RUHR-UNIVERSITÄT BOCHUM

# WireGuardKeyStore vs. WireGuardHSM: A Comparison of Android's Key Storage Solutions

Micha Jonathan Eyl

Bachelor's Thesis – July 11, 2022
Chair for Systems Security

|  |  |
|---|---|
| 1st Supervisor: | Dr. Veelasha Moonsamy |
| 2nd Supervisor: | Prof. Dr.-Ing.Tim Güneysu |
| Advisor: | Prof. Dr. Thomas Östreich |

## Eidesstattliche Erklärung

Ich erkläre, dass ich keine Arbeit in gleicher oder ähnlicher Fassung bereits für eine andere Prüfung an der Ruhr-Universität Bochum oder einer anderen Hochschule eingereicht habe.
Ich versichere, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Die Stellen, die anderen Quellen dem Wortlaut oder dem Sinn nach entnommen sind, habe ich unter Angabe der Quellen kenntlich gemacht. Dies gilt sinngemäß auch für verwendete Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen.
Ich erkläre mich damit einverstanden, dass die digitale Version dieser Arbeit zwecks Plagiatsprüfung verwendet wird.

## Official Declaration

Hereby I declare that I have not submitted this thesis in this or similar form to any other examination at the Ruhr-Universität Bochum or any other institution or university.
I officially ensure that this paper has been written solely on my own. I herewith officially ensure that I have not used any other sources but those stated by me. Any and every parts of the text which constitute quotes in original wording or in its essence have been explicitly referred by me by using official marking and proper quotation. This is also valid for used drafts, pictures and similar formats.
I agree that the digital version will be used to subject the paper to plagiarism examination.
Not this English translation, but only the official version in German is legally binding.

11.07.2022
Datum / Date

[illegible]
Unterschrift / Signature

# Contents

# Abstract

Android developers use *Android KeyStores* to secure the cryptographic keys of their applications. Android KeyStores are implemented by the vendor-specific Keymaster *Trusted Application (TA)*, which is hardware-protected by the *Trusted Execution Environment (TEE)*. Unfortunately, past research has called their security into question by uncovering several vulnerabilities caused by lackluster TEE implementations with proprietary undocumented designs. To free users and developers from this dependency on mobile phone manufacturers, we propose the *SmartCard-HSM* as a possible replacement for the Keymaster TA. The SmartCard-HSM is an external *Hardware-Security Module (HSM)* in microSD card format supported on Android devices.

In this work, we assess this alternative by developing a *hardware-backed WireGuard (HWWireGuard)* implementation, which successfully integrates either device into the package encryption, creating *WireGuardKeyStore* and *WireGuardHSM*. During their performance evaluation, we discovered that the SmartCard-HSM operations incurred a performance overhead compared to similar operations using Android KeyStores.

However, the SmartCard-HSM, while less performant than Android KeyStores, offers several features that can appeal to developers in specific use cases. These cases include developers who want to create opinionated applications regarding their hardware-backed security. Where WireGuard is cryptographically opinionated, WireGuardHSM is also opinionated regarding the actual hardware-backed implementation used to secure the key storage rather than relying on several vendor-specific implementations, as is the case when using Android KeyStores. Furthermore, it offers cross-platform capabilities and sensitive hardware protection that can lead to the device destroying itself and becoming unusable to protect the stored cryptographic key material.

# 1 Introduction

Over the last years, the Android Operating System (OS) has become a popular OS for mobile phones. In 2014, Singh [Sin22] described several security features protecting Android devices. Amongst them were features like application sandboxing, application signing, and user-granted permissions. Unfortunately, evolving attacks combined with a climbing number of security-critical operations have forced developers and smartphone vendors to intensify their efforts to protect relevant data. In order to secure applications and data, a separate execution environment was necessary. One that preferably is entirely hardware isolated from the Android kernel and its application. Over the last years, *Trusted Execution Environments (TEEs)* have fulfilled this role. TEEs allow *Trusted Applications (TAs)* to run inside the so-called *secure world (SW)* separated from the *non-secure world (NW)*. As Pinto and Santos [PS19] described, "[b]oth worlds are completely hardware isolated and granted uneven privileges, with non-secure software prevented from directly accessing secure world resources."

