Unearthing the TrustedCore: A Critical Review on Huawei's Trusted Execution Environment

Marcel Busch, Johannes Westphal, and Tilo Müller
Friedrich-Alexander-University Erlangen-Nürnberg, Germany
{marcel.busch, johannes.westphal, tilo.mueller}@fau.de

Abstract

Trusted Execution Environments (TEEs) are an essential building block in the security architecture of modern mobile devices. In this paper, we review a TEE implementation, called TrustedCore (TC), that has been used on Huawei phones for several years. We unveil multiple severe design and implementation flaws in the software stack of this TEE, which affect devices including the popular Huawei P9 Lite, released in 2016, and partially the more recent Huawei P20 Lite, released in 2018. First, we reverse-engineer TC's components, their interconnections, and their integration with the Android system, focusing on security aspects. Second, we examine the Trusted Application (TA) loader of the TC platform and reveal multiple design flaws. These flaws allow us to decrypt any TA found on our target devices and, thus, break code confidentiality. Third, we describe the design of Huawei's keystore system, the heart of all services using hardware-backed cryptography. We found severe vulnerabilities in this keystore system and demonstrate the leakage of export-protected keys from the TEE, which considerably weakens full-disk encryption. Fourth, along with these findings, we additionally discovered an exploitable memory corruption within Huawei's keymaster TA, enabling us to execute arbitrary code within the ARM TrustZone at the highest privilege level. The exploit requires us to bypass several mitigation techniques such as stack canaries and Address Space Layout Randomization (ASLR), which are all flawed in this TEE's design. We reported our findings to Huawei in a responsible disclosure procedure and publicly discuss our analyses for the first time in this paper.

1 Introduction

TEEs are an integral part of the security architecture of mobile devices. They provide an execution context where security-critical services, such as user authentication, mobile payment, and digital rights management, can run isolated from the Rich Operating System (Rich OS). The Rich OS, *e.g.,* Android or iOS, typically has a complex software stack and is thus prone to error. In theory, any bug in the feature-rich domain, including severe kernel-level bugs, cannot affect the integrity and confidentiality of a TEE, as TEEs are isolated from the rest of the system by means of hardware primitives. As ARM is the predominant architecture used for chipsets in mobile devices, the ARM TrustZone (TZ) [6] provides the trust anchor for virtually all TEEs in mobile devices, including Huawei models.

Although TEEs have been extensively used by millions of products for years, security analyses targeting these systems are rarely discussed in public since all major vendors, including Huawei, Qualcomm, and Samsung, maintain strict secrecy about their individual proprietary implementations. However, the correct implementation of a TEE is complex and comparable with the design of an operating system, posing innumerable challenges for vendors. These challenges regularly lead to severe bugs that potentially undermine the whole mobile device's security architecture [13–16, 31, 32].

In this work, we unveil several design and implementation flaws in a TEE implementation by Huawei, called TrustedCore (TC), that has been commercially used on millions of devices. We found all of these issues independently and are not aware of any other research covering them. The reviewed version of TC was deployed on the popular Huawei P9 Lite, released in 2016, and our discoveries partially apply to the more recent Huawei P20 Lite, released in 2018. Note that on newer Huawei devices (*e.g.,* Huawei P40 and Huawei P30), the TEE architecture and implementation has changed, and a TEE called *iTrustee* replaced TC [42]. Our research focuses on certain versions of TC and, in detail, we make the following contributions:

First, this review is the first holistic architectural description of TC, to the best of our knowledge. Using a combination of static and dynamic analysis, we reverse-engineer TC's components and systematically describe their interconnections as well as their integration with the Android system.

Second, we reveal, as well as exploit, multiple design flaws in the decryption and verification routines used to load confidential TAs, enabling us to recover all plaintext binaries from our device. We conduct this analysis by examining the TA loader used by TC to transfer confidential TAs into the TEE during runtime. As a consequence and even more compromising, we detected the usage of the same decryption key on several devices of the Huawei P-series, including the Huawei P20 Lite, Huawei P10, and Huawei P9 Lite. Although other TZ-based TEEs are capable of loading TAs during runtime, the confidentiality protection is unique to Huawei's implementation and studied for the first time in this research.

Third, we analyze Huawei's version of the Android keystore system, which is the heart of all services using hardware-backed cryptography [24]. We describe the design of Huawei's Android keystore system, unveil fatal design flaws, and demonstrate the leakage of export-protected cryptographic keys from the TEE. Complementary to other research reviewing the keystore system on Qualcomm's TEE [14] (used on Nexus and Pixel devices) and Samsung's TEE [31] (used on the Galaxy series), we present the first review of the keystore system as implemented by Huawei.

Fourth, we found a vulnerability that allows us to execute arbitrary code and escalate our privileges to the highest level of the TEE context. While writing the exploit code, we probed all mitigation techniques present on the Huawei TEE and found weaknesses in all mitigations, for instance, stack canaries and ASLR. More specifically, we gain arbitrary code execution within the keymaster TA, which is the TEE component of the keystore system. From this TA context, we identify the TEE kernel's attack surface and argue that the Application Programming Interface (API) available to the keymaster TA is unnecessarily powerful.

Responsible Disclosure We found all of the issues covered in this paper in August 2019 and, in coordination with Huawei, tried to assess their scope and impact by investigating newer phones and firmware images. Huawei's restrictive bootloader unlock policy from May 2018 [20], and the introduction of encrypted firmware images, are effective mitigations against our triaging efforts as well as static and dynamic analyses of TEE components in general. In April 2020, we filed comprehensive reports covering the scope and impact of our findings. Huawei acknowledged all bugs and indicated that all issues are already known and fixed with updates distributed in November 2019 and January 2020.

2 Background

This section provides the reader with the necessary preliminaries to get the most out of our analysis of Huawei's TC. First, we introduce the privilege levels of modern ARM-based devices. Second, we briefly mention the TEE implementations found on commercially available Android devices and give examples of their usage. Third, we elaborate on the adversary model assumed for the attacks presented in our research.

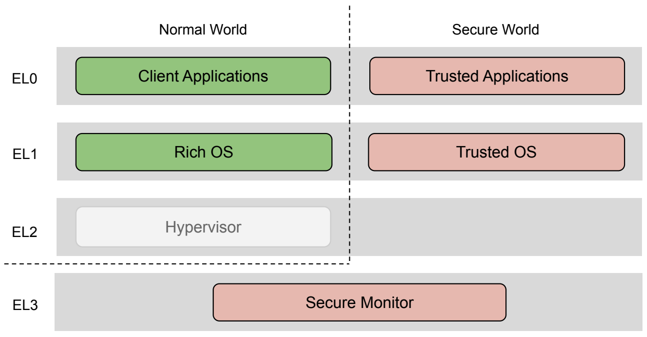

Figure 1: The ARMv8-A architecture supports up to four privilege levels, called exception levels (ELs). ARM TrustZone splits these ELs into two worlds, the Normal World and the Secure World. Commonly, a prefix indicates the world of the EL (e.g., S-EL0), if not clear from the context.

