Trust Dies in Darkness: Shedding Light on Samsung's TrustZone Keymaster Design

Alon Shakevsky
shakevsky@mail.tau.ac.il

Eyal Ronen
eyal.ronen@cs.tau.ac.il

Avishai Wool
yash@eng.tau.ac.il

Tel-Aviv University

Abstract

ARM-based Android smartphones rely on the TrustZone hardware support for a Trusted Execution Environment (TEE) to implement security-sensitive functions. The TEE runs a separate, isolated, TrustZone Operating System (TZOS), in parallel to Android. The implementation of the cryptographic functions within the TZOS is left to the device vendors, who create proprietary undocumented designs.

In this work, we expose the cryptographic design and implementation of Android's Hardware-Backed Keystore in Samsung's Galaxy S8, S9, S10, S20, and S21 flagship devices. We reversed-engineered and provide a detailed description of the cryptographic design and code structure, and we unveil severe design flaws. We present an IV reuse attack on AES-GCM that allows an attacker to extract hardware-protected key material, and a downgrade attack that makes even the latest Samsung devices vulnerable to the IV reuse attack. We demonstrate working key extraction attacks on the latest devices. We also show the implications of our attacks on two higher-level cryptographic protocols between the TrustZone and a remote server: we demonstrate a working FIDO2 WebAuthn login bypass and a compromise of Google's Secure Key Import.

We discuss multiple flaws in the design flow of TrustZone based protocols. Although our specific attacks only apply to the ≈100 million devices made by Samsung, it raises the much more general requirement for open and proven standards for critical cryptographic and security designs.

1 Introduction

Beyond their usage in many and various daily activities, smartphones are increasingly used for many security-critical tasks, such as the protection of sensitive data (messages, images, files), cryptographic key management [24], FIDO2 web authentication [65], Digital Rights Management [64] (DRM), mobile payment services [53] (e.g., Samsung Pay) and enterprise identity management [53].

Simultaneously, smartphones are becoming more and more complex and present an increasingly larger attack surface. The result is that they have become a major target for malware and malicious attackers. There have been many public exploits that allow an attacker to escalate privileges in the Android OS, gaining execution as root or even as the OS kernel [9, 14, 20, 21, 39]. Ideally, such attacks should not be able to compromise the devices' security-critical tasks.

Trusted Execution Environments (TEEs) are largely used in modern mobile devices to provide an isolated environment for execution of Trusted Applications (TAs) that can securely perform security-critical tasks. They have a relatively small codebase and limited APIs.

In contrast, the Rich Execution Environments (REEs), such as Android OS, cannot be fully audited and trusted (due to their complexity). An isolated TEE can be used alongside the REE to implement security-sensitive functions. This makes it harder for an attacker to compromise these functions, as the attack surface is significantly reduced and is limited to communication with the TEE.

In other words, the goal of the TEE is to withstand attacks from a fully compromised REE, including by privileged adversaries with kernel or root capabilities.

ARM is the most widely used processor in the mobile and embedded markets [46], and it provides TEE hardware support with ARM TrustZone [3, 8]. TrustZone separates the device into two execution environments:

1. A non-secure REE where the "Normal World" operating system runs.

2. A secure TEE where the "Secure World" operating system runs.

The REE and TEE use separate resources (e.g., memory, peripherals), and the hardware enforces the protection of Secure World.

In most mobile devices, the Android OS runs the non-secure Normal World. As for the Secure World, there are more choices. Even among Samsung devices, there are at

least three different TrustZone Operating Systems (TZOS) in use (see Section 2.2).

The Android Keystore [28] provides hardware-backed cryptographic key management services through a Hardware Abstraction Layer (HAL) that vendors such as Samsung implement. The Keystore exposes an API to Android applications, including cryptographic key generation, secure key storage, and key usage (e.g., encryption or signing actions). Samsung implements the HAL through a Trusted Application (TA) called the Keymaster TA, which runs in the TrustZone.[1] The Keymaster TA performs the cryptographic operations in the Secure World using hardware peripherals, including a cryptographic engine.

The Keymaster TA's secure key storage uses *blobs*: these are "wrapped" (encrypted) keys that are stored on the REE's file system. The "wrapping", "unwrapping", and usage of the keys are done inside the Keymaster TA using a device-unique hardware AES key. Only the Keymaster TA should have access to the secret key material; the Normal World should only see opaque key blobs.

Although it is crucial to rigorously verify and test such cryptographic designs, real-world TrustZone implementations received relatively little attention in the literature. We believe that this is mainly due to the fact that most device vendors do not provide detailed documentation of their TZOS and proprietary TAs and share little-to-no information regarding how the sensitive data is protected. To advance and motivate this research area, we decided to use the leading Android vendor Samsung as a test case. We reversed-engineered the full cryptographic design and API of several generations of Samsung's Keymaster TA, and asked the following questions:

Does the hardware-based protection of cryptographic keys remain secure even when the Normal World is compromised? How does the cryptographic design of this protection affect the security of various protocols that rely on its security?

1.1 Our Contribution

In this work, we focus on the Keymaster TA used by Samsung's flagship devices, including the Samsung Galaxy S8, S9, S10, S20, and S21. For the first time, we expose its cryptographic design, and unveil severe design flaws that can allow an attacker to extract hardware-protected key material. We present an IV reuse attack on AES-GCM that allows the attackers to extract keys from hardware-protected key blobs; and a downgrade attack that makes even the latest Samsung flagship devices vulnerable to our IV reuse attack. As summarised in Table 1, our attacks affect over 100 million devices [66].

We also show the implications of these vulnerabilities on bypassing key usage restrictions and on the security of two higher-level cryptographic protocols between the TrustZone and a remote server. We demonstrate working Proof-of-Concept attacks on Galaxy S9, S10, and the latest S21 model. To summarise our contributions:

1. We expose the proprietary Keymaster TA implementation in Samsung devices, focusing on its key derivation and blob encryption implementation.

2. We show that the hardware protection in Samsung Galaxy S9 devices is vulnerable to an IV reuse attack on AES-GCM, allowing the extraction of protected key material.

3. We show a downgrade attack on Samsung Galaxy S10, S20, and S21 devices, making them vulnerable to our IV reuse attack.

4. We evaluate the impact of our attacks and describe how to exploit them to misuse the Keystore key attestation to bypass FIDO2 WebAuthn login and compromise Google's Secure Key Import.

5. We discuss the root causes leading to each of the vulnerabilities we identified, focusing on possible countermeasures and problems in the closed cryptographic design methodology.

In order to implement our attacks, we developed an open-source Keymaster client that we will make available on [58]. Our client interacts with the Keymaster TA without passing through the Keymaster HAL API, which allows us full control of the input passed to the Trusted Application.

1.2 Responsible Disclosure

We reported our IV reuse attack on S9 to Samsung Mobile Security in May 2021. In August 2021 Samsung assigned CVE-2021-25444 with High severity to the issue and released a patch that prevents malicious IV reuse by removing the option to add a custom IV from the API. According to Samsung [57], the list of patched devices includes: S9, J3 Top, J7 Top, J7 Duo, TabS4, Tab-A-S-Lite, A6 Plus, A9S.

We reported the downgrade attack on S10, S20 and S21 in July 2021. In October 2021 Samsung assigned CVE-2021-25490 with High severity to the downgrade attack and patched models that were sold with Android P OS or later, including S10, S20, and S21. The patch completely removes the legacy key blob implementation.

1.3 Structure of the Paper

Section 2 provides some background on TrustZone. In Section 3 we dissect the Keymaster TA in Samsung Galaxy S8, S9, S10, S20, and S21 devices. In Section 4 we present an IV reuse attack against hardware-protected keys as well as a downgrade attack. In Section 5 we show how our attacks

[1] For brevity, we refer to TrustZone-based TEEs simply as "TrustZone".

Table 1: Susceptibility of Samsung Galaxy devices to IV reuse and downgrade attack - ✓ means vulnerable and ✗ means not.

Device	IV reuse attack	Downgrade attack
S8	✗	✗
S9	✓	✓
S10	✗	✓
S20/S21	✗	✓

break the composability of higher-level cryptographic protocols, focusing on Google's Secure Key Import and FIDO2 WebAuthn. In Section 6 we discuss the root causes of the attacks and gaps in the higher-level protocols' design. In Section 7 we survey related work in TrustZone research, and we conclude our work in Section 8. Multiple appendices provide technical details.

2 Background

2.1 AES GCM

The Advanced Encryption Standard (AES) is the most widely used symmetric block cipher. Galois Counter Mode (GCM) is a mode of operation for block ciphers that provides Authenticated Encryption. AES-GCM is a stream cipher that uses AES-CTR (Counter Mode) and the Galois Message Authentication Code (GMAC) internally.

Like every stream cipher, AES-GCM is vulnerable to Initialization Vector (IV) reuse attacks. When an IV is reused while encrypting with the same key, the resulting keystream is identical. In that case, knowledge of one plaintext immediately reveals the other. Furthermore, in AES-GCM, Joux [38] showed how IV reuse could be exploited to break authentication and create new valid messages.

2.2 ARM TrustZone

The ARM TrustZone technology [7] adds an additional virtual processor mode called "Secure World" that complements the "Normal World". The two modes are separated and can communicate using the "Secure Monitor" (running in the highest EL3 execution level) or by memory mapping of "World Shared Memory". The separation allows to implement a TEE, since a compromised Normal World will not be able to access the memory of the Secure World. Fig. 1 shows the components in each exception level in the TrustZone architecture. See Appendix A for a detailed overview of ARM TrustZone and ARM exception levels.

As the implementation of the TZOS is left to vendors, there are multiple implementations by various vendors, including:

- Qualcomm Secure Execution Environment (QSEE) by Qualcomm: used in Google Pixel devices and in Snapdragon models of Samsung Galaxy devices.
- Kinibi by Trustonic: used in older Exynos models of Samsung Galaxy devices, prior to S10.
- TrustedCore (TC) by Huawei.
- TEEGRIS by Samsung (used in newer Exynos models of Samsung Galaxy devices).