An important TA is the *Keymaster TA*. The Keymaster TA is responsible for *Android KeyStores (AKSs)*, which developers can use to generate, store and use cryptographic keys. As a result, even an attacker with root privileges should not be able to extract these protected keys. Unfortunately, previous vulnerabilities have proven that attackers can penetrate TEEs and extract hardware-protected key material. It is alarming that most major smartphone vendors were affected by some vulnerability or other.

This thesis explores a possible alternative to the Keymaster TA: the *SmartCard-HSM* by CardContact Systems GmbH [Carb]. With the microSD card format, the Hardware-Security Module (HSM) is compatible with most smartphones. Furthermore, it supports Android and provides a Java middleware called *OpenCard Framework (OCF)* [Cara].

In order to properly discuss the alternatives, we compare the performance of both hardware devices. The comparison includes evaluating the performance of the isolated hardware devices and an actual real-life application. For this real-life application, we developed a hardware-backed WireGuard implementation called *HWWireGuard*. HWWireGuard can be used as either *WireGuardKeyStore* using Android KeyStore or *WireGuardHSM* using the SmartCard-HSM.

**Related Work.** Over the last years, the research community has discovered many vulnerabilities in TEEs and their TAs. This problem persists among major smartphone vendors and is not limited to one manufacturer. As such, TEEs have come under fire for their lack of transparency and security analyses.

Busch *et al.* [Mar] demonstrated the inadequacy of Huawei's TEE implementation *TrustCore (TC)*. They unearthed several design and implementation flaws that allowed attackers to extract the *Key Encryption Key (KEK)*. The Keymaster TA uses the KEK to wrap and protect generated keys. Unfortunately, the same KEK was used across several devices and firmware versions and was *not* device-specific! The KEK was hard-coded into the Keymaster TA, exclusively protected by the encryption of its code. Unfortunately, security-by-obscurity design principles were the only ones safeguarding the encrypted TA.

Another TEE implementation is *Qualcomm's Secure Execution Environment (QSEE)* which Beniamini examined in [Gala, Galc, Galb]. By exploiting dormant code in the *widevine TA*, Beniamini could execute code inside QSEE, as seen in [Galc]. In his blog, Beniamini [Galb] explained how the code execution could be abused to extract key material protected by the Keymaster TA. Notably, the vulnerability was not caused by the *TrustZone OS* but rather by a flawed TA running inside the TrustZone OS.

Another TrustZone OS is *TEEGRIS* by Samsung, used on newer Exynos models and examined by Shakevsky *et al.* [SRW14]. Furthermore, by attacking the Keymaster TA, they could extract protected key material from newer models using the so-called *StrongBox Keymaster* functionality. The StrongBox Keymaster uses an *embedded Secure Element (eSE)* with a separate processor, RAM, and storage. The attack was possible due to a downgrade vulnerability exploiting previous implementations. However, as Alendal *et al.* [AAD21] have proven, even without the downgrade attack, the eSE of Samsung's Galaxy S20 was vulnerable to a stack overflow which allowed attackers to execute code arbitrarily. Additionally, Beniamini [Gald] proved the vulnerability of the TEEGRIS predecessor called *Kinibi* by Trustonic.

To analyze these different attacks, Cerdeira *et al.* [CSFP20] wrote a rather extensive analysis of TrustZone-assisted TEEs. They described the comprehensive history of implementation bugs in TAs, which allowed attacks against TEEs. Furthermore, they identified architectural shortcomings in TrustZone OSes by giving compromised TAs too many privileges, as shown by Beniamini [NoA]. Overall, they argued that TrustZone OSes have a shortage of safeguard mechanisms that we have come to expect from OSes, e.g., memory protection.