2.1 ARMv8-A Privilege Levels

The ARMv8-A architecture supports up to four privilege levels [7], called Exception Levels (ELs), as illustrated in Figure 1. In addition to the typical dual-mode split into userland (EL0) and kernel (EL1), this architecture can host a hypervisor in EL2. Since hypervisors tend to be more relevant for cloud-computing scenarios than mobile devices, they are not in the scope of our work. A further addition to the traditional model are the ARM Security Extensions, also known as ARM TrustZone (TZ) [8]. TZ allows system designers to partition the device's hardware and software resources into two states, a non-secure state, and a secure state.

The partitioning of the software resources on mobile devices, which are predominantly based on ARM System-on-Chips (SoCs), usually results in a split as illustrated in Figure 1. In the non-secure state, also referred to as Normal World (NW), the userland application stack (*e.g.,* Android) and the kernel (*e.g.,* Linux) run in N-EL0 and N-EL1, respectively. When in the secure state, also referred to as Secure World (SW), Trusted Applications (TAs) run in the userland (S-EL0) and are hosted by a Trusted Operating System (Trusted OS) executed in S-EL1. The two worlds have their own page tables and, if properly configured, the NW cannot access physical memory regions assigned to the SW, whereas the SW has unrestricted access. Switching between these worlds is possible through a component called the Secure Monitor, which is executed on the highest privilege level (S-EL3). On all commercially available platforms, a secure boot chain only permits the execution of vendor-signed software components within the SW. Thus, all SW components are part of the system's Trusted Computing Base (TCB). In contrast, the NW is more open and allows for the installation of third-party apps. Many vendors allow developers to modify the Rich OS kernel (N-EL1) after unlocking the bootloader of their device. Unfortunately, Huawei does not offer this option for their devices anymore since May 2018 [20].

The partitioning of the hardware resources of the SoC allows system designers to grant the SW exclusive access to peripherals. A commonly known use case is the fingerprint sensor on phones. This sensor is usually exclusively accessed by the SW software to keep the captured fingerprint image secret. For this use case, the SW provides an API for the NW to enroll and verify fingerprints, without the NW ever accessing the sensor or the captured data.

2.2 TEEs on Android

Different vendors implement different TEEs, leading to even more fragmentation of the systems that we summarize under the umbrella term “Android”. Depending on the SoC used in a mobile device, we can at least identify four commercially and widely used TEE implementations. Qualcomm chipsets, present in Google’s flagship Pixel series and many more devices, run the Qualcomm Secure Execution Environment (QSEE) [34]. Samsung devices, if shipped with an Exynos chipset, either run a TEE named Kinibi [35], developed by Trustonic, or Samsung’s own TEE named TEEGris [36]. On Huawei devices, we have chipsets by HiSilicon, and can find a TEE called TrustedCore (TC) [41]. Huawei’s TC is the primary focus of our work.

The features brought to Android systems by TEEs enhance many services, including user authentication and full-disk encryption. For instance, all Android systems implement user authentication by three TEE-backed NW components [25]: the Gatekeeper daemon, responsible for Personal Indentification Number (PIN), pattern, and password authentication; the Fingerprint daemon, responsible for fingerprint-based authentication; and the Keystore daemon, providing cryptographic operations for the two previous daemons. All three of these daemons usually have a counterpart in S-EL0 (*e.g.,* the gatekeeper, fingerprint, and keymaster TAs), hosting the security-sensitive parts of the authentication operation.

TEE-backed full-disk encryption is carried out by the volume daemon (`vold`) and its interplay with the keymaster TA in the TEE. A cryptographic key, only available in plaintext in the TEE, guarantees that an encrypted partition can only be decrypted on the device. This hardware-binding property allows vendors to thwart brute-force attacks against the user’s PIN, authentication pattern, or password by introducing delays if too many login attempts occur, or, *e.g.,* in corporate environments, wipe the partition after a certain number of failed login attempts.

2.3 Attacker Model

The attacker model underlying our work assumes all NW components to be untrusted. This model corresponds to the attacker commonly assumed for TZ-based TEE systems and is most realistic for Android-based mobile devices. In practice, all of the demonstrated attacks can be carried out by having full control over N-EL0, corresponding to a root-level attacker.

Note that the privileges of a sandboxed Android app are generally insufficient to execute our attacks, but TEEs were designed to protect against strong attackers such as root and kernel-level attackers. Having root or kernel privileges in Android is a reasonable assumption as many documented flaws led to this situation in the past [11, 40].

However, we do not need to modify and execute code on N-EL1. For the privilege escalations, first, to S-EL0, and, second, to S-EL1, discussed in Section 6.1 we do not even need root-level privileges in N-EL0 (depending on the firmware version) since the privileges of a system service having access to a specific kernel module are sufficient.

3 Reversing the TrustedCore Architecture

We base our study of TC’s architecture on the dynamic analysis of various components of two Huawei phones, the Huawei P9 Lite and the Huawei P10 Plus that we had root-level access to. Furthermore, we collected multiple firmware images for both devices, analyzed relevant components statically, and, thus, were able to identify changes during the evolution of the architecture. Lastly, we had access to the Linux kernel source code for both of our devices [33].

During the following discussion of the different components and their interconnections, Figure 2 serves as an overview. This figure is organized using the world-split and the ARMv8 exception levels, as discussed above in Section 2.1. In the following sections, we first cover the NW components and then the SW components.

3.1 Normal World

The central components making use of TEE features are Android system services. These system services expose their functions to apps, or other system services, using the binder Inter-Process Communication (IPC) framework [38]. This interface is specified and forms the backend of the Android Framework API commonly known to all Android app developers. Since different SoCs usually ship with different TEEs, system services that make use of the TEE must deal with different implementations. For this purpose, Google decided to define a common interface, the Hardware Abstraction Layer (HAL) [23], to access vendor-specific implementations to interact with heterogeneous hardware. This HAL library, individual to each system service, is implemented by the vendor (in this case, Huawei) and implements the system service’s HAL interface.

On Huawei devices, the HAL library is responsible for two tasks. First, it maps the data structures passed to the common HAL interface to the data structures expected by

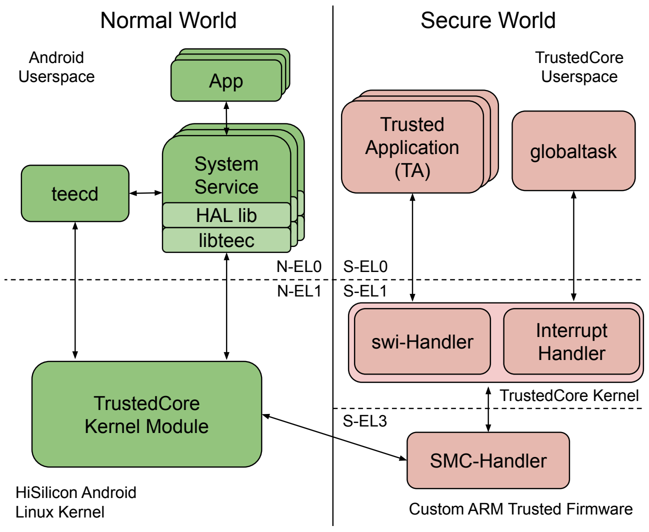

Figure 2: Taking the ARMv8-A privilege levels as a scaffold, this figure illustrates TrustedCore's components, their interconnections, and their integration with the Android system.

the TEE implementation, *i.e.,* the data structures expected by the addressed TA, and second, it accounts for the TEE's state machine. In order to fulfill these two tasks, the HAL library uses Huawei's TEE interface, encapsulated within the `libteec` library. `libteec`'s interface primarily implements the GlobalPlatform (GP) TEE client API [21], which we use next to elaborate on common interaction patterns. The HAL library as well as the `libteec` are proprietary.