2.3 Trusted Applications

A Trusted Application (TA) is a program that runs in the TEE and exposes security services to Android client applications. The application can open a session with the TA and invoke commands within the session. After receiving a command, a TA parses the commands input, performs required processing and sends a response back to the client. See the extended version of this paper [59] for more details on the TEE client API. Control is transferred to the TA via the dedicated SMC (Secure Monitor Call) instruction, and the TA and Normal World application usually exchange arguments and output using a shared memory buffer called World Shared Memory. As performing SMCs requires EL1 privileges, a device driver in the Android kernel handles the communication with the TA and exposes an API for Normal World applications.

2.4 Android Hardware Backed Keystore

The Android Keystore [24] allows Normal World applications to perform cryptographic operations while protecting the cryptographic keys from extraction or unauthorized use. On devices with TrustZone technology, Keystore utilizes the TEE to perform cryptographic operations. This Hardware-Backed Keystore implementation is called the Keymaster TA.

The Keystore's main functions are: key generation and import, asymmetric encryption/decryption/signing/verification, symmetric encryption/decryption and generation/verification of symmetric MACs.

Keys can be generated and used inside the TrustZone by the Keymaster TA, and are thus protected from any Normal World attacker. However, the Keymaster TA relies on the Normal World application to store the keys [28]. To protect the key material, the keys are encrypted or "wrapped" with a hardware-derived key inside the TrustZone. The encrypted key "blobs" are then passed to the Normal World to be stored.

Fig. 2 shows a simplified overview of an Android application using the Keymaster TA. The general flow runs as follows:

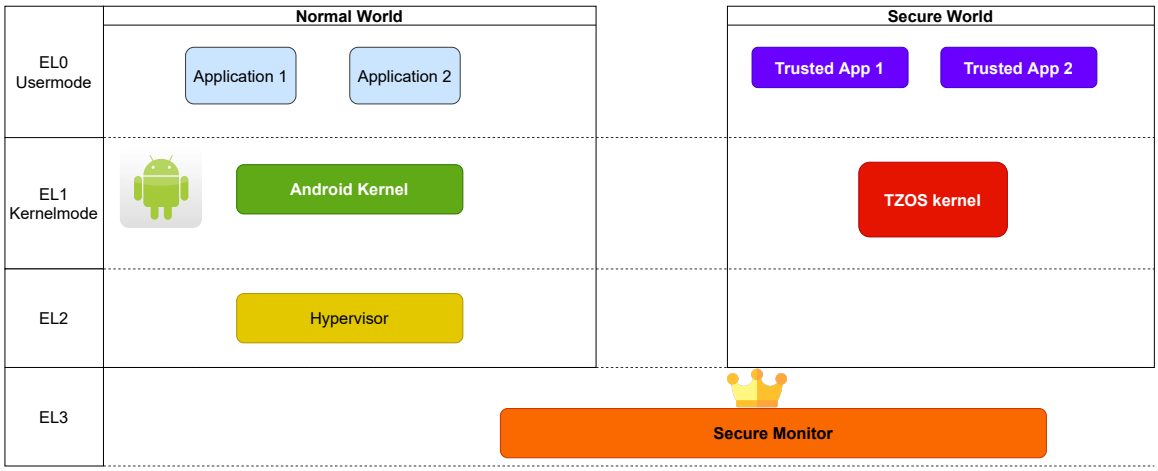


Figure 1: The TrustZone software and hardware isolation architecture [2]

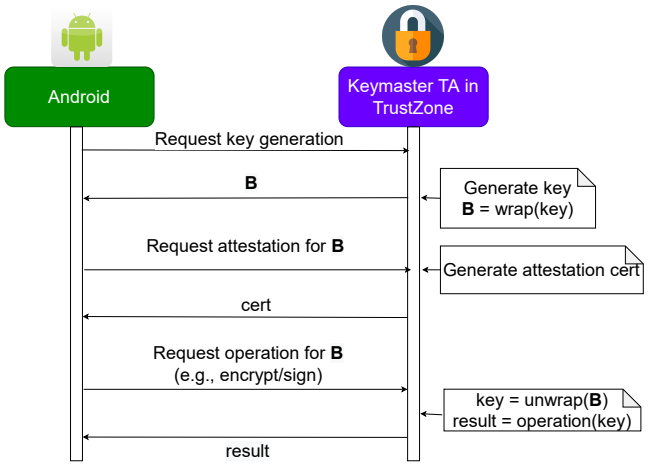


Figure 2: A usage example of an Android application using a Hardware-Backed Keystore (i.e., the Keymaster TA) for cryptographic key management: key generation, attestation, and encryption/signature.[2]

1. The Android application requests a new key to be generated. The Keymaster TA generates a new key, and encrypts it using a hardware-derived key. The resulting encrypted key blob *B* is passed back to the Android application.

2. The Android application saves the blob *B* in the Normal World's file system.

3. Some protocols such as WebAuthn [65] require that the Android application must prove that a specific key was generated securely by the Keymaster TA. In these cases, the application can pass the encrypted blob *B* to the Keymaster TA and ask it to generate an attestation certificate. For details on the attestation process see Section 2.5.

4. The application can ask the Keymaster TA to perform a cryptographic operation on its behalf. For example, to sign a message, the application passes the encrypted blob *B* and the message to be signed to the Keymaster TA. The Keymaster TA decrypts the blob to recover the signing key. It then uses the key to sign the message and return the resulting signature to the application in the Normal World.

The Keystore protects keys from extraction using the following measures:

1. Key material is not present in the application memory, hence compromising the application will not lead to key material extraction.

2. The Keystore can provide access control that restricts the usage of keys in various ways, such as restricting a key's purpose (e.g., encryption only), setting an expiration date, rate-limiting, or requiring user authentication (e.g., passing biometric authentication/unlocking the screen).

3. Hardware binding: Supported devices can bind keys to the secure hardware (i.e., the TEE), so they cannot be used outside of the secure hardware on that device.

In order to support multiple TZOS implementations, the Android Keystore uses a HAL, as we shall describe in Section 3.2. Every TZOS implements the HAL for Android services that require hardware support, usually by having a TA that performs the needed operations. This is the case for Samsung's Keymaster TA, which is used to implement Android Hardware-Backed Keystore and is the main focus of our research.

[2]Designed using resources from Flaticon.com

2.5 Android Key Attestation

Keystore Key Attestation [33, 67] allows remote parties to verify that a key was generated within the secure hardware by having the Keymaster TA generate a certificate chain whose root certificate is Google (or Samsung, for applications such as KNOX attestation). This allows the remote party to trust public-keys generated within the TrustZone without trusting the Normal World, despite the fact that all the communications with the Keymaster TA go through the Normal World. As we show in Section 5, Secure Key Import and FIDO2 WebAuthn use key attestation (using Google's root certificate) exactly for this purpose.

2.6 The Attack Model

In this paper, we assume that an attacker can fully compromise the Normal World, e.g., an attacker with root or even kernel privileges. Moreover, we assume that the attacker is able to compromise the Normal World without setting the bootloader fuses that are attested by the Keymaster (e.g., bootloader unlocked, Samsung's warranty bit). Such attacks were shown by [21,44]. The attacker aims to compromise data that is secured by the Trusted World, such as Keystore hardware-protected keys, or higher-level protocols that rely on remote attestation (e.g., cloning a FIDO2 token or by stealing a key that was securely imported).

We follow the Android Platform Security Model by Mayrhofer et al. [42] that states that the hardware protection and isolation of cryptographic keys offered by TrustZone and the Keymaster TA should prevent any key compromise even by such an attacker. This is consistent with the attack model used by Harrison et al. [36] as well as Lapid and Wool [40].

Note that the attacks described in Sections 4 and 5 do not require us to actually run code in the Android Kernel. For our attack, we only require code execution in EL0 (Android user mode) with sufficient privileges to read key blobs, and appropriate SELinux permissions to communicate with the TZOS drivers. For instance, a vulnerability in the Android Keystore user mode daemon/HAL (such as [37]) would likely suffice. Alternatively, a root malware or a supply-chain attack that patches the Keymaster HAL can be used for both attacks.

In our experiments, we used Samsung Galaxy S9, S10, and S21 devices, rooted using Magisk [68]. Note that when we rooted the devices, the Samsung KNOX warranty fuse was set. This does not affect our attacks as we don't target any KNOX functionality.

3 Dissecting the Keymaster TA

3.1 Survey of the Keymaster TA Family

In this paper we focus on Samsung's implementation of the Keymaster TA and its new TZOS named TEEGRIS. Like other TZOSs and TAs, Samsung's implementation is vendor-specific and is a proprietary closed-source system with little-to-no documentation available. To understand the cryptographic design implemented by the Keymaster TA, we statically analyzed the binaries of firmwares and TAs in 3 TZOS's (TEEGRIS, Kinibi and QSEE) using Ghidra [45]. Additionally, we used the Samsung Open Source [55] website to download the Android kernel sources for our device.

Overall, we evaluated 26 firmwares for both Exynos and Snapdragon models of S8, S9, S10, S20, and S21 (including variants such as S9+, Note9, S10+, S20+, etc.) published between 2018 and 2021. We found that Keymaster TA's code base is extremely similar in theses firmwares (except the S8), even across TZOSs (Kinibi/QSEE/TEEGRIS).

In our analysis, we noticed that there is a significant difference between the Keymaster TA in S8 and in the newer models. As we shall see in Section 4, one specific code change introduced in the S9 makes it and all the newer models vulnerable to attacks. Although S20 and S21 models include the more secure Strongbox Keymaster functionality (using a dedicated tamper-resistant hardware security module), they are still vulnerable to our attacks as they share the same vulnerable cryptographic design and API.

As Kinibi and QSEE have been more thoroughly studied by the security community [12, 48, 49], when we discuss details we refer to TEEGRIS unless otherwise noted. See the extended version of this paper [59] for details on firmware analysis.

3.2 The Keymaster HAL

The Keymaster HAL is an interface between Android and the vendor-specific Keymaster implementation. The Android documentation provides reference guidelines for implementers of Keymaster HALs [30]. It is implemented in the Android user mode and communicates with the Keymaster TA using kernel drivers and World Shared Memory buffers (see Fig. 3). The extended version of this paper [59] contains a more detailed overview of the Keymaster HAL in TEEGRIS.

Android provides an open-source API for functions that the Keymaster should implement [29]. This API includes many functionalities such as key generation, key import, and cryptographic operations such as encrypt/decrypt/sign/verify using the keys stored in the encrypted blobs.