Compared to TEEs, the research on external HSMs such as the SmartCard-HSM is limited, even though they could offer a solution to some of the problems plaguing TEEs. Lee *et al.* studied using a HSM to secure communication on mobile devices [LKP$^+$11]. They explored a theoretical solution independent from mobile phone manufacturers to not rely on their implementations. In this thesis, we will build upon their work by considering the theoretical aspects and testing them inside HWWireGuard. Lee *et al.* discussed several limiting characteristics caused by the mobile platform. Among the limitations they identified is the necessity of considering the limited resource of a mobile phone, the risk of loss or theft, and user-friendliness. These are all topics this thesis needs to consider when evaluating the practicality of using the SmartCard-HSM. Furthermore, Lee *et al.* proposed an information protection model which strongly informs the HWWireGuard design. Some of the characteristics similarly adopted by HWWireGuard are the generation of a session key with the HSM rather than using it directly for encryption or decryption and the key management approach of symmetric keys between communication peers.

Lastly, this thesis also builds upon the work done on WireGuard by Donenfeld [Jas], specifically the Android implementation of WireGuard. This thesis improves upon this implementation by adding hardware-backed elements into the encryption key generation resulting in HWWireGuard being protected by a physical hardware device and not just the OSes file permissions.

**Goals and Contribution.** As mentioned above, TEEs have a troubled history of lackluster TrustZone OS implementations and security-critical bugs within TAs. This thesis aims to explore the SmartCard-HSM as a possible alternative to the Keymaster TA running inside the TEE. In order to be a viable replacement, the HSM needs to meet several goals:

1. The *integration into a real-world example* like HWWireGuard needs to be achievable.

2. The *expected performance overhead* incurred by the HSM must conform to a mobile platform's resource restrictions.

3. The thesis needs to *identify security-critical advantages* the HSM can offer compared to TEEs.

**Organization.** After introducing the subject of this thesis in Chapter 1, Chapter 2 will give the required technical background around Android KeyStores, the HSM, and WireGuard. Afterward, Chapter 3 will examine how HWWireGuard works and is structured, while Chapter 4 presents the actual implementation. Then, Chapter 5 evaluates the performance of both hardware-backed solutions inside and outside of HWWireGuard, and Chapter 6 discusses the security-critical considerations brought forth by the different solutions. Finally, Chapter 7 presents the conclusion that can be drawn from the results of this thesis.

# 2 Technical Background

This chapter provides the necessary context on the two hardware-backed key storage devices contrasted in this thesis. This context includes information on their general structure, function, and usage. Furthermore, this chapter introduces WireGuard, which is relevant to understanding HWWireGuard later in the thesis.

## 2.1 Android KeyStores

The *Android KeyStore System (AKS)* is the standard solution for key storage on Android OSes. It allows developers to store cryptographic keys in hardware-protected containers. In order to achieve hardware protection, many mobile phone manufacturers use the Arm TrustZone TEE architecture. The following will explain how the TrustZone TEE works and how the Keymaster TA stores cryptographic keys with it.

**Arm TrustZone TEE.** As already examined in Chapter 1, different vendors have differing TEE implementations because the TrustZone architecture is only a framework for SoC designers [Tru]. Every vendor then uses this architecture to implement their actual security solutions. As such, the implementation can vary from vendor to vendor.

Nonetheless, the overall architecture and structure of the TrustZone TEE are the same across vendors. The idea is to create a separate execution environment for security-critical applications, called *Trusted Applications (TAs)* or *trustlets*. The environment for TAs is called *secure world (SW)* or *Secure Execution Environment (SEE)*, while the actual Android OS is restricted to the *non-secure world (NW)* or *Rich Execution Environment (REE)*. Figure 2.1 illustrates this principle. Both worlds run on the same processor and, as such, have its full power at their disposal. They are separated by hardware mechanisms preventing unauthorized access. Responsible for switching between the environments is the *Secure Monitor* by receiving either *Secure Monitor Calls (SMCs)* or a subset of hardware exception mechanisms [Tru]. Data flow is only possible on the same *Execution Level (EL)*, while control flow happens between adjacent ELs. The Keymaster HAL, Keymaster TA, and the TrustZone OS Kernel, shown in Figure 2.1, are all software components implemented by the vendor. The goal of this separation is to be strong enough to secure the TAs in the SW even with a compromised NW.