A common sequence of functions using the GP TEE Client API, as used on Huawei, devices is the following:

1. `TEEC_InitializeContext`: During the context initialization call, the client connects to a Unix domain socket of `teecd`. `teecd` is the only process that is allowed to open a file descriptor to `/dev/tc_ns_client`, which is the device node exposing the Linux kernel module to interact with TC. Its executable file is a statically compiled and stripped ELF64 binary, which might be a protection measure against reverse engineering. As another protection measure, `teecd` only forwards an opened file descriptor to the kernel module to certain clients. For Android apps, the `apk` certificates are verified. For system services (*e.g.,* `keystored`), the package names and the user ids are checked. `teecd` uses a whitelist of allowed clients for these checks. Furthermore, `teecd` passes the login credentials to the TEE driver, which associates the credentials with the file descriptor.

2. `TEEC_OpenSession`: With an initialized context, a client can open a session to a TA. First, it is checked if the TA is already loaded. If it is not loaded, the client tries to load it. In the SW, the handlers for loading TAs are implemented in the `globaltask` TA, which acts like an `init` process in this regard. `globaltask` distinguishes between built-in TAs and *secure* TAs in the loading process. Built-in TAs are loaded during TC's initialization. Secure TAs can be loaded on demand and their corresponding encrypted binaries, identified by a `*.sec` extension, reside in the Android file system. The process of loading encrypted TAs is covered in more detail in Section 4.

   When the target TA is loaded within TC, the client issues the actual open-session request. In this process, its credentials are once again checked within the kernel. In our analysis, we observed different evolutions of this authentication inside the driver. The most advanced technique creates a SHA-256 hash of all executable pages of the requesting client and compares it against a hash stored in a signed file (*e.g.,* `/vendor/etc/native_packages.xml`). This file is also sent to the TEE during the initialization of the system to allow for verification within the trusted context. The request is forwarded to the target TA and processed in its corresponding open-session handler. If the TA accepts the open-session request, a session id is assigned and the client can issue commands.

3. `TEEC_InvokeCommand`: Next, the client can finally invoke commands of the TA. The code for context and session establishment is self-contained and implemented in the HAL library. In the command-invocation stage, the data passed to the HAL is used. To exchange data with the SW, the TC kernel module implements a shared memory infrastructure that is based on a physical memory region dedicated to sharing memory between the two worlds. Note that this dedicated memory region is necessary because each world has its own page tables and does not know about the page tables of the other world. Using this shared memory infrastructure, a client (*i.e.,* a system service) can pass data over to the SW. These data structures consist of information helping the SW to dispatch the request to the proper TA, and the actual payload for the TA.

4. `TEEC_CloseSession` and `TEEC_FinalizeContext`: The remaining two functions are used to close the session and free all resources connected with the context.

3.2 Secure World

While the NW components can be studied using static and dynamic analysis, we are restricted to static analysis and the input/output behavior observable from the NW when investigating SW components. Many of the TEE-related software components on Huawei devices can be found on the `teeos` partition. This partition contains the following components:

- The Trusted OS kernel, `TrustedCore`.
- The "init process", `globaltask`.
- Multiple TAs including the keymaster TA.

While the TAs are 32-bit ARM ELF binaries, the `TrustedCore` and the `globaltask` binaries do not contain headers reveiling their internal structure. However, we found the remainders of a string table and a symbol table in both binaries. Using entries from the symbol table (*i.e.,* `CODE_START`, `DATA_START`, and `BSS_START`), we were able to identify the original sections (*i.e.,* `.text`, `.data`, and `.bss`) of the binaries. As a result, we implemented a tool based on Python's elftools library to fill the missing section header string table, *e.g.,* `.shstrtab`, and create proper ELF headers from the gathered information. This step allowed us to study the inner workings of `TrustedCore` and `globaltask`.

For completeness, before we dive into TC, the `trustedfirmware` partition contains a modifed version of ARM's reference implementation of the secure monitor (ARM TrustedFirmware [9]). We did not investigate this part of the SW software stack further, except for noticing that it uses the 64-bit instruction set of the ARMv8-A architecture (AArch64) and identifying the `smc`-handlers responsible for the context switch, either from NW to SW or vice-versa.

The TC kernel acts as a dispatcher for TA requests from the NW and exposes its system call interface to all its TAs. Additionally, it handles interrupts originating from peripherals like the fingerprint sensor. To connect the GP TEE Client API call sequence observed in the NW with their corresponding components in the SW, we systematically go through the interaction from a TA's perspective.

By reviewing `globaltask` and multiple TAs, we noticed that all TAs implement the GP Internal Core API [22]. Before a client can interact with a TA, it has to initialize a context with the TEE by contacting `globaltask`. Internally, this context initialization ensures that the TA is properly loaded and ready to process requests. Thus, `globaltask` can be seen as the `init` process of TC. A common interaction sequence with a TA includes the following functions:

1. `TA_CreateEntryPoint`: This function is the constructor for TAs. It is called only once during the lifetime of the TA when the first session is established (*i.e.,* by a NW client calling `TEEC_OpenSession`). The gatekeeper TA gives an example of the state initialized by this constructor. This TA needs the total number of failed login attempts across client sessions to introduce a delay after a certain number of failed attempts.
2. `TA_OpenSessionEntryPoint`: This function is the direct counterpart of a NW client calling `TEEC_OpenSession`. In many TAs we found authentication logic (*i.e.,* checking user ids or signatures) and the initialization of session-specific state (*i.e.,* data structures to save state across multiple command invocations).
3. `TA_InvokeCommandEntryPoint`: This function invokes the actual command handler of the TA. Its direct counterpart from a NW client's perspective is the `TEEC_InvokeCommand` function. For instance, the keystore daemon invokes the keymaster TA to generate export-protected cryptographic keys. These keys can be used by further command invocations to perform cryptographic operations with them (*i.e.,* encrypt, decrypt, verify, or sign data). The keymaster TA will be later discussed in detail (see Section 5).
4. `TA_CloseSessionEntryPoint`: This function frees all session-specific state. Its NW counterpart is the `TEEC_CloseSession` function.
5. `TA_DestroyEntryPoint`: This function deallocates all resources reserved in the initial entry point creation when a TA is unloaded from TC.