The Keymaster HAL API in TEEGRIS is implemented in a number of undocumented shared-objects that use TEEGRIS kernel drivers. To explore the Keymaster TA we reversed engineered the Keymaster HAL and implemented our own Keymaster client—an Android process that sends our custom requests to the Keymaster TA without any input validation or filtering.

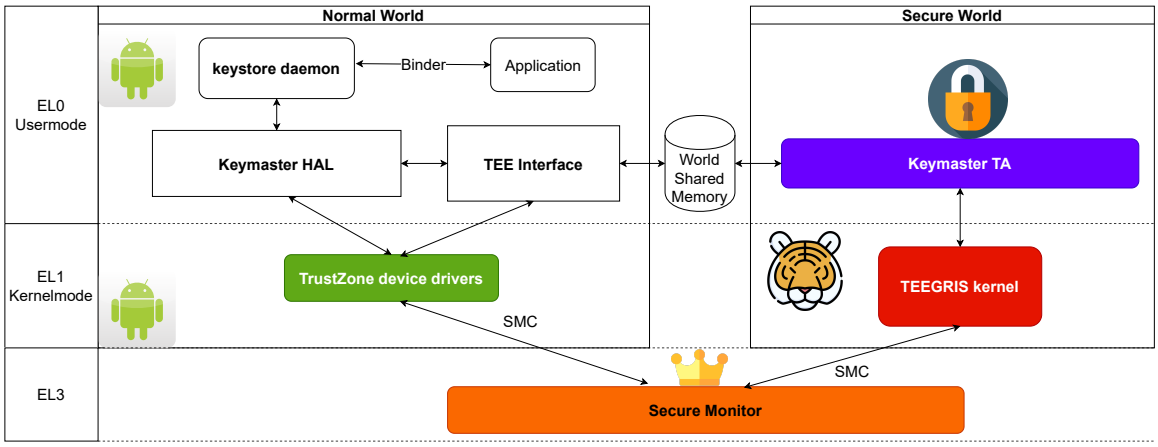


Figure 3: Overview of the Hardware backed keystore[2]

v15 blob
"MDFPP HW Keymaster HEK v15\x00"
"ID"
"\x02\x00\x00\x00"
"id"
"DATA"
"\x04\x00\x00\x00"
"data"

v20-s9 blob
"MDFPP HW Keymaster HEK v20\x00"
root_of_trust
"ID"
"\x02\x00\x00\x00"
"id"
"DATA"
"\x04\x00\x00\x00"
"data"
integrity_flags

v20-s10 blob
"MDFPP HW Keymaster HEK v20\x00"
root_of_trust
"ID"
"\x02\x00\x00\x00"
"id"
"DATA"
"\x04\x00\x00\x00"
"data"
integrity_flags
hek_randomness

Figure 4: The three KDF versions for HDK salt derivation, assuming the Application ID is "id" and the Application Data is "data". The salt value is the SHA256 digest of the concatenation of the values. Values shaded in blue are optional, values shaded in green are new to that version.

3.3 Key Blob Encryption

The main focus of our research is how key blobs are decrypted/encrypted. Based on our analysis and reverse engineering of the proprietary Keymaster TA, we will now describe the process at a high level. For a detailed description of the control flow inside the Keymaster TA, see Appendix B. As mentioned in Section 2.4, the Keymaster TA encrypts key material inside a blob. This protects the key material from extraction while allowing it to be stored by the Normal World application. As we discovered during our research, each blob contains an encrypted part that includes the key material and various parameters. The blob also contains a clear-text part containing the information required for decryption, such as the IV and AAD used in the encryption.

The cryptographic foundation which the Keymaster TA relies upon is a permanent, hardware, device-unique 256-bit AES key called the Root Encryption Key (REK) [61]. This key is only present in the secure hardware cryptographic engine—the Keymaster TA cannot access it directly.

Each key blob is encrypted using its own Hardware Derived Key (HDK) that is derived from the REK. A blob's HDK is derived using a Key Derivation Function (KDF) that mixes the REK with a blob-specific salt value. The KDF itself is accessed by the Keymaster TA through a TZOS-internal ioctl API. The salt for deriving the HDK is computed as the SHA-256 digest of a concatenation of several values. Samsung's salt-generation method evolved between device models, as we discuss next.

3.4 KDF Versions of Key Blobs

The National Information Assurance Partnership (NIAP) created the Mobile Device Fundamentals Protection Profile (MDFPP) that includes core security requirements for devices. The constant string used in the salt derivation (in all key blobs flavors) suggests that Samsung complies with the certification, and indeed Samsung devices are MDFPP CC certified.

In our analysis, we identified three different blob salt-derivation versions, which we call v15, v20-s9, and v20-s10. The differences between the versions are the values used in

the string that is hashed to generate the salt for the key derivation. All versions can include optional values that are called Application ID and Application Data. These values are set by the Normal World. Fig. 4 shows example strings for each of the three blob versions.

The first blob key version, which we call v15, is the version used in Galaxy S8. The salt in v15 key blobs only depends on the application ID and data set by the Normal World (and a constant string).

Galaxy S9 introduced a new version which we call v20-s9. This version adds two new values to the SHA-256 digest, which we call `root_of_trust` and `integrity_flags`. This might be due to a new MDFPP regulation that requires derived keys to be bound to the device's integrity.

The `root_of_trust` is "a collection of values that defines key information about the device's status" [33]. The Keymaster TA computes `integrity_flags` based on the integrity status of the device (a normal device should have 0, a rooted device has value 7).

In our analysis, we noticed that although S9 uses the new salt version v20-s9 by default, it still includes code that implements the older v15 blob version. The Keymaster TA API exposed the option to use this older version to the Normal World. As we were not able to find any use for this option by the Normal World, we believe that this is latent code that is never used.

On S10, S20, and S21 devices, we found a revised KDF that is very similar to the v20-s9: it uses exactly the same strings, root of trust and integrity flags, with one crucial addition: in v20-s10 the salt also includes a fresh per-blob 16-byte random value `hek_randomness` generated inside the TrustZone. Similar to S9, we also found latent code that implements v15 and is exposed to the Normal World.

4 Attacking the Keymaster TA

This section describes two attacks against the Keymaster TA that allow us to extract hardware-protected key material. Table 2 includes a summary of the Samsung Galaxy devices that we've examined and their susceptibility to IV reuse.

4.1 IV Reuse Attack on v15 and v20-s9 Blobs

As we discussed in Section 3, the wrapping key used to encrypt the key blobs (HDK) is derived using a salt value computed by the Keymaster TA. In v15 and v20-s9 blobs, the salt is a deterministic function that depends only on the application ID and application data (and constant strings), which the Normal World client fully controls. This means that for a given application, all key blobs will be encrypted using the *same key*. As the blobs are encrypted in AES-GCM mode-of-operation, the security of the resulting encryption scheme depends on its IV values never being reused.

Surprisingly, we discovered that the Android client is allowed to *set* the IV when generating or importing a key. All that is necessary is to place an attacker-chosen IV as part of the key parameters, and it is used by the Keymaster TA instead of a random IV. As the Normal World also controls the application ID and application data, this means that an attacker can force the Keymaster TA to reuse the same key and IV that were previously used to encrypt some other v15 or v20-s9 blobs. Since AES-GCM is a stream cipher, the attacker can now recover hardware-protected keys from key blobs.

Given a key blob B_A wrapping an unknown key K_A, an attacker can import another key blob B_B with a known (sufficiently long) key K_B that was encrypted using the same IV and the same salt. To do so the attacker first extracts the IV from the key blob B_A, and passes this IV and the same application ID and data to the Keymaster TA's import key function. The known K_B will be encrypted using the same blob encryption key HDK (as the salt only depends on the application ID and data) and the same IV (by construction) as was used to encrypt K_A.

As with any stream-cipher encryption, we can now use our knowledge of K_B together with the ciphertexts B_A and B_B to recover K_A. Let us denote the key-stream created from key HDK with a given IV as $E(\text{HDK}, \text{IV})$, then:

$$\begin{aligned} & B_A \oplus B_B \oplus K_B \\ & \quad = (E(\text{HDK}, \text{IV}) \oplus K_A) \oplus (E(\text{HDK}, \text{IV}) \oplus K_B) \oplus K_B \\ & \quad = K_A \oplus K_B \oplus K_B = K_A \end{aligned}$$

All key blobs created on the Galaxy S9 are vulnerable, as its default blob version is the vulnerable v20-s9. However, on S10, S20, and S21 devices, the default version is v20-s10, and its salt is randomised by the `hek_randomness` field (recall Fig. 4), hence each blob is encrypted with a uniquely derived encryption key. Although the attacker can still cause IV reuse, this reuse is not exploitable. We note that, although S8 can create only blobs with version v15, its Keymaster TA ignores the IV parameter in key generation and key import functions, i.e., it does not allow us to set the IV and is thus not vulnerable to any of our attacks.

To demonstrate our IV reuse attack, we implemented it on Galaxy S9 (all key blobs), S10, and S21 (by forcing the creation of v15 key blobs), and we were able to recover securely generated AES, RSA, and ECDSA keys from encrypted blobs.

4.2 The Downgrade Attack

On S10 and later models, the default blob version is v20-s10. However, to our surprise, we discovered the existence of latent code that allows the Normal World to request the creation of v15 blobs by simply passing an "encryption version" parameter with a specific value. Although the Keymaster HAL API does not normally pass this parameter, its existence can be exploited by a privileged attacker to force all new blobs to version v15.

Table 2: Summary of Samsung Galaxy devices Keymaster features that make them vulnerable to IV reuse - ✓ means true and ✗ means false

Device	Default blob version	Deterministic HDK	Can attacker set IV	Vulnerable to IV reuse	Can attacker downgrade to v15
S8	v15	✓	✗	✗	N/A
S9	v20-s9	✓	✓	✓v15, ✓v20-s9	✓
S10	v20-s10	✓v15, ✗v20-s10	✓	✓v15, ✗v20-s10	✓
S20/S21	v20-s10	✓v15, ✗v20-s10	✓	✓v15, ✗v20-s10	✓

This downgrade attack makes newer devices, including Galaxy S10, S20, and S21 vulnerable to the IV reuse attack (after the Normal World is compromised). In Section 5 we show that this can be exploited to attack security-critical protocols such as Google's Secure Key Import and FIDO2 WebAuthn.