**Keymaster TA.** Because of the hardware-protected nature, AKS needs to be able to interact with hardware components specifically designed for security-critical operations. In order to facilitate this feature, the Android documentation defines the functions the *Keymaster Hardware Abstraction Layer (HAL)* needs to offer. The smartphone vendors then implement the Keymaster HAL in the NW and the corresponding Keymaster TA in the SW. By communicating over the predefined Keymaster HAL functions, developers can now interact with the Keymater TA. Figure 2.2 shows how a cryptographic key can be generated and used with the Keymaster TA. At first, the user app requests the generation of a key. This key is then wrapped into a so-called *keyblob* and stored in the NW's file system. Different Keymaster TA implementations use different methods of wrapping the keyblob and securing the *Key Encryption Key (KEK)*. Next, this keyblob is used to perform a cryptographic operation like an encryption. During encryption, the keyblob and message are sent to the Keymaster, where the keyblob is unwrapped, and the resulting key is used.

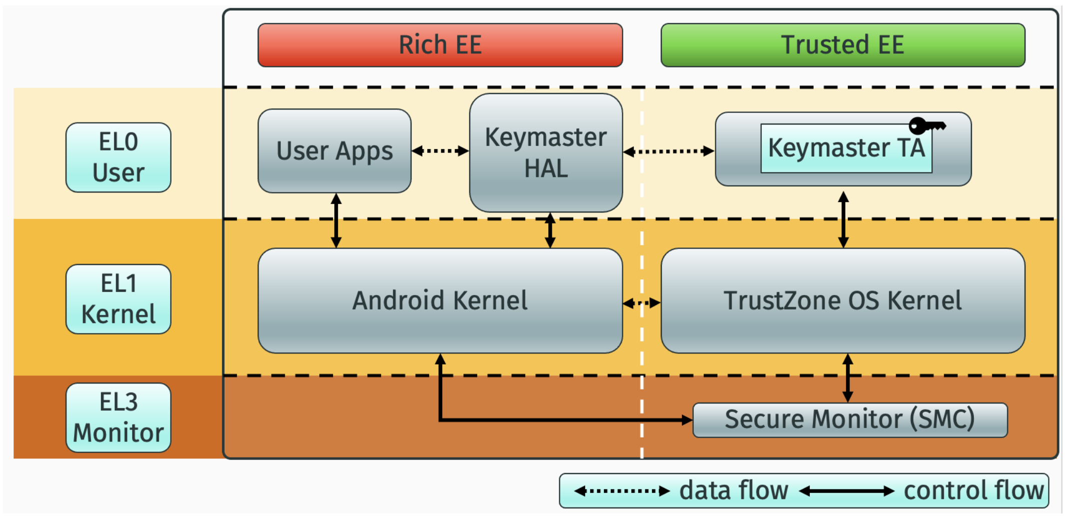


Figure 2.1: Structure of TEE with the example of Keymaster TA [Anda].

**Embedded Secure Elements (eSE).** Another term that should not be omitted when talking about hardware-backed security on mobile phones are eSEs. As mentioned in Chapter 1, newer Android smartphones support the so-called *StrongBox Keymaster* feature, which uses an eSE to secure cryptographic material like the KEK. In the Android Developer documentation [Str], the StrongBox is described as a HSM-like secure element which is either an embedded Secure Elements (eSE) or an *on-SoC secure processing unit (iSE)*. In this thesis, we will not examine these devices more closely, but this is something that could be considered for future work.