This sequence of functions summarizes the interaction pattern from a TA's perspective. There is one more element to be discussed, which are TEE agents. TEE agents are implemented as threads in `teecd` and enable TAs to leverage features of the Linux kernel. For example, TC does not implement file system drivers itself. Instead, it calls the corresponding agent to carry out storage operations. This agent runs in a common Linux userland process (*e.g.,* `teecd`) and, therefore, can leverage the file system implementation of the kernel. We identified three agents: the `fs` agent, for file system operations, the `socket` agent, for networking operations, and the `misc` agent, for time retrieval operations.

4 Breaking Code Confidentiality of TAs

In this section, we first discuss TC's loader capable of decrypting confidential TAs. Then, we elaborate on the identified design flaws and their consequences. The analysis in this section has been conducted on a Huawei P9 Lite device.

4.1 Loading Encrypted TAs

During system startup, we noticed that the `fingerprintd` interacts with the TEE driver to load TAs from the `/system/bin` directory. The files loaded have a `*.sec` extension and their contents are encrypted. In total, we can find eleven `*.sec` files in this directory, all being encrypted TAs. Since we have access to the loader due to the previous analyses, we can study the loading process for the encrypted TAs.

We found the loader implementation within the `globaltask` TA. The `*.sec` files have restrictive Linux permissions and can only be loaded by certain system

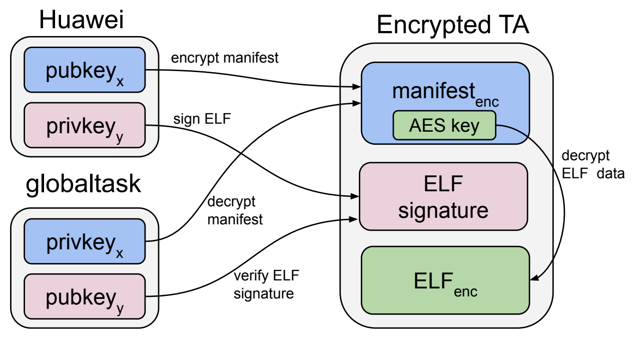

Figure 3: An encrypted Huawei TA consists of an encrypted manifest, a signature for the unencrypted ELF file, and the encrypted ELF file. The encrypted manifest contains an AES key to decrypt the ELF file.

services. During the loading process, a buffer containing the encrypted TA from the file system is passed to `globaltask`.

Figure 3 gives an overview of the encrypted TA and the involved cryptographic keys. The loading process takes place in four stages:

First, a region of the passed buffer, called *manifest*, is decrypted using an *RSA private key*. Second, another region containing a signature of a SHA-256 hash is verified using an *RSA public key*. Third, an *AES key*, which is part of the previously decrypted manifest, is used to decrypt a further region of the passed buffer, resulting in the ELF file of a TA. Fourth, the SHA-256 hash from the second stage is compared against the SHA-256 hash of the decrypted ELF file.

As can be seen in Figure 3, the RSA private key and the RSA public key used within `globaltask` are part of different key pairs. Huawei has the corresponding parts of the keys present in `globaltask`.

After `globaltask` successfully decrypted a TA, it requests the creation of a process from the TC kernel and updates its management structures for running TAs. As soon as the initial `ioctl` system call returns, requests can be sent to the newly spawned TA.

4.2 Extracting TA Decryption Keys

The essential question about the loading mechanism as described in the previous section is: *how is the RSA private key protected?* Ideally, it would never be loaded to Random Access Memory (RAM), and all operations making use of it would take place in a Trusted Platform Module (TPM) or a crypto unit. Thus, even if `globaltask` were to be compromised, the key would stay confidential. However, the design of TC's loading mechanism is based on white-box cryptography to protect its keys. White-box cryptography is essentially obfuscation of cryptographic code and keys. Instead of hardcoding the key, it is hidden within the structures of the

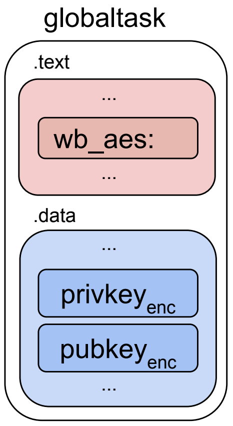

Figure 4: The white-box AES implementation is part of `globaltask`'s `.text` section while the ciphertext for both keys resides in its `.data` section.

cryptographic algorithm used, making it hard to extract.

This design's problem is that the key can be extracted with some effort [17], or directly extracted by executing the white-box cryptography code. In the case of `globaltask`, a white-box AES implementation is used to decrypt the RSA private key from a cyphertext present in `globaltask`'s `.data` section. Since TAs on TC contain 32-bit ARM instructions, we can slice the relevant parts from `globaltask` and piece an executable program together that performs the decryption operation for us.

To investigate the key re-usage across multiple firmware images, we built a tool based on Python and the QEMU emulator [10] that extracts the private and the public RSA keys from a given `globaltask` binary. Figure 4 illustrates the relevant parts of the `globaltask` binary. Using a Python script, we generate C sources, as illustrated in Listing 1. The `globaltask` binary and all variables indicated with `<...>` are placed into this program template by the script since they depend on the version at hand. In this program, we first map the content of the `globaltask` binary readable, writable, and executable (Line 10 in Listing 1). Second, we calculate the addresses to the encrypted keys and the white-box AES function (Lines 11-13 in Listing 1). Lastly, we call the function to decrypt the keys and print their plaintexts (Lines 15-19 in Listing 1). The resulting C program can be compiled using a 32-bit ARM toolchain and executed using the QEMU emulator in ARM user emulation mode. This mode allows us to execute an ELF32 ARM binary on our AMD64 machine.

Having the loader's RSA private key, we successfully decrypted the manifests of all TAs found on the file system of our target device, the Huawei P9 Lite. Within the decrypted manifests, we found that all encrypted TAs have different AES keys. Next, we decrypted all encrypted TA ELF files using their corresponding AES key and manually verified the resulting ELF files. By completing this step, we have defeated

```c
char globaltask[] = { ... }; // globaltask binary

int main(){
    char *pubkey_dec[0x1000] = { 0 };
    char *privkey_dec[0x1000] = { 0 };
    char* (*wb_aes) (char*, char*, unsigned int);

    mprotect(globaltask, sizeof(globaltask),
        PROT_READ|PROT_WRITE|PROT_EXEC);

    pubkey_enc = globaltask + <pubkeyenc_off>;
    privkey_enc = globaltask + <privkeyenc_off>;
    wb_aes  = globaltask + <wb_aes_off>;

    wb_aes(pubkey_enc, pubkey_dec, <pubkey_sz>);
    hexdump("privkey:", pubkey_dec, <pubkey_sz>);

    wb_aes(privkey_enc, privkey_dec, <privkey_sz>);
    hexdump("privkey:", pubkey_dec, <privkey_sz>);

    return 0;
}
```

Listing 1: Given the `globaltask` binary, offsets to the white-box AES implementation, and the ciphertexts, this C program template decrypts the keys for us.

the code confidentiality of TAs as implemented on our target device.