To demonstrate our findings, we wrote a GDB script (see the extended version of this paper [59]) that intercepts the call to the Keymaster HAL in the Normal World. We hooked the key generation function and modified the passed parameters in flight to add the encryption version parameter with the value $0xf$ indicating v15. This caused the Keymaster TA to always generate v15 key blobs and allowed us to recover the encrypted keys using the IV reuse attack. Creative malware authors could achieve a similar effect in other ways, such as always returning a pre-computed v15 key blob, or possibly patching the Keymaster HAL so that it always downgrades to v15 blobs.

4.3 Persistence of v15 Blobs

According to the Keymaster API [29], key blobs become "old" when the Keymaster device is updated or when the Normal World OS is upgraded to a newer version. When a key becomes "old", the API functions return a special error code that indicates that the key must be "upgraded". The Keymaster API exposes the `upgradeKey` method which unwraps (decrypts) the key, examines the OS versions inside the key parameters and compares them to the current OS version. If the current OS version is higher it "upgrades" the key by wrapping (encrypting) it again (and adding the current OS version to the key parameters list).

However, we found that the key parameters that are used in the new blob's wrapping are the same as those in the old key. As any key blob created by our downgrade attack includes the encryption version parameter, "upgrading" such a downgraded v15 key blob will result in a new but still vulnerable v15 blob.

According to Samsung [57], this is the intended behavior, and since the S10 and newer devices have v20-s10 as the default version, no v15 key should exist "in the wild". However, this behavior of the `upgradeKey` function allows our downgrade to persist through firmware updates.

5 Implications of the Attacks

In this section, we explore the possible implications of the attacks described in Section 4. Naturally, an attacker that controls the Normal World can simply ask the TrustZone to locally perform any permitted individual cryptographic operation on their behalf. However, we shall see that by extracting the keys, the attacker is able to bypass key usage limitations. Moreover, they can perform advanced attacks that break cryptographic protocols with remote parties, protocols specifically designed to utilize the security guarantees offered by the Secure World.

As we have seen in the recent line of exploits observed in the wild [14, 23], such normal-world compromises can be done remotely, covertly, and without changing the state of the bootloader fuses. The attacker then aims to compromise the security properties of the higher-level protocols such as WebAuthn and Secure Key Import. While our attacks cannot decrypt v20-s10 keys generated before the compromise, the downgrade attack can break the security of remote attestation on the latest devices (including S10, S20, and S21) by compromising any key that is generated after the covert compromise.

5.1 Authentication and Confirmation Bypass

The Keymaster TA can be used to enforce restrictions on the use of cryptographic keys to prevent misuse of the keys without the user's consent or knowledge (e.g., by an attacker controlling the Normal World).

For example, keys can be limited for specific use, such as signature only. Moreover, applications can create "authentication-bound" keys that require the user to be recently authenticated in order to use the key, e.g., by specifying a timeout since the last time the user entered their passcode, or requiring a biometric prompt authentication [27].

The usage and authentication requirements are enforced by the Keymaster TA when it is attempting to use the blob. This

prevents attackers from using the keys on a device without the user's consent — e.g., the secure hardware can refuse to use the key if the user is not authenticated. For instance, the attacks shown by Cooijmans et al. [18] and by Breński et al. [15] fail when the restrictions are enforced.

Similarly, Protected Confirmation [19] allows signing keys to be used only if the user provides confirmation of the data to be signed (via a trusted UI interface). Google discussed some use cases for Protected Confirmation [19], including Medical applications, such as an injection of insulin by Bigfoot Biomedical [13], and Enterprise applications such as Two-Factor Authentication including Duo Mobile [10].

However, the security of these restrictions is based on the assumption that the protected keys cannot be extracted from inside the TrustZone. An attacker can use our attacks to extract the keys and completely bypass any restrictions. For example, they can use "authentication-bound" without knowledge of the user's password or sign payment transactions without user interaction or consent. Moreover, they can continue to use the keys even if they no longer have access to the device. This is especially useful if the attacker has only limited-time physical access to the device (e.g., search by border agents, law enforcement, evil housekeeping).

5.2 Extracting Keys from Secure Key Import

The Keymaster TA supports Secure Key Import [69] which allows applications to securely provision existing keys into Keystore. The protocol's goal is to allow servers to securely share a secret key with an Android device while preventing the key from being intercepted or extracted from the device, even if the device is compromised. The key is encrypted by the server and only decrypted inside the secure hardware. Thus it is bound to the device, allowing it to be used in various scenarios such as SSH/RDP, DRM, secure payment, etc. For example, Google Pay uses Secure Key Import to provision some of its keys [69].

Secure Key Import allows a server to securely send some key material K to the device. A simplified version of the protocol and our attacks is shown in Fig. 5. For the full protocol details, see [25]. In high-level, Secure Key Import works as follows:

1. The application requests the Keymaster TA to generate an RSA private-public key pair $(Pub, Priv)$, and receives an encrypted key blob B_{RSA} that contains $Priv$.

2. The application also requests the Keymaster TA for the attestation certificate $Cert$ that verifies that B_{RSA} was generated inside the secure hardware. $Cert$ is signed by asymmetrically using a dedicated private key that is only accessible inside the TrustZone, and its corresponding public key is signed by Google.

3. The application sends $Cert$ to the server. After verifying $Cert$, the server uses Pub to encrypt K, generating $C = Enc_{Pub}(K)$, and sends C to the device.

4. To finish the import process, the application passes C and B_{RSA} to the Keymaster TA. The Keymaster TA decrypts B_{RSA}, and uses $Priv$ to decrypt C and recover K. Then it returns to the application an encrypted key blob B_K that contains K.

While Secure Key Import protects the keys in transit, after importing, they are encrypted inside a key blob as other imported keys. Therefore, an attacker that can recover the key material (as in our attack) can decrypt securely imported keys and break the security of applications that use Secure Key Import.

As before, any key K imported into Galaxy S9 can be extracted using our IV reuse attack on the encrypted blob B_K. However, unlike the regular key import and key generation functionalities, the Secure Key Import API call does not allow us to specify the version of the key blob, making it resilient to our downgrade attack. To be able to break Secure Key Import on the newer S10, S20, and S21, we need to use a different approach: Instead of extracting the key from B_K, we performed our downgrade and IV reuse attacks against B_{RSA} and used the recovered private key $Priv$ to decrypt C and recover the encrypted key in transit, as shown in Fig. 5.

Since B_{RSA} was securely generated in the TrustZone — as a v15 blob — the Keymaster TA function will happily attest to its validity and generate a valid $Cert$ that will allow us to continue with the Secure Import process. When we receive the encrypted key C from the server, we can use $Priv$ to perform the same decryption process as the Keymaster does to extract the imported key K.

We demonstrate this attack in our proof-of-concept on Galaxy S10 and S21 by performing the downgrade attack from Section 4.2 on the wrapping key B_{RSA}. We intercept the request to generate the RSA keys and modify it to wrap the generated RSA keys in blob B_{RSA} with version v15. We then continue to recover $Priv$ with the IV reuse attack.

5.3 Bypassing FIDO2 WebAuthn

FIDO2 WebAuthn [65] is a specification by W3C and FIDO that allows the creation and use of public-key cryptography to register and authenticate to websites instead of passwords. The authentication keys can be generated and used inside an internal secure element called a "platform authenticator" (e.g., TrustZone, Trusted Platform Module (TPM) [34]) or an external secure element called a "roaming authenticator" (e.g., Yubikey [70] and Solo [60]). Such secure elements aim to provide two main security guarantees:

1. An attacker should not be able to extract the keys from the secure element. Meaning that authentication is only possible using the secure element, and it can't be cloned.

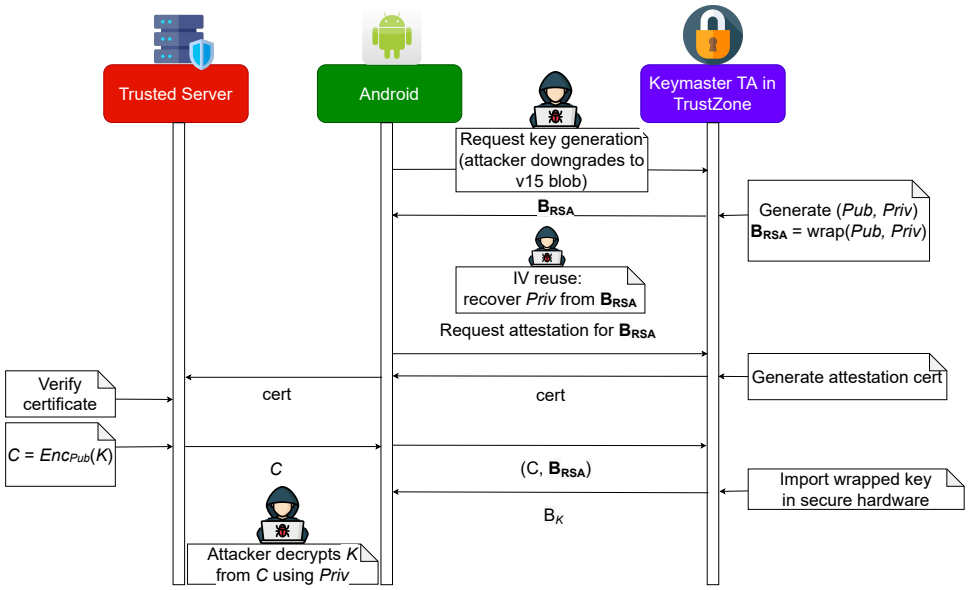


Figure 5: Simplified Secure Key Import: the "hacker" icons indicate the interception points used in our attack [2]

2. Authentication can require user presence. For example, the user can be required to press a button or authenticate using a biometric scan.

Android devices can use the Android hardware-backed Keystore to provide similar security guarantees, using TrustZone instead of a dedicated external hardware secure element.

WebAuthn includes two main stages (see Fig. 6):

1. Registration: the device creates a key pair and sends an attestation to the web server. If the attestation is verified, the server associates the public key with the user.

2. Assertion: when the user tries to login, the server sends a challenge to the device, the device requires user presence and authentication (e.g., a Biometric Prompt) and after receiving user consent the device signs the challenge with the private key. If the server verifies the signature (with the public key) - the user is logged in.

