LNCS, vol. 4356, pp. 147–162. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74462-7_11
20. Neve, M., Tiri, K.: On the complexity of side-channel attacks on AES-256 - methodology and quantitative results on cache attacks. Technical report (2007). https://eprint.iacr.org/2007/318
21. Artenstein, N., Goldman, G.: Exploiting android s-boot: getting arbitrary code exec in the Samsung bootloader (2017). http://hexdetective.blogspot.co.il/2017/02/exploiting-android-s-boot-getting.html
22. OpenSSL. ARM AES implementation using cryptographic extensions. https://github.com/openssl/openssl/blob/master/crypto/aes/asm/aesv8-armx.pl
23. OpenSSL. ARMv7 AES bit sliced implementation. https://github.com/openssl/openssl/blob/master/crypto/aes/asm/bsaes-armv7.pl
24. OpenSSL. OpenSSL FIPS. https://www.openssl.org/docs/fips.html
25. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: the case of AES. In: Pointcheval, D. (ed.) CT-RSA 2006. LNCS, vol. 3860, pp. 1–20. Springer, Heidelberg (2006). https://doi.org/10.1007/11605805_1
26. Oliva, P.: ldpreloadhook. https://github.com/poliva/ldpreloadhook
27. Qualcomm. Snapdragon security (2018). https://www.qualcomm.com/solutions/mobile-computing/features/security
28. Samsung. Mobile processor: Exynos 7 Octa (7420) (2018). http://www.samsung.com/semiconductor/minisite/exynos/products/mobileprocessor/exynos-7-octa-7420/
29. Samsung. Platform security (2018). http://developer.samsung.com/tech-insights/knox/platform-security
30. Samsung. Android security updates, June 2018. https://security.samsungmobile.com/securityUpdate.smsb
31. Spreitzer, R., Plos, T.: Cache-access pattern attack on disaligned AES T-tables. In: Prouff, E. (ed.) COSADE 2013. LNCS, vol. 7864, pp. 200–214. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40026-1_13
32. Trustonic. Trustonic Kinibi technology. https://developer.trustonic.com/discover/technology
33. Trustonic. Trustonic mobicore driver daemon - client library. https://github.com/Trustonic/trustonic-tee-user-space/tree/master/MobiCoreDriverLib/ClientLib

34. Trustonic. Trustonic mobicore driver daemon - command header. https://github.com/Trustonic/trustonic-tee-user-space/blob/master/MobiCoreDriverLib/Daemon/public/MobiCoreDriverCmd.h
35. Trustonic. Trustonic mobicore driver daemon - FSD. https://github.com/Trustonic/trustonic-tee-user-space/tree/master/MobiCoreDriverLib/Daemon/FSD
36. Trustonic. Trustonic mobicore driver daemon - source code. https://github.com/Trustonic/trustonic-tee-user-space/tree/master/MobiCoreDriverLib/Daemon
37. Xinjie, Z., Tao, W., Dong, M., Yuanyuan, Z., Zhaoyang, L.: Robust first two rounds access driven cache timing attack on AES. In: 2008 International Conference on Computer Science and Software Engineering, vol. 3, pp. 785–788. IEEE (2008)
38. Zhang, N., Sun, K., Shands, D., Lou, W., Thomas Hou, Y.: TruSpy: cache side-channel information leakage from the secure world on ARM devices. IACR Cryptology ePrint Archive, 2016(980) (2016)