To better understand the impact of this finding, we investigated our dataset of 133 firmware images distributed to different Huawei device models from July 2015 until April 2018. We were able to directly extract the private keys without modifying our tooling, from 22 of these firmware versions. Shockingly, we discovered that the private key was the same across all firmware images. In a further analysis, we found artifacts (*i.e.,* log strings) indicating the presence of the same white-box cryptography design in 119 of our 133 images, the last one distributed in April 2018. We found evidence for this flawed design being used on recent device models, like the P10 Plus and P20 Lite, by successfully decrypting TAs obtained from these devices' firmwares.

5 Extracting Export-Protected Keys

In this section, we introduce the Android keystore system and elaborate on the design flaws found in Huawei's implementation. Afterward, we explain the impact of our finding using the example of Full-Disk Encryption (FDE) and provide evidence that this design flaw has been existing for several years. The dynamic analysis underlying this examination was conducted on the Huawei P9 Lite.

5.1 The Android Keystore System

The Android keystore system is capable of binding key material to the hardware [24]. This hardware-binding feature is leveraged through the TEE. In particular, an app can request the generation of cryptographic keys and operations performed using these keys from the keystore system service

```c
struct keymaster1_device {

  struct hw_device_t common;

  /* ... */

  keymaster_error_t (*generate_key)(
    const struct keymaster1_device* dev,
    const keymaster_key_param_set_t* params,
    keymaster_key_blob_t* key_blob,
    keymaster_key_characteristics_t** characteristics);

  /* ... */

};
typedef struct keymaster1_device keymaster1_device_t;
```

Listing 2: An indicative keymaster HAL function to generate keys. The `key_blob` output parameter contains the encrypted key material on the successful return of the function.

(`keystored`) using the binder IPC framwework [38]. The requesting app can specify an alias to refer to cryptographic key material. In a key-generation request, the `keystored` converts this request to the vendor-specific format and contacts the TEE driver to forward the request to the keymaster TA. Next, the keymaster TA performs the key-generation request and returns an encrypted keyblob to the `keystored`. This keyblob is encrypted using a key we call *Key Encryption Key (KEK)*. Lastly, `keystored` acknowledges the app's request and, hence, the app can use the specified alias to refer to this keyblob in succeeding cryptographic operations. Figure 5 in Section A provides a further illustration of the Android keystore system.

The goal of this mechanism is to bind the key material to the secure hardware and never expose its plaintext to the NW. The app can use the alias to the keyblob maintained by `keystored` to perform cryptographic operations on data. `keystored` represents an additional layer of protection and exposes a unified interface to the hardware-backed keymaster, which, in our case, is a TA running in TC, but could also be implemented using a secure element. This design ensures that the keys will never be in NW memory in plaintext form, and also that even the key's encrypted form (*i.e.,* the keyblob) will never be in the app's memory. Only the TEE can access the plaintext keys.

While the Android Open Source Project (AOSP) provides the keystore's integration with the Android Framework, the NW infrastructure to communicate with the TEE is vendor-specific, as explained in Section 3.1. For instance, the part of the keystore implementation that converts the key-generation request into a format that the keymaster TA understands is implemented through Huawei's HAL and `libteec` libraries. The keystore HAL defines functions to carry out cryptographic operations. To investigate the keyblob from above, we take a closer look at the HAL interface function responsible for generating keys, which is illustrated in Listing 2.

The `dev` parameter is an input parameter and used as a reference to the `keymaster1_device_t` itself. `params` is an input parameter and points to an array containing key generation parameters. The `key_blob` is an output parameter and contains an opaque `uint8_t` pointer to the key material. This keyblob represents the handle from above. `characteristics` is also an output parameter and contains the characteristics of the generated key, *i.e.,* key generation parameters like algorithm, block mode, padding, or digest, and authorization modes of usage or access restrictions.

The returned keyblob is used within further cryptographic operations like encrypting, decrypting, signing, and verifying messages. Since this parameter is defined as an opaque pointer, its format is up to the vendor.

5.2 Extracting the Keymaster's Master Keys

In the design of the keymaster TA, we found that a Keyed-Hash Message Authentication Code (HMAC) is used to verify the integrity and authenticity of the presented keyblob. If the HMAC that is part of the presented keyblob does not match the HMAC calculated from the keyblob, the keymaster TA rejects the request. An HMAC requires a *secret* cryptographic key. In the case of the keymaster TA, this secret key must only be known by the TEE to guarantee that nobody else can generate an authentic keyblob.

The keyblob, as the name suggests, contains the encrypted key material. In our review, we found that this key material is encrypted with an `AES-CBC` scheme. Surprisingly, we unveiled that the AES key, *e.g.,* the KEK, used is a *constant*. Using the KEK, we can decrypt the key material outside of the TEE.

Regarding the HMAC, we discovered the usage of an `HMAC-SHA256` scheme also using a *constant* key, which is different from the KEK. Thus, both keys are constants residing in the virtual address space of the TA, and even worse, they are contained within the TA's binary and not dynamically loaded from the Trusted OS or special-purpose hardware, like it is the case on other platforms [31]. In Section A Listing 3 we provide an illustration of the keyblob structure.

Having both keys, we can decrypt export-protected keys outside of the TEE and create authentic keyblobs ourselves.

5.3 Breaking Full-Disk Encryption

Android's FDE is based on the security guarantees of the used TEE [26]. An essential goal of hardware-backed FDE is to bind the decryption of the secured content to the device and prevent offline attacks that would allow an adversary to brute-force a user's PIN, unlock pattern, or password. Binding the encryption key to the device allows for the introduction of additional defenses, *i.e.,* an enforced delay after a certain amount of failed login attempts, or even the erasure of data from the device if a maximum number of failed attempts is reached.

For an in-depth technical explanation of Android's TEE-backed FDE scheme, we recommend Beniamini's technical blog [14]. Beniamini did a similar analysis regarding the keymaster TA on Qualcomm's TEE, called QSEE. The significant difference to the keymaster TA design on the Nexus 6 device compared to the one employed on our Huawei P9 Lite is that the KEK is derived from a device-specific hardware key, called *SHK*, rendering the KEK device-specific as well. In contrast, we did not find such a device-specific, hardware-key-based derivation in Huawei's keymaster TA. The disastrous consequence is that, while on the Nexus 6, an adversary would need to retrieve the KEK for each device, the KEK for the Huawei P9 Lite is the same on all devices using the same keymaster TA. Therefore, an adversary can launch offline brute-force attacks on arbitrary devices in order to break the device's FDE.

In our analysis of Huawei's FDE, we found that it works similar to the description provided by Beniamini [14]. One of the noteworthy differences to his analysis on the Nexus 6, is the location of the *crypto footer* which contains the TEE-backed encrypted keyblob of an RSA private key used in combination with the user credentials and a salt to decrypt the Device Encryption Key (DEK). The DEK is eventually used to decrypt the `userdata` partition of the device. We found the crypto footer within the last 4K bytes on the very same `userdata` partition.