FIDO2 WebAuthn implementations for Android use the Hardware-Backed Android Keystore for key generation and key attestation during registration, and for performing assertions (signing with a key blob) that require user confirmation at login time. When using a Hardware-backed Keystore, WebAuthn is supposed to withstand a compromise of the Normal World. Similarly to what is done in Secure Key Import (Section 5.2), the $(Pub, Priv)$ key pair is generated by the Keymaster inside the TrustZone, and the application receives only an encrypted key blob B_{AUTH} that contains the key pair. The application sends the attestation certificate (including the public key) to the FIDO server. When authentication is required, the application passes the server's challenge and B_{AUTH} to the Keymaster TA, asking it to sign it with *Priv*.

However, we can use our attacks to extract the private key used for authentication and to violate the expected security guarantees. This may allow attackers to clone the "platform authenticator" and allow attestation from other devices without having further access to the target smartphone. Moreover, we can use the recovered private keys to authenticate to a website without the user's presence or consent. Our attack is similar to our previous attack against Secure Key Import (Section 5.2): On Galaxy S9 we use our IV reuse attack to extract *Priv* from the B_{AUTH} key blob; and on Galaxy S10, S20, and S21 we must first perform a downgrade attack. A simplified version of the protocol and our attack is shown in Fig. 6. At a high-level, out attack works as follows:

1. When the device is registered to a website (e.g., Paypal.com), the attacker uses the downgrade attack from Section 4.2. They intercept the request to generate the keys and modify it to force the generated keys in blob B_{AUTH} to use the v15 KDF method.

2. The attacker uses the IV reuse attack to extract the private key material of the key blob.

3. The attacker can now silently authenticate to the website by signing the Assertion challenge with the private key—without user confirmation.

We demonstrate this attack on Samsung Galaxy S10 using StrongKey FIDO sample Android native application and client for FIDO [62]. The StrongKey system has two components: a Linux server that runs the FIDO(R) Certified

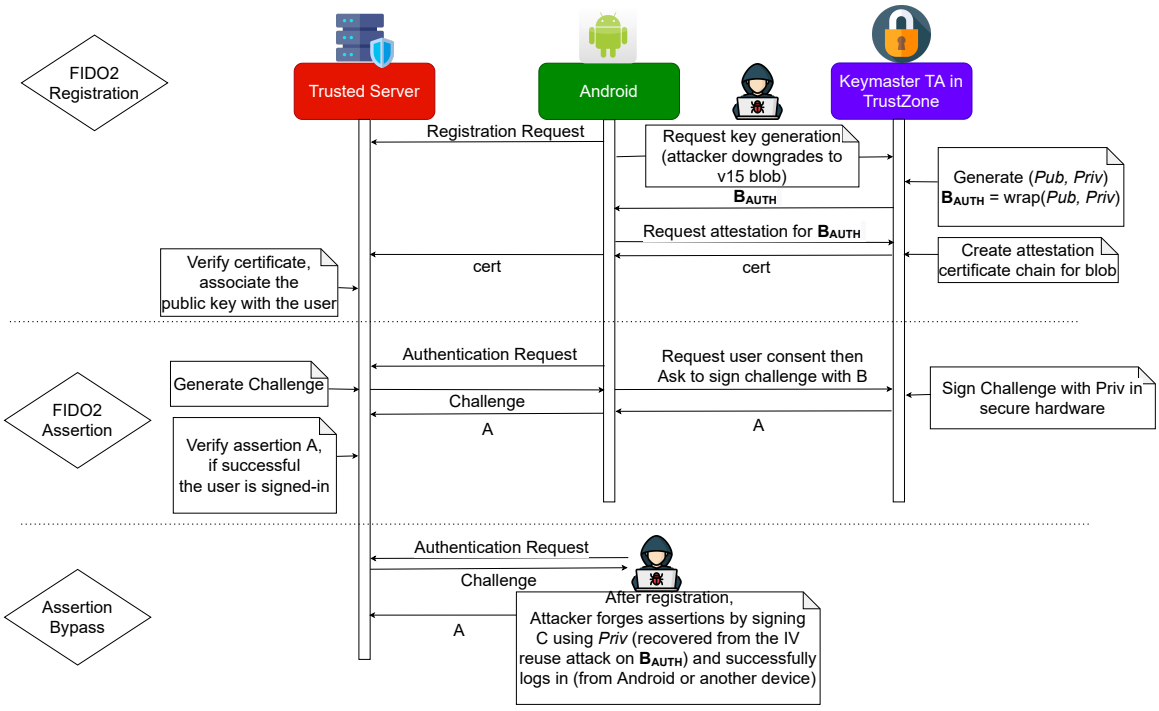


Figure 6: FIDO2 WebAuthn: the "hacker" icons indicate the interception points used in our attack [2]

StrongKey FIDO Server, and the client application, which in our case is the sample Android application. We installed the application on the device without modifications, and used it to register and authenticate against StrongKey's demo server. When the user registers to a website the Android StrongKey application requests to generate a key through the Keymaster HAL. Using the downgrade attack (recall Section 4.2 and the extended version of this paper [59]), the call to the function `nwd_generate_key` is intercepted by our debugger and the generated blob is forced to be a v15 blob B_{AUTH}. We then use the IV Reuse attack to on this blob and recover its private key *Priv*. As in the Secure Key Import, the attestation works seamlessly.

To validate our attack, we verified the recovered private key *Priv* against the public key in the attestation certificate. Using this private key an attacker is able to forge signatures and bypass the Assertion stage. To complete our demo, we created an alternative, modified version of the sample application, which signs the website's challenge using the recovered private key K_P instead of using Keystore—see Fig. 7:

1. Fig. 7a shows how we attach a debugger to the Keymaster HAL process.

2. Fig. 7b shows the GDB output of the downgrade attack.

3. Fig. 7c shows that we are successfully registered against the FIDO server—in the unmodified application.

4. Fig. 7d shows that the attacker successfully authenticates using the alternative application.

5. Fig. 7e and Fig. 7f show an example of re-authentication in the alternative application in order to approve a transaction.

Note that in our demo, we did not make changes to the Android sample application by StrongKey: the interception is done outside the application, and the alternative application (for the assertion during login) required a minimal change to use the recovered key. The registration and authentication was done against StrongKey's own demo server.

6 Discussion

6.1 Low-Level Cryptographic Issues

A fundamental issue in the Keymaster design is the choice of a stream cipher, AES-GCM. The advantages of GCM, being fast and parallelizable, seem less critical for blob encryption—which is always coupled with slow I/O operations; whereas its susceptability to keystream reuse is a cause for concern, as we demonstrated. If the designers choose to retain AES-GCM as a building block then using a nonce-misuse resistant AEAD such as AES-GCM-SIV [35] should prevent IV reuse attacks.

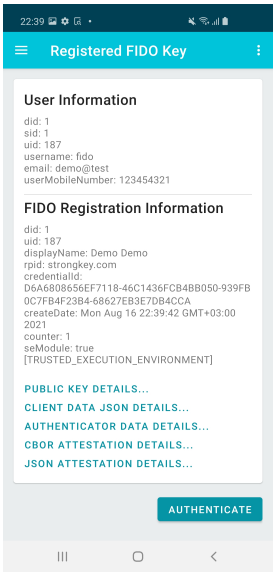
The root cause of the IV reuse attack is that the API offered by Keymaster TA allows the Normal World to *set* the value

```
beyond1:/data/local/tmp # ./gdbserver --attach :1337 $(pidof android.hardware.keymaster@4.0-service)
Attached; pid = 5190
Listening on port 1337
```

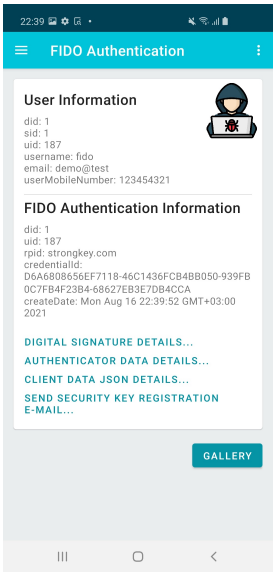
(a) Attaching a GDB debugger to the Keymaster HAL process

```
Breakpoint 2, 0x00000077dae6e514 in nwd_generate_key () from target:/vendor/lib64/libkeymaster_helper_vendor.so
intercepted request to nwd_generate_key
copy old key parameters to new buffer
$1 = 0x7759c24000
$2 = 0x7759c24000
add new parameter (KM_EKEY_BLOB_ENC_VER, 15)
switch to new parameters - this forces the generation of a v15 blob
Breakpoint 4, 0x00000077dae6e544 in nwd_generate_key () from target:/vendor/lib64/libkeymaster_helper_vendor.so
dump the key blob that the keymaster returned
start 0x7759c3b280, end 0x7759c3b4d2, len 252
dumped to result.bin
```

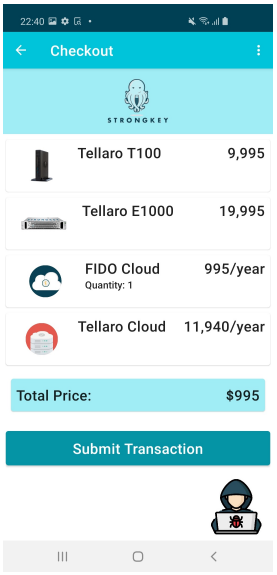
(b) During registration, the GDB script performs the downgrade attack



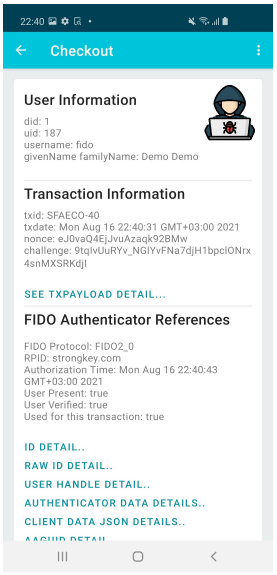
(c) Registration success



(d) Authentication success



(e) Checkout example



(f) Re-authentication success

Figure 7: Screenshots from bypasing of the FIDO2 WebAuthn Sample Application by StrongKey Demo. (7c) shows the successful registration of the legitimate application; (7d)-(7f) show the successful re-authentication of the alternative application.

of the IV. The Keymaster API should not allow the user to set the IV and instead always generate a random 12 byte IV. Modern encryption libraries such as Tink [31] and Google's Trusty Keymaster implementation [32] handle the IV internally without exposing it in the API, thus protecting the user from known pitfalls.