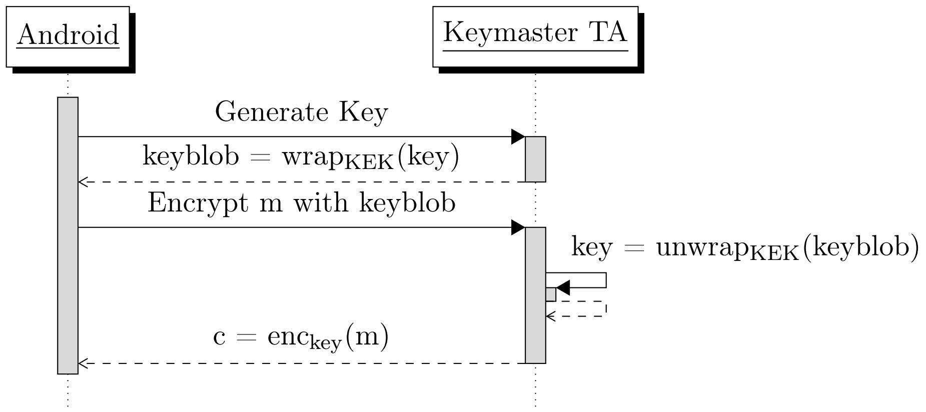

Android
Keymaster TA
Generate Key
keyblob = $\text{wrap}_{\text{KEK}}$(key)
Encrypt m with keyblob
key = $\text{unwrap}_{\text{KEK}}$(keyblob)
c = $\text{enc}_{\text{key}}$(m)

Figure 2.2: Cryptographic key use with the Keymaster TA [SRW14].

## 2.2 SmartCard-HSM

The SmartCard-HSM is a *Hardware-Security Module (HSM)* that we want to use instead of the Keymaster TA. HSMs are specialized hardware devices able to perform different cryptographic operations. They enjoy great popularity, especially in areas with heightened security requirements. As such, institutions like banks but also governmental organizations have come to rely on them. The following will explain what a HSM is more generally and what features the SmartCard-HSM offers.

**Hardware-Security Module (HSM).** Wong [Dava, Davb] described different hardware solutions for highly-adversarial environments. Next to SEs, he also mentioned HSMs and described them as *bigger SEs*. Wong categorized HSMs as possessing better performance qualities and being more portable. As such, HSMs come in many forms, from big devices with their own shelf on a server rack over USB sticks to microSD cards. They can perform more or less computationally intensive tasks depending on their size. Most HSMs are accessible via the *Public-Key Cryptography Standard 11 (PKCS#11)*, which offers the standard operations to generate keys, manage them on the device, and use them in cryptographic operations.

A HSM's foremost duty is to protect its stored key material. According to Smirnoff [Pet], this is achieved with the help of several characteristics. Firstly, HSMs possess specialized hardware that prevents side-channel attacks and memory extraction. HSMs often offer features that allow the destruction of the module in case of physical attacks or an overrun of failed password entries. Secondly, HSMs run security-focused OSes. Thirdly, they only respond to a limited number of commands, which reduces the risk of vulnerable commands. Fourthly, they actively hide and protect the key material. Unfortunately, HSMs have not escaped some catastrophic software bugs. Wong attributes this to the fact that HSMs often only focus on the hardware protection side and neglect software implementations.

**SmartCard-HSM.** As already mentioned, the SmartCard-HSM is a HSM developed by CardContact Systems GmbH [Carb]. CardContact Systems offers it in different formats, from USB tokens to SIM cards. For this thesis, we use the SmartCard-HSM in microSD card format. The SmartCard-HSM can generate AES, RSA, and ECC keys and uses them in signature, encryption, and decryption operations. The HSM is controlled through so-call *Application Protocol Data Units (APDUs)*, as seen in Table 2.1. The first two bytes identify the class of the command and the actual instruction. The following bytes classify the parameters, such as the key ID or the algorithm ID. Afterward follows the length of the command data, the actual command data, and the expected length of the response data. The response to the APDU command is either the result of the operation, e.g., an encrypted message, or a status word which can also be an error message.

With these commands, all operations that the HSM offers could be performed, but because they are time-consuming to create, CardContact Systems offers the *OpenCard Framework (OCF)* [Cara] with functions such as the *deriveSymmetricKey()*. In this thesis, we will use the OCF for all operations necessary on the HSM.

On the SmartCard-HSM, keys are stored in so-called *slots*. These slots are identified by a number which we also use to select the key in operations. When generating a new key, the user can choose which algorithms are allowed to be used in combination with this key. Furthermore, the SmartCard-HSM offers the use of a *Device Key Encryption Key (DKEK)*. This feature allows us to perform the vitally important transfer of keys between devices which will be necessary for this thesis. The DKEK ensures that exported keys are protected and only can be imported into a SmartCard-HSM with the same DKEK share.