In addition to our dynamic analysis, we investigated four different devices by statically analyzing their firmware images. In total, we obtained 133 firmware images. We were able to extract the keymaster TA from 73 firmware images and verified the presence of constant KEK's in all of them. The firmware images investigated were distributed from July 2015 until April 2018. We found that all firmware images after April 2018 are using more sophisticated obfuscation schemes to thwart static analyses, which hinders us from reviewing keymaster TAs released after this date.

6 Writing a Keymaster Exploit

An HMAC, as used in the present keymaster TA implementation, constitutes an obstacle for internal software tests, and, therefore, the respective code might not have received as much attention during testing. Based on this intuition, we studied how the keymaster TA uses the keyblob and, indeed, found an exploitable memory corruption flaw.

6.1 Arbitrary Code Execution

We identified a stack-based buffer overflow in a function related to exporting the public key of an RSA key pair. The allocated stack space holding the public key after extracting it from the keyblob is of constant size. Since we can craft our own keyblobs, we can manipulate the size of the key pair's modulus and trigger a classical stack-based buffer overflow.

In order to hijack the control flow of the keymaster TA, we had to overcome the exploit mitigations in place. During our analysis of exploit mitigations, we found that ASLR and stack canaries are present but ineffective. The stack canaries are constants and can be overwritten using the known value. The ASLR lacks entropy and cannot withstand brute-force attacks. Furthermore, a crashing TA is *not* subject to an additional round of address space randomization and loaded to the same base address again. As a consequence, it was straightforward to defeat these mitigations and craft a more powerful primitive, which essentially allows us to execute arbitrary code within the keymaster TA context.

If the KEK described in Section 4 would not be constant already, but a more secure key derived from a device-specific value, as it is the case on Qualcomm devices [14], we would now be able to leak this key from the keymaster TA.

6.2 Privilege Escalation

The ability to execute code in S-EL0, *i.e.,* SW userpace, allows us to directly interface the Trusted OS. In order to map the attack surface of the TC kernel, we identified 174 syscall handlers, which is quite a large attack surface for a trusted computing base. The separation of privileges to invoke a syscall happens via permission flags assigned to TAs during loading. The Trusted OS kernel maintains a task structure containing these flags.

One particularly useful system call enables TAs to map physical memory addresses to their virtual address space. The implementation of this function receives a physical address, a size, and a flag, which we identified as a *secure mode* flag. Only if the secure mode flag is enabled, a set of allowed ranges is checked against the requested physical address. From the device tree, used by the Linux kernel to get information about the physical address layout of the SoC, we were able to derive that the first allowed range is a shared memory region within the memory reserved for the Trusted OS, and the second allowed rule matches the remaining memory except the Trusted OS region.

Since this system call is directly exposed to the TA under our control, we can disable the secure mode flag in order to bypass these checks. Furthermore, this check does not exclude mapping of code and data regions used by the ARM TrustedFirmware running in S-EL3. Thus, it would be ineffective either way.

Using this system call, a TA can map arbitrary physical memory to its virtual address space, even if a higher privilege level is using this physical memory, *i.e.,* the kernel in S-EL1 or the Secure Monitor in S-EL3. We successfully extended our exploit from the previous section in order to escalate privileges to the Trusted OS kernel. We argue that exposing an API with these capabilities is unnecessary. Given the functionality provided by the keymaster TA, we do not see any reason why it has access to such capabilities.

7 Lessons Learned

In this section, we summarize our lessons learned related to the absence of hardware-protected crypto keys and the TCB attack surface of TZ-based TEEs.

7.1 Absence of Hardware-Protected Crypto Keys

In our work, we demonstrated how security-by-obscurity design principles were used despite the presence of a technology that is expected to protect the confidentiality of data by hardware means. Although this is an individual case and we know from independent work focussing on QSEE [14] and Kinibi [16, 31] that confidential data (*e.g.,* crypto keys) are generally secured by hardware, opposed to being hidden in firmware blobs, we can refine our understanding of TZ based on this case. TZ is not an all-in-one solution but rather a construction kit for TEEs. An end product's capabilities regarding integrity and confidentiality guarantees highly depend on the system designer's choices of hardware and software components.

A further important insight relates to the severity of impact inherent to software-based designs. In the reviewed keymaster TA's design, the KEK was not device-specific and reused across devices and firmware versions. The leakage of this key essentially disables the disk encryption's device binding employed by a whole generation of devices and allows for off-device brute-force attacks against the user's PIN, pattern, or password.

7.2 TCB Attack Surface

Complementary to the TCB of Huawei's TC reported by Cerdeira *et al.* [18] in their recent systematization of knowledge on TZ-based TEEs, we found that the aggregated size of TAs (encrypted and non-encrypted) is 7.6 MB. Thus, the TCB in TC's userspace is more than *16* times larger than reported by Cerdeira *et al.* and, according to their data, the largest userspace TCB of all commercially used TEE implementations. Due to TAs being directly exposed to the NW this TCB is not only huge but also comprises a wide attack surface.

Accompanying the large TCB, we demonstrated the risk of memory corruption bugs due to the usage of memory-unsafe languages. This design choice seems to be widely spread, and all major TEE implementations have been victim to memory corruption attacks [12,16,40]. Although the usage of managed runtime environments [37] or memory-safe languages (*e.g.,* Rust [19]) would be a viable way to reduce the risk of memory corruptions significantly, these proposals are not adopted by vendors.

The choice for memory-unsafe languages makes effective exploit mitigation techniques mandatory. In our research, we found that this is not the case for the reviewed system and, according to other research, effective mitigations are also missing on other TEE implementations [4, 16].

After we took over the keymaster TA, we could use the API of the Trusted OS to map physical memory. Although the fact that we could map and modify memory pages of a higher privileged context (*e.g.,* S-EL3) is probably an implementation flaw, the availability of powerful memory mapping functions to the keymaster TA is a design flaw as such. A lack of proper TA API designs and access policies has also been shown on Kinibi [31]. Future designs should consider the principle of least privilege regarding the capabilities of TAs.

8 Related Work

Our work was particularly inspired by all researchers that targeted Huawei devices and published their results. Shen [39] and Stephens [40] works covering full-chain exploits from the NW userland (*i.e.,* N-EL0) to the SW kernel (*i.e.,* S-EL1) on Huawei devices were invaluable resources for us. Complementary to their work, we, for the first time, systematically describe TC's architecture as well as its integration with the Android system. Furthermore, we are the first to review the secure TA loader. Lastly, we are not aware of any other research elaborating on Huawei's keystore system.