The root cause of the downgrade attack is that the API allows the user to choose the blob version. The user should not be given a choice over any option that might affect the security of the encrypted blobs, especially if the user might be malicious. Moreover, the existence of latent code in a security-critical application such as the Keymaster TA increases the size of the attack surface on the application and should be avoided. As we have shown, exposing such latent code to an external API invites exploitation by attackers.

Furthermore, the persistence of our downgrade to v15 in blobs should not have been allowed. As the process to "upgrade" old encryption blobs is already supported, it should also encrypt the new blobs with the latest (and hopefully safest) encryption option. A properly designed "upgrade" can mitigate many downgrade attacks.

On the positive side, using internal randomness as part of the key derivation process in the Keymaster TA makes the encryption process more robust and actually blocks our attack, despite the fact that the API still allows the IV to be set. The reason the latest Samsung devices were still vulnerable is our ability to downgrade the blob version to a version that does not randomize the key derivation.

6.2 Composability: The Gap in Attestation

In protocols such as FIDO2 WebAuthn and Secure Key Import, a trusted remote server uses key attestation to verify that a key was generated in secure hardware. As we've shown

for Samsung devices, and Busch et al. [16] have shown for Huawei, the TEE implementation can be flawed, which allows attackers to compromise the keys. The protocol step of key attestation is supposed to mitigate such scenarios.

The problem is that the attestation, as defined in the Keymaster HAL, *does not commit to the cryptographic method used to secure the key*. In fact it does not even commit to the version number of the Keymaster TA. This gap means that the remote server that receives the attestation cannot set a policy such as "only accept attestations for keys secured with non-vulnerable KDF versions".

The attestation data that is accessible to the remote server [33, 67] includes general information about the key (`KeyDescription`), whether the key is protected by a TEE or HSM (`SecurityLevel`), key properties (`AuthorizationList`), and information about the device's status (`RootOfTrust` and `VerifiedBootState`) - e.g., if the bootloader is locked. As recent attacks showed [14, 23], the device can be remotely compromised without changing the bootloader state.

The attestation certificate does contain a field called `osPatchLevel`, which could possibly allow a server to identify vulnerable devices. However, as we've shown, Samsung's latest Keymaster simultaneously supports *two* KDF methods: the insecure v15 and the secure v20-s10, and the `osPatchLevel` field does not indicate which method was used. Therefore, relying on the Keymaster TA's software patch level may still leave opportunities for misuse.

The security of protocols such WebAuthn depends on its composition with the implementation and cryptographic design of the Keymaster TA. The current approach of using vendor specific black-box designs makes it impossible to analyze the security of the composition. As we have shown, this provides ample room for vulnerabilities.

6.3 TrustZone-based Keystore Standard

The attacks we described in this paper highlight the critical vulnerabilities that can arise from problems in the cryptographic design of Trustzone-based Keystore. However, so far, these cryptographic designs and protocols have not received much attention in the academic literature. We believe that this is mainly due to the fact that the current ecosystem is based on blackbox designs, with an API that is inconsistent and fragmented between different vendors. Indeed, uncovering the vulnerabilities presented in this paper required a significant amount of time-consuming reverse-engineering effort.

We hope that our work will motivate further research on Keystore security and lead to a uniform open standard for the Keymaster HAL and TA. Such a standard can reduce the current barriers preventing researchers from analyzing the security of the cryptographic designs and protocols. Similar to the standardization process of TLS 1.3 [50], a collaboration between academia and industry will allow for formal analysis of the security of the overall design. This should include a fine-grain threat model that will motivate breakdown resilient designs.

For example, if several key encryption options are available, the attestation certificate should provide details on the encryption method that was used for the key. This will allow servers to block requests using vulnerable encryption methods and mitigate attacks similar to our downgrade attack. Moreover, formal analysis that includes the full API and key encryption schemes could detect issues like the IV reuse vulnerability early in the standardization process.

7 Related Work

Despite their prevalence and importance, there have been very few studies of the cryptographic design of TrustZone instantiations and their composability with higher-level protocols. One exception is the review of the Huawei TrustedCore TZOS by Busch et al. [16]. They have shown that, in fact, it does not provide any hardware protection at all, as it uses *hard-coded fixed keys*. To the best of our knowledge, our work is the first to target and break the cryptographic design of a mature hardware protection instantiation of TrustZone.

There have been several works showing protected keys extraction using side-channels attacks. Lapid and Wool [40] showed that the Kinibi TZOS's AES-GCM implementation is vulnerable to cache timing side-channel attacks, allowing Key Encryption Key (KEK) to be compromised. Keegan [51] showed that the ECDSA implementation in the QSEE's Keymaster TA was vulnerable to a cache timing side-channel attack that can be exploited to leak a hardware-protected EC key. In contrast, as we target the cryptographic design, we were able to extract keys even when side-channels mitigations such as Samsung's Strongbox security processor were implemented.

There have been several previous works analyzing the usage of Keystore-protected keys in applications and higher-level protocols. Sabt et al. [52] showed a forgery attack against the *software-only* Keymaster provided by Google. In comparison, our attacks work against *hardware-backed* Keymaster on the latest Samsung devices. Cooijmans et al. [18], and Breński et al. [15] showed that a privileged attacker could simply use Keystore keys without user consent if the keys are not authentication-bound. In contrast, our attack recovers the full keying material, allowing an attacker to use even authentication-bound keys in unauthorized ways (bypassing authentication/Protected Confirmation/Cloning). Prünster et al. [47] explored the usage of key attestation in the Android Keystore for sensitive operations. Our attacks on FIDO2 WebAuthn and Google's Secure Key Import bypass attestation because the key is indeed generated in secure hardware (and the attacker recovers it).

Software vulnerabilities in TrustZone-based TEEs were studied by many: Pinto and Santo [46] surveyed research on

TrustZone and weaknesses of existing systems, Cerdeira et al. [17] classified different software vulnerabilities in TEEs and analyzed their architectural flaws, and Fleischer et al. [22] evaluated the exploitability of memory corruptions in TEEs. Alendal [1] exploited a stack-based buffer overflow to compromise the secure element of Samsung S20 Exynos devices. Other attacks against Trustonic Kinibi and Qualcomm QSEE include Sang et al. [48], Adamski, Guilbon and Peterlin [49], Beniamini [11, 12], and Machiry et al. [41]. Our work shows cryptographic design flaws that are not implementation flaws and will therefore persist even if a memory-safe programming language is used or if a separate hardware security model is deployed.

When our research began, there were few resources available on the TEEGRIS TZOS. One exception is a blog by Tarasikov [63] which provided useful insight for reverse-engineering TEEGRIS. Later, Menarini et al. [43] published a detailed blog on exploiting TEEGRIS. However, both are focused on software vulnerabilities and not on the cryptographic design.

8 Conclusions

Vendors including Samsung and Qualcomm maintain secrecy around their implementation and design of TZOSs and TAs. As we have shown, there are dangerous pitfalls when dealing with cryptographic systems. The design and implementation details should be well audited and reviewed by independent researchers and should not rely on the difficulty of reverse engineering proprietary systems.

In this work, we examined the cryptographic design and implementation of Android's Hardware-Backed Keystore in Samsung's Galaxy S8, S9, S10, S20, and S21 flagship devices. By an extensive reverse engineering effort, we were able to analyze the Keymaster TA in multiple TZOSs (TEEGRIS, Kinibi, and QSEE). To the best of our knowledge, we are the first to explore the details of the Keymaster TA implementation in TEEGRIS.

Through our analysis we unveiled severe cryptographic design flaws. We identified an IV reuse attack on AES-GCM that allows an attacker to extract hardware-protected key material, and a downgrade attack that makes even the latest Samsung devices vulnerable to the IV reuse attack. We demonstrated a working key extraction attacks on the latest devices. We also showed the implications of our attacks on two higher-level cryptographic protocols between the TrustZone and a remote server: we demonstrated a working FIDO2 WebAuthn login bypass and a compromise of Google's Secure Key Import.

Finally, we note that our attacks on the higher-level cryptographic protocols work on new devices due to subtle attacks arising from their composability with the lower-level key-encryption. Furthermore, we argue that the design choice of using the fragile AES-GCM stream cipher for authenticated blob encryption deserves discussion. These issues further motivate the need for an open and standardized cryptographic design.

Acknowledgement

The authors would like to thank Federico Menarini and Alexander Tarasikov for their interesting insights.

This work was supported by the Robert Bosch Foundation; Len Blavatnik and the Blavatnik Family foundation and Blavatnik ICRC at Tel-Aviv University; The second and third authors are members of CPIIS.

References

[1] Gunnar Alendal. Chip chop - smashing the mobile phone secure chip for fun and digital forensics. BlackHat USA, 2021. URL: https://www.blackhat.com/us-21/briefings/schedule/#chip-chop---smashing-the-mobile-phone-secure-chip-for-fun-and-digital-forensics-23566.

[2] ARM. ARM trusted firmware design. URL: https://chromium.googlesource.com/external/github.com/ARM-software/arm-trusted-firmware/+/v0.4-rc1/docs/firmware-design.md.

[3] ARM. ARM TrustZone. URL: https://developer.arm.com/ip-products/security-ip/trustzone.

[4] ARM. Privilege and exception levels. URL: https://developer.arm.com/documentation/102412/0100/Privilege-and-Exception-levels.

[5] ARM. SMC calling convention (SMCCC). URL: https://developer.arm.com/documentation/den0028/latest.

[6] ARM. Trusted Firmware-A. URL: https://github.com/ARM-software/arm-trusted-firmware.

[7] ARM. The TrustZone hardware architecture. URL: https://developer.arm.com/documentation/100935/0100/The-TrustZone-hardware-architecture-.

[8] ARM. ARM security technology: Building a secure system using TrustZone technology, 2009. URL: https://developer.arm.com/documentation/PRD29-GENC-009492/c.

[9] Brandon Azad. An iOS hacker tries Android, 2020. from Project Zero. URL: https://googleprojectzero.blogspot.com/2020/12/an-ios-hacker-tries-android.html.

[10] James Barclay, Robbie Small, and Taylor Mccaslin. Humans only: Duo mobile and android protected confirmation. Duo blog, 2018. URL: https://duo.com/blog/humans-only-duo-mobile-and-android-protected-confirmation.

[11] Gal Beniamini. Extracting Qualcomm's KeyMaster keys - breaking Android full disk encryption, 2016. Accessed: 2019-11-01. URL: https://bits-please.blogspot.com/2016/06/extracting-qualcomms-keymaster-keys.html.

[12] Gal Beniamini. QSEE privilege escalation vulnerability and exploit (CVE-2015-6639), 2016. Accessed: 2019-11-01. URL: http://bits-please.blogspot.com/2016/05/qsee-privilege-escalation-vulnerability.html.

[13] Bigfoot Biomedical, 2018. URL: https://www.bigfootbiomedical.com/about/press-room/press-releases/google-io-2018.

[14] Mark Brand. In-the-wild series: Android exploits, 2021. from Project Zero. URL: https://googleprojectzero.blogspot.com/2021/01/in-wild-series-android-exploits.html.

[15] Kamil Breński, Krzysztof Pranczk, and Mateusz Fruba. How secure is your Android keystore authentication? F-Secure Labs, 2019. URL: https://labs.f-secure.com/blog/how-secure-is-your-android-keystore-authentication/.

[16] Marcel Busch, Johannes Westphal, and Tilo Mueller. Unearthing the TrustedCore: A critical review on Huawei's trusted execution environment. In *14th USENIX Workshop on Offensive Technologies (WOOT'20)*, 2020.

[17] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted TEE systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1416–1432. IEEE, 2020.

[18] Tim Cooijmans, Joeri de Ruiter, and Erik Poll. Analysis of secure key storage solutions on Android. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, pages 11–20, 2014.

[19] Janis Danisevskis. Android protected confirmation: Taking transaction security to the next level, 2018. URL: https://android-developers.googleblog.com/2018/10/android-protected-confirmation.html.

[20] Dirty cow, 2016. Accessed: 2019-11-01. URL: https://dirtycow.ninja/.

[21] ENKI. Galaxy's meltdown - exploiting SVE-2020-18610, 2020. Accessed: 2021-02-01. URL: https://github.com/vngkv123/articles/blob/main/Galaxy'sMeltdown-ExploitingSVE-2020-18610.md.

[22] Fabian Fleischer, Marcel Busch, and Phillip Kuhrt. Memory corruption attacks within Android TEEs: A case study based on OP-TEE. In *Proceedings of the 15th International Conference on Availability, Reliability and Security*, pages 1–9, 2020.

[23] Guang Gong. An exploit chain to remotely root modern android devices. BlackHat USA, 2020. URL: https://i.blackhat.com/USA-20/Thursday/us-20-Gong-TiYunZong-An-Exploit-Chain-To-Remotely-Root-Modern-Android-Devices.pdf.

[24] Google. Android keystore system. URL: https://developer.android.com/training/articles/keystore.

[25] Google. Android keystore system - import encrypted keys more securely. URL: https://developer.android.com/training/articles/keystore#ImportingEncryptedKeys.

[26] Google. Boringssl. URL: https://boringssl.googlesource.com/boringssl/.

[27] Google. Gatekeeper. URL: https://source.android.com/security/authentication/gatekeeper.

[28] Google. Hardware-backed keystore. URL: https://source.android.com/security/keystore.

[29] Google. Ikeymasterdevice.hal. URL: https://android.googlesource.com/platform/hardware/interfaces/+/master/keymaster/4.0/.

[30] Google. Keymaster functions. URL: https://source.android.com/security/keystore/implementer-ref.

[31] Google. Tink cryptographic library. URL: https://developers.google.com/tink.

[32] Google. generate_nonce. URL: https://android.googlesource.com/platform/system/keymaster/+/master/key_blob_utils/auth_encrypted_key_blob.cpp#40.

[33] Google. Verifying hardware-backed key pairs with key attestation. URL: https://developer.android.com/training/articles/security-key-attestation.

[34] Trusted Computing Group. Trusted platform module (tpm) summary, 2007. URL: https://trustedcomputinggroup.org/wp-content/uploads/Trusted-Platform-Module-Summary_04292008.pdf.

[35] Shay Gueron, Adam Langley, and Yehuda Lindell. AES-GCM-SIV: Specification and analysis. *IACR Cryptol. ePrint Arch.*, 2017:168, 2017.

[36] Lee Harrison, Hayawardh Vijayakumar, Rohan Padhye, Koushik Sen, and Michael Grace. PARTEMU: Enabling dynamic analysis of real-world TrustZone software using emulation. In *29th USENIX Security Symposium (USENIX Security'20)*, pages 789–806, 2020.

[37] Roee Hay and Avi Dayan. Android keystore stack buffer overflow. CVE-2014-3100, 2014. URL: https://securityintelligence.com/android-keystore-stack-buffer-overflow-to-keep-things-simple-buffers-are-always-larger-than-needed/.

[38] Antoine Joux. Authentication failures in NIST version of GCM. *NIST Comment*, page 3, 2006.

[39] Mateusz Jurczyk. Samsung android multiple interactionless rces and other remote access issues in qmage image codec built into skia, 2020. from Project Zero. URL: https://bugs.chromium.org/p/project-zero/issues/detail?id=2002.

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

[41] 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 *NDSS*, 2017.

[42] René Mayrhofer, Jeffrey Vander Stoep, Chad Brubaker, and Nick Kralevich. The Android platform security model. *ACM Transactions on Privacy and Security (TOPS)*, 24(3):1–35, 2021.

[43] Federico Menarini. Samsung investigation part 1: TEEs, TrustZone and TEEGRIS, 2021. Accessed: 2021-02-23. URL: https://www.riscure.com/blog/samsung-investigation-part1.

[44] Gyorgy Miru. [bugtales] a nerve-racking bug collision in samsung's npu driver, 2021. from TASZK. URL: https://labs.taszk.io/articles/post/bug_collision_in_samsungs_npu_driver/.

[45] NSA. Ghidra software reverse engineering framework. URL: https://github.com/NationalSecurityAgency/ghidra.

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

[47] Bernd Prünster, Gerald Palfinger, and Christian Kollmann. Fides: Unleashing the full potential of remote attestation. In *ICETE (2)*, pages 314–321, 2019.

[48] Quarkslab. Reverse engineering Samsung s6 sboot - part i, 2017. Accessed: 2019-11-01. URL: https://blog.quarkslab.com/reverse-engineering-samsung-s6-sboot-part-i.html.

[49] Quarkslab. A deep dive into Samsung's trustzone (part 1), 2019. Accessed: 2019-12-11. URL: https://blog.quarkslab.com/a-deep-dive-into-samsungs-trustzone-part-1.html.

[50] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, IETF, August 2018. URL: http://tools.ietf.org/rfc/rfc8446.txt.

[51] Keegan Ryan. Hardware-backed heist: extracting ECDSA keys from Qualcomm's TrustZone. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 181–194, 2019.

[52] Mohamed Sabt and Jacques Traoré. Breaking into the keystore: A practical forgery attack against Android keystore. In *European Symposium on Research in Computer Security*, pages 531–548. Springer, 2016.

[53] Samsung. KNOX white paper: Root of trust. URL: https://docs.samsungknox.com/admin/whitepaper/kpe/hardware-backed-root-of-trust.htm.

[54] Samsung. Real-time kernel protection (RKP). URL: https://www.samsungknox.com/en/blog/real-time-kernel-protection-rkp.

[55] Samsung. Samsung open source. URL: https://opensource.samsung.com/uploadList.

[56] Samsung. Samsung SCrypto cryptographic module, version 2.0. FIPS 140-2 Non-Proprietary Security Policy v1.3, 2017. URL: https://csrc.nist.gov/CSRC/media/projects/cryptographic-module-validation-program/documents/security-policies/140sp3027.pdf.

[57] Samsung Mobile Security. Personal communications, 2021.

[58] Alon Shakevsky, Eyal Ronen, and Avishai Wool. Keybuster: a keymaster client for samsung devices. URL: https://github.com/shakevsky/keybuster.

[59] Alon Shakevsky, Eyal Ronen, and Avishai Wool. Trust Dies in Darkness: Shedding Light on Samsung's TrustZone Keymaster Design. *IACR Cryptol. ePrint Arch.*, 2022. URL: https://eprint.iacr.org/2022/208.

[60] SoloKeys. The first open-source fido2 security key. URL: https://solokeys.com/.

[61] Gossamer Security Solutions. Samsung Electronics Co., Ltd. Samsung Galaxy devices on Android 10 – spring security target, 2020. URL: https://www.niap-ccevs.org/MMO/Product/st_VID11042-st.pdf.

[62] StrongKey. Strongkey FIDO server (skfs), community edition. URL: https://github.com/StrongKey/fido2.

[63] Alexander Tarasikov. Reverse-engineering Samsung S10 TEEGRIS TrustZone OS, 2019. Accessed: 2019-11-03. URL: https://allsoftwaresucks.blogspot.com/2019/05/reverse-engineering-samsung-exynos-9820.html.

[64] Robert Triggs. Widevine digital rights management explained. Android Authority, 2019. URL: https://www.androidauthority.com/widevine-explained-821935/.

[65] W3C. Web authentication: An API for accessing public key credentials level 2. W3C Recommendation, 2021. URL: https://www.w3.org/TR/webauthn-2/.

[66] Wikipedia. List of best-selling mobile phones. URL: https://en.wikipedia.org/wiki/List_of_best-selling_mobile_phones.

[67] Shawn Willden. Keystore key attestation. Google blog, 2017. URL: https://android-developers.googleblog.com/2017/09/keystore-key-attestation.html.

[68] John Wu. Magisk: The magic mask for Android. URL: https://github.com/topjohnwu/Magisk.

[69] Lilian Young, Shawn Willden, and Frank Salim. New keystore features keep your slice of Android pie a little safer. Google security blog, 2018. URL: https://security.googleblog.com/2018/12/new-keystore-features-keep-your-slice.html.

[70] Yubico. Yubikey: Built for high security. URL: https://www.yubico.com/.

A ARM TrustZone Overview

ARM provides a reference implementation of secure world software called ARM Trusted Firmware [6] (ATF), and the Secure World is usually implemented by a specific vendor (e.g., Qualcomm, Trustonic, Samsung) based on ATF. ATF is responsible for performing Secure Boot, loading the different bootloaders and launching the REE and TEE [2]. It also contains a reference implementation for a Secure Monitor.