Regarding other TEE implementations, Beniamini [16] shares his insights on loading signed TAs into the TEE and the simultaneously arising problem of revocation. If a TEE has no means to reject the loading of an outdated and vulnerable TA, an attacker will always be able to intrude the TCB. Komaromy [28–30] describes Kinibi's architecture in great detail, which is entirely different from TC's architecture due to the usage of a microkernel. Additionally, Lapid and Wool [31] systematically reviewed the Kinibi system and took cache-based side-channel attacks into account. Later, Adamski *et al.* [2] [3] took the research on Kinibi to the next level. Not only do they share their technical insights accumulated during their Kinibi review, more importantly for the community, they also share many of their tools to support other researchers in replicating their research. By open-sourcing all our tools created during the analysis of Huawei's TC, we follow their example. Details about Samsung firmware analysis and approaching the problem of feedback-driven fuzzing for Kinibi TAs can also be found in Akimov's work [5]. Samsung changed their TEE implementation with the Samsung Galaxy S10 model. Only a few resources exist for this TEE called TEEGris. Tarasikov [1] shares his insights from reverse engineering and building a prototypical emulator for TEEGris.

The most insightful research addressing Qualcomm's QSEE has been carried out by Beniamini [13, 15, 16]. In his research, he covers several design and implementation flaws of this platform. Harrison *et al.* [27] approached the problem of re-hosting TEEs in general. With their system called PartEmu, they can run various TEE implementations. Huawei's TC is not part of their evaluation. The semantic gap problem inherent to the NW-SW split of TZ-based TEE architectures was discussed by Machiry *et al.* [32] for the first time. In their recent Systematization of Knowledge on vulnerabilities in TZ-based TEEs, Cerdeira *et al.* [18] summarize many of the above mentioned works and counduct a cross-vendor comparison. With our insights, we complement their work by, for the first time, discussing TC's secure TA loader, which is unique to this platform. Furthermore, we are able to uncover many of the issues discussed in their work on Huawei's platform and show their instantiation in practice.

9 Conclusions

We reviewed Huawei's TEE, called TC, and uncovered several design flaws in different subsystems. We provide a systematic description of TC's components, their interconnections, and their integration with the Android system. Uniquely used on this platform, we study the secure loader of TC responsible for loading confidential TAs into the TEE. We found several issues in the loader's design, like protecting a constant key using white-box cryptography, and were able to break the code confidentiality of encrypted TAs distributed to many Huawei devices. Furthermore, we examined the keystore system and revealed considerable design flaws that allowed us to leak export-protected cryptographic keys from the TEE.

Lastly, we also found an exploitable memory corruption vulnerability enabling us to probe and bypass the exploit mitigations used in TAs on TC. We unveiled that the stack canary and ASLR implementations on this platform are insufficient. It was possible to escalate our privileges to the keymaster TA context inside of the TEE. From within the TEE, we analyzed the TEE kernel's attack surface and discovered an unnecessarily powerful API being exposed to this execution context. Using this API, we were finally able to escalate our privileges to the highest privilege level present on this platform.

Acknowledgments

We thank our shepherd and the anonymous reviewers for their helpful comments. We also thank Huawei for their professional communication. This research was supported by the German Federal Ministry of Education and Research (BMBF) as part of the Software Campus project (Förderkennzeichen: 01/S17045).

Availability

Along with this research, we release TCKit, a collection of tools to reproduce our work: `https://github.com/teesec-research/tckit`.

References

[1] Tarasikov Alexander. Reverse-engineering samsung exynos 9820 bootloader and tz. https://allsoftwaresucks.blogspot.com/2019/05/reverse-engineering-samsung-exynos-9820.html, 2019. Accessed: 2019-08-30.

[2] Adamski Alexandre, Guilbon Joffrey, and Peterlin Maxime. A deep dive into samsung's trustzone (part 1). https://blog.quarkslab.com/a-deep-dive-into-samsungs-trustzone-part-1.html, 2019. Accessed: 2020-03-15.

[3] Adamski Alexandre, Guilbon Joffrey, and Peterlin Maxime. A deep dive into samsung's trustzone (part 2). https://blog.quarkslab.com/a-deep-dive-into-samsungs-trustzone-part-2.html, 2019. Accessed: 2020-03-15.

[4] Adamski Alexandre, Guilbon Joffrey, and Peterlin Maxime. A deep dive into samsung's trustzone (part 3). https://blog.quarkslab.com/a-deep-dive-into-samsungs-trustzone-part-3.html, 2020. Accessed: 2020-07-03.

[5] Akimov Andrey. Launching feedback-driven fuzzing on trustzone tee. https://zeronights.ru/wp-content/themes/zeronights-2019/public/materials/5_ZN2019_andrej_akimovLaunching_feedbackdriven_fuzzing_on_TrustZone_TEE.pdf, 2019. Accessed: 2020-03-17.

[6] ARM. Arm security technology: 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, 2008. Accessed: 2019-08-28.

[7] ARM. Armv8 architecture reference manual, 2020. https://static.docs.arm.com/ddi0487/fa/DDI0487F_a_armv8_arm.pdf.

[8] ARM. Security in an armv8 system, 2020. https://static.docs.arm.com/100935/0100/security_in_an_armv8_system_100935_0100_en.pdf.

[9] ARM. Trusted firmware open governance project. https://developer.arm.com/tools-and-software/open-source-software/firmware/trusted-firmware, 2020. Accessed: 2020-04-15.

[10] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*, pages 41–46. USENIX, 2005.

[11] Gal Beniamini. Android privilege escalation to mediaserver from zero permissions (cve-2014-7920 + cve-2014-7921). https://bits-please.blogspot.com/2016/01/android-privilege-escalation-to.html, 2016. Accessed: 2020-04-02.

[12] Gal Beniamini. Cve-2015-6639 exploit. https://github.com/laginimaineb/cve-2015-6639, 2016. Accessed: 2019-08-28.

[13] Gal Beniamini. Exploring qualcomm's secure execution environment. https://bits-please.blogspot.com/2016/04/exploring-qualcomms-secure-execution.html, 2016. Accessed: 2019-08-28.

[14] Gal Beniamini. Extracting qualcomm's keymaster keys - breaking android full disk encryption. https://bits-please.blogspot.com/2016/06/extracting-qualcomms-keymaster-keys.html, 2016. Accessed: 2019-12-28.

[15] Gal Beniamini. Qsee privilege escalation vulnerability and exploit (cve-2015-6639). https://bits-please.blogspot.com/2016/05/qsee-privilege-escalation-vulnerability.html, 2016. Accessed: 2019-08-28.

[16] Gal Beniamini. Trust issues: Exploiting trustzone tees. https://googleprojectzero.blogspot.com/2017/07/trust-issues-exploiting-trustzone-tees.html, 2017. Accessed: 2019-08-28.

[17] Olivier Billet, Henri Gilbert, and Charaf Ech-Chatbi. Cryptanalysis of a white box AES implementation. In Helena Handschuh and M. Anwar Hasan, editors, *Selected Areas in Cryptography, 11th International Workshop, SAC 2004, Waterloo, Canada, August 9-10, 2004, Revised Selected Papers*, volume 3357 of *Lecture Notes in Computer Science*, pages 227–240. Springer, 2004.

[18] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P), San Francisco, CA, USA*, pages 18–20, 2020.

[19] Eric Evenchick. Rustzone: Writing trusted applications in rust. https://i.blackhat.com/eu-18/Thu-Dec-6/eu-18-Evenchick-RustZone.pdf, 2018. Accessed: 2020-07-16.

[20] Joe Fedewa. Huawei will stop providing bootloader unlocking for all new devices. https://www.xda-developers.com/huawei-stop-providing-bootloader-unlock-codes/, 2018. Accessed: 2020-04-20.

[21] GlobalPlatform. Tee client api specification. https://globalplatform.org/specs-library/tee-client-api-specification/, 2019. Accessed: 2019-12-05.

[22] GlobalPlatform. Tee internal core api specification. https://globalplatform.org/specs-library/tee-internal-core-api-specification-v1-2/, 2019. Accessed: 2019-12-05.

[23] Google. Android hal, 2018. https://source.android.com/devices/architecture/hal.

[24] Google. Android keystore, 2018. https://source.android.com/security/keystore.

[25] Google. Android authentication, 2020. https://source.android.com/security/authentication/.

[26] Google. Android fde, 2020. https://source.android.com/security/encryption/full-disk#storing_the_encrypted_key.

[27] Lee Harrison, Hayawardh Vijayakumar, Rohan Padhye, Koushik Sen, and Michael Grace. Partemu: Enabling dynamic analysis of real-world trustzone software using emulation. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security 2020) (To Appear)*, August 2020.

[28] Daniel Komaromy. Unbox your phone - part i. https://medium.com/taszksec/unbox-your-phone-part-i-331bbf44c30c, 2018. Accessed: 2019-08-28.

[29] Daniel Komaromy. Unbox your phone - part ii. https://medium.com/taszksec/unbox-your-phone-part-ii-ae66e779b1d6, 2018. Accessed: 2019-08-28.

[30] Daniel Komaromy. Unbox your phone - part iii. https://medium.com/taszksec/unbox-your-phone-part-iii-7436ffaff7c7, 2018. Accessed: 2019-08-28.

[31] Ben Lapid and Avishai Wool. Navigating the samsung trustzone and cache-attacks on the keymaster trustlet. In Javier López, Jianying Zhou, and Miguel Soriano, editors, *Computer Security - 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part I*, volume 11098 of *Lecture Notes in Computer Science*, pages 175–196. Springer, 2018.

[32] Aravind Machiry, Eric Gustafson, Chad Spensky, Christopher Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. BOOMERANG: exploiting the semantic gap in trusted execution environments. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, 2017.

[33] Team OpenKirin. Openkirin source code repositories, 2020. https://github.com/OpenKirin/.

[34] Qualcomm. Qualcomm mobile security. https://www.qualcomm.com/solutions/mobile-computing/features/security, 2018. Accessed: 2020-04-15.

[35] Samsung. Trustonic for knox. https://news.samsung.com/global/samsung-and-trustonic-launch-trustonic-for-knox-delivering-a-whole-new-level-of-trust-enhanced-experiences-on-samsung-mobile-devices, 2015. Accessed: 2020-04-15.

[36] Samsung. Samsung teegris. https://developer.samsung.com/teegris/overview.html, 2020. Accessed: 2020-04-15.

[37] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Using ARM trustzone to build a trusted language runtime for mobile applications. In Rajeev Balasubramonian, Al Davis, and Sarita V. Adve, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*, pages 67–80. ACM, 2014.

[38] Thorsten Schreiber. Android binder. http://www.nds.rub.de/media/attachments/files/2012/03/binder.pdf, 2011. Accessed: 2019-12-05.

[39] Di Shen. Attacking your Trusted Core. https://www.blackhat.com/docs/us-15/materials/us-15-Shen-Attacking-Your-Trusted-Core-Exploiting-Trustzone-On-Android.pdf, 2015. Accessed: 2019-11-28.

[40] Nick Stephens. Behind the pwn of a trustzone. https://www.slideshare.net/GeekPwnKeen/nick-stephenshow-does-someone-unlock-your-phone-with-nose, 2017. Accessed: 2019-08-28.

[41] Huawei Technologies. Emui 8.0 security technical white paper. https://consumer-img.huawei.com/content/dam/huawei-cbg-site/en/mkt/legal/privacy-policy/EMUI8.0SecurityTechnologyWhitePaper.pdf, 2017. Accessed: 2020-07-20.

[42] Huawei Technologies. Emui 9.0 security technical white paper. https://consumer-img.huawei.com/content/dam/huawei-cbg-site/en/mkt/legal/privacy-policy/EMUI%209.0SecurityTechnologyWhitePaper.pdf, 2018. Accessed: 2020-07-20.

A Appendix

A.1 Android Keystore System

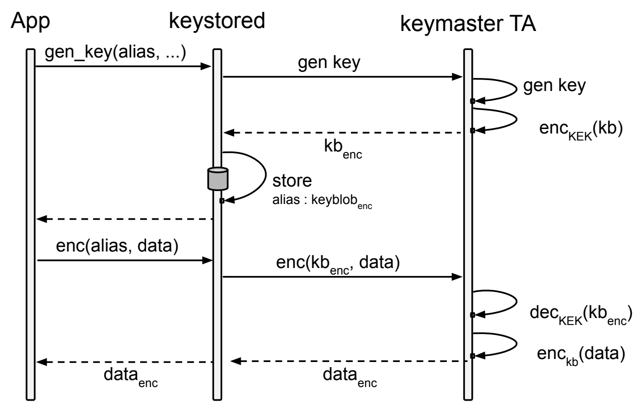

Figure 5: An app requests cryptographic operations from `keystored`. `keystored` uses the keymaster TA hosted in the TEE to carry out the key generation and encryption requests. The keyblob (kb), containing the generated key, is never exposed to NW memory in plaintext. Only the keymaster TA can decrypt its plaintext using the KEK.

```c
struct keyblob {
  uint8_t hmac[32];
  uint8_t iv[16];
  uint8_t magic[4];
  uint32_t unknown;

  uint32_t keymaterial_offset;
  uint32_t keymaterial_size;
  uint32_t key_params1_count_offset;
  uint32_t key_params2_count_offset;
  uint32_t key_params1_data_offset;
  uint32_t key_params1_data_size;
  uint32_t hidden_params_count_offset;
  uint32_t hidden_params_data_offset;
  uint32_t hidden_params_data_size;
  uint32_t keyblob_size;
  uint8_t blob[]; // C99 FAM
}
```

Listing 3: The keyblob structure used by TC's keymaster TA. We obtained the keys to generate the HMAC and to encrypt or decrypt the key material which is part of the `blob` member.

Our static analysis of the keymaster TA and dynamic analysis of the NW keystore system components revealed a keyblob structure, as shown in Listing 3. The actual `key_material`, `key_params1`, `key_params2`, `key_params_data`, and `hidden_params` are part of the last structure member, which is a flexible array member. The first member is an HMAC of the entire keyblob, including the dynamically sized `blob` member. The `key_material` and the `hidden_params` are encrypted.