To achieve the isolation of the TEE and the REE, TrustZone uses the NS (Non-Secure) bit which is set to 0 if the processor is in Secure state and set to 1 if the processor is in Non-Secure state. The secure state can be switched by executing the SMC opcode (in exception level higher than EL0, e.g., EL1).

The Secure state applies to hardware peripherals and memory, by using the TZASC register (allows to restrict memory to Secure World only) and the TrustZone Protection Controller (TZPC). Menarini et al. [43] shows an example of how the Trusted User Interface (TUI) uses the TZPC to modify

the display and touch controllers as secure and the TZASC configures secure memory for the display. Thus, a user can enter a pin for a payment which will be safe from any Normal World attacker (even if the attacker executes code in the Android OS kernel) and will not be leaked.

The Normal World can only access Non-secure memory, but the Secure World can access Non-Secure memory. The ARM documentation [7] states that Secure and Non-secure cache entries can coexist, and that the Normal World can only get a cache hit on Non-secure cache lines.

The ARMv8-A processor supports 4 exception levels [4]:

1. EL0 - usermode (application in Android, TA in TZOS)
2. EL1 - kernelmode (Android kernel, TZOS kernel)
3. EL2 - hypervisor (used by Samsung to implement RKP [54], which protects the integrity of the Android kernel)
4. EL3 - Secure Monitor

Fig. 1 shows the components in each exception level in the TrustZone architecture.

When the processor is in Secure mode, we can denote S-ELx, e.g., S-EL0 is the secure EL0. Most of our research focuses on S-EL0 (where the Keymaster TA executes), S-EL1 (where the TZOS kernel handles ioctls that the Keymaster TA calls) and EL3 (where the Secure Monitor executes a function handler for a given SMC).

The Secure Monitor provides the interface between the two worlds and performs switching when the SMC (Secure Monitor Call) opcode is executed. Per the ARM SMC Calling Convention [5], "The SMC instruction is used to generate a synchronous exception that is handled by Secure Monitor code running in EL3. The arguments are passed in registers and then used to select which Secure function to execute. These calls may then be passed on to a Trusted OS in S-EL1.". Note that the Secure World also uses SMC for some operations, such as power management or privileged operations that can only be done in the Secure Monitor (EL3).

B The Control Flow in the Keymaster TA

Upon receiving control from an API call (from our client or from the Keymaster HAL), the Keymaster TA has the following flow in `TA_InvokeCommandEntryPoint`:

1. Validates the parameter types for the input and output buffers and makes sure that the memory references that are sent from the Normal World belong to the REE.
2. Parses the input buffer as an ASN.1 structure `indata` and validates it.
3. Calls the appropriate command handler based on `indata->cmd`.
4. Fills the output buffer with ASN.1 structure `outdata`.

There are more than 21 command handlers in the Keymaster TA, including the following, that implement the similarly named API calls as in Section 3.2:

- `swd_generate_key`
- `swd_import_key`
- `swd_import_wrapped_key`
- `swd_get_key_characteristics`
- `swd_export_key`
- `swd_attest_key`
- `swd_begin`/`swd_update`/`swd_finish`

Blob-creating commands accept key parameters that are delivered in the `indata` structure. The parameters control how the key is generated and are also placed inside the blob. They are subsequently used during the cryptographic operations that take the blob as input. Key parameters include:

- Cipher information including:
  - Algorithm (RSA/EC/AES/DES/HMAC)
  - Key size (e.g., 768/1024/2048/3072/4096 for RSA or 128/192/256 for AES)
  - Mode of operation (e.g., ECB/CBC/CTR/GCM)
  - Padding (e.g., none/RSA-OAEP/RSA-PSS)
  - Digest (e.g., none/md5/sha1/sha256)
- The parameters can also include optional access control restrictions on the created blob, including:
  - Purpose (e.g., limit to encryption/signing only, or only encryption and decryption).
  - Maximum number of uses per boot / minimum seconds between operations / expiration date.
  - Require authentication (e.g., by password or biometric prompt) or confirmation by the user.

The main focus of our research is how key blobs are decrypted/encrypted. The blob structure is as follows: The key material is serialized into an ASN.1 structure called `km_key_blob` that contains a version number, key material and key parameters. The ASN.1 structure is then encrypted using AES-256-GCM with an Hardware Derived Key-encryption-key (HDK). This encryption is called "wrapping" and is the topic of much of our work. The "wrapped" key blob is serialized again into another ASN.1 structure called `km_ekey_blob` that contains information that is required for decryption, such as the IV and AAD that was used to encrypt. Fig. 8 shows the process of key wrapping/unwrapping in the Keymaster TA which we describe in this section.

To ensure that key blobs are hardware-protected, the device uses the following keys:

- Root Encryption Key (REK): a 256-bit AES key that is available only in secure hardware and is device-unique.
- Hardware Derived Key (HDK): a 256-bit AES key that is derived from the REK per blob encryption using the Key Derivation Function (KDF) which we discussed in Section 3.4.

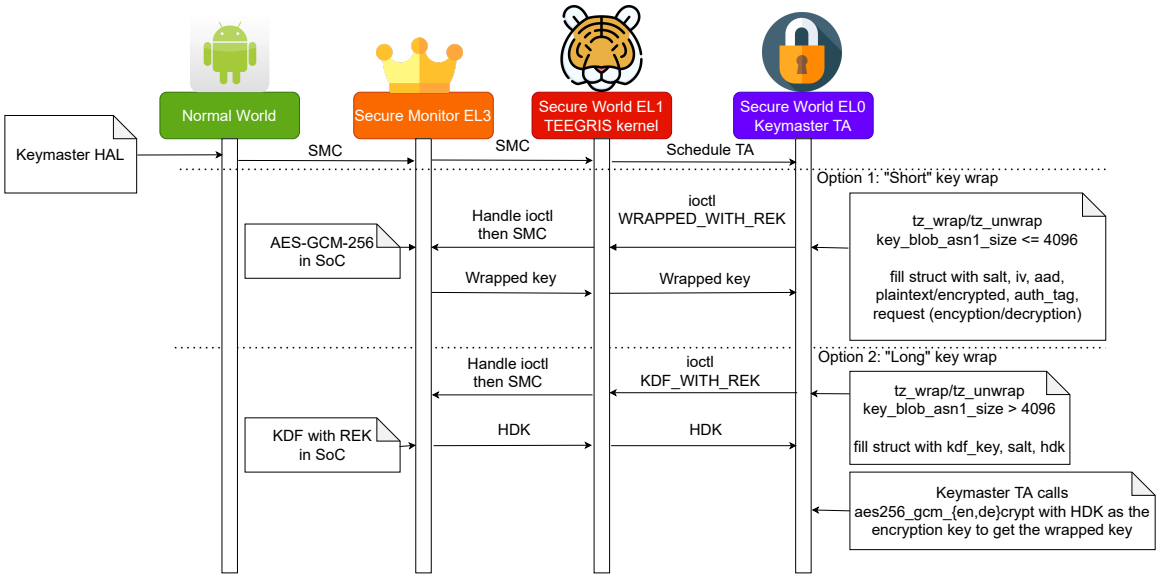


Figure 8: Key wrapping in Keymaster TA in TEEGRIS [2]

At a high level, the AES operation uses the following fields:

- The IV, that is either generated or is located in the parameters that are required for decryption (`KM_TAG_EKEY_BLOB_IV`)
- The AAD that is computed in `swd_get_aad`
- The data to encrypt/decrypt
- The authentication tag for decryption (`KM_TAG_EKEY_BLOB_AUTH_TAG`)
- A salt value that is computed in `swd_get_salt` and is used by KDF to derive the HDK from the REK.

The salt value is computed in the `swd_get_salt` function as the SHA256 digest of a concatenation of values based on the encryption version `ekey_blob->enc_ver`. We refer to values of `enc_ver` symbolically as either "v15", "v20-s9" or "v20-s10" based on the constant strings that are used by the KDF and the device model we observed them on (technically `enc_ver` is a byte value).

The decryption/encryption of ASN.1-serialized key material occurs in the `tz_unwrap`/`tz_wrap` functions (resp.), which call `TEES_WrappedWithREK`/`TEES_DeriveKeyKDF` from `libteesl.so`, which in turn does a ioctl to the crypto driver (`dev://crypto`). See Appendix C for details on how TEEGRIS uses the hardware crypto engine to compute the KDF with REK and AES-GCM operations.

C KDF and Key Wrapping in TEEGRIS

Figure 8 illustrates the two flows that use the salt, IV, AAD, and authentication tag to perform the cryptographic wrapping/unwrapping in TEEGRIS. If the length of the ASN.1-serialized key is at most 4096 bytes, the Keymaster TA calls the `TEES_WrappedWithREK` library function to derive the HDK from the salt and then perform AES-GCM in the crypto engine. Conversely, if the length is greater than 4096 bytes, the Keymaster TA uses the `TEES_DeriveKeyKDF` library function to derive the HDK by calling the crypto driver, and then uses a software implementation of AES-GCM-256 (using the SCrypto library [56] that is based on BoringSSL [26]) to perform the encryption.

In order to understand how the key blobs are encrypted, we reversed engineered TEEGRIS, found the `dev://crypto` driver and analysed its ioctl method. We focus on two specific ioctl commands: `CRYPT_FUNC_WRAPPED_WITH_REK` (that encrypts or decrypts key blobs) and `CRYPT_FUNC_KDF` (that derives a HDK from the REK), that are called from `TEES_WrappedWithREK`/`TEES_DeriveKeyKDF` in the Keymaster TA (resp.).

`CRYPT_FUNC_WRAPPED_WITH_REK` checks that the calling task in TEEGRIS is the Keymaster TA by comparing the current UID to the UID of the Keymaster (10 bytes of null, then "KEYMST") and rejects any other task. It then copies the struct that the Keymaster TA sent to DMA memory, edits the salt by appending the Keymaster TA's own UID (16 bytes) and executes an SMC instruction (passing the physical address of the memory where the struct resides as the third argument). If the SMC returns 0, the modified struct is copied back to the Keymaster TA.

`CRYPT_FUNC_KDF` also calls the same SMC function but with a different arguments (0 as the first argument instead of 1). It computes the SHA-256 digest of the KDF key, the task UID and group and the salt, then passes the address of the struct that contains both the hash and the HDK (with its length). The SMC fills the bytes of the HDK.