## 2.3 WireGuard

WireGuard is an application providing a fast and secure VPN tunnel. Compared to other VPN providers such as OpenVPN, WireGuard possesses several unique qualifiers. Firstly, as Donenfeld describes, "WireGuard is cryptographically opinionated" [Jas]. This means that WireGuard lacks so-called cipher agility. For instance, it uses the *BLAKE2* algorithm for hashing and does not allow users to select from a number of different options as in other handshake protocols. Secondly, WireGuard is also opinionated regarding the underlying transport protocol. Unlike OpenVPN, WireGuard only supports the *User Datagram Protocol (UDP)*. Thirdly, WireGuard also possesses a 1-RTT handshake for efficient key negotiations, further improving performance. Fourthly, WireGuard has a relatively small code base of about 4,000 lines of code. This feature enables us to easily modify the code, as will be necessary for this thesis.

**WireGuard Protocol.** Figure 2.3 shows a standard WireGuard handshake. Usually, WireGuard does not establish a connection unless there is a need to send some packages. If a peer wants to send data, they establish a handshake as the initiator. This handshake contains data like a type identifier, an ephemeral public key, and message authentication codes (MACs). Similar data is also within the handshake

| Field name | Value | Description |
|---|---|---|
| **CLA** | '50' / '5C' | Instruction class |
| **INS** | '48' | DERIVE SYMMETRIC KEY |
| **P1** | Var | Key identifier for symmetric key |
| **P2** | Var | Algorithm Identifier<br>'59' - AES CBC ENCRYPT<br>'60' - AES CBC DECRYPT<br>'61' - AES CMAC |
| **LC** | Var | Number of bytes of command data |
| **C-DATA** | Var | Derivation parameter |
| **Le** | '00' | Expected number of bytes of response |
| **R-Data** | Var | Derived key value |
| **SW1/2** | '10 00' | Normal processing |
| | '64 00' | Wrong length |
| | '66 00' | Security status not satisfied |
| | '69 00' | Incorrect data in the command field |

Table 2.1: Altered APDU command for AES operations from SmartCard-HSM Manuel [Sch21].

response. The MACs ensure the authentication of the peers, and the ephemeral keys provide a unique cryptographic primitive to every new handshake. From the data in the handshake messages and a possible additional preshared key, WireGuard then derives encryption keys that are used to encrypt the transport data. The key generation process is described in more detail in [Jas].

If a peer wants to keep the connection to another peer without sending data, they can activate the *keep-alive* mode by assigning the keep-alive variable in the configuration file a number $X$. Then, the peer sends so-called keep-alive messages every $X$ seconds. The keep-alive message is a standard message that ensures that both peers stay connected and perform new handshakes every two minutes.

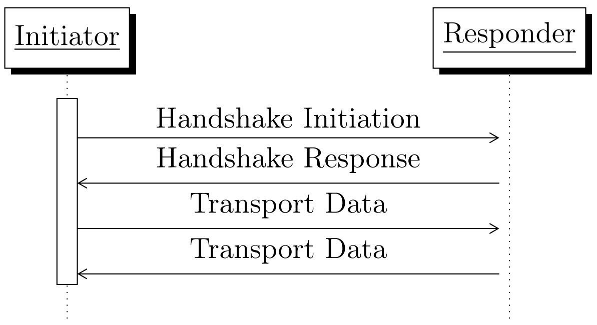


Figure 2.3: WireGuard handshake.

**WireGuard Configuration.** All parameters to define a connection between WireGuard peers are assigned within the WireGuard configuration file. In Linux, this is an actual file in */etc/wireguard*. Android's WireGuard version, on the other hand, offers an interactive interface to enter these values. An example of a WireGuard configuration file is shown in Figure 2.1.

The file defines the interface within the tag *[Interface]*. Here the static private key is defined alongside the IP address the peer has within the VPN network. Furthermore, the physical port is defined on which WireGuard runs. Each configuration file can have multiple peers described under the *[Peer]* tag. The necessary data for a peer includes the static public key, the IP address range the peer is allowed to have, and the actual endpoint IP address and port. Two optional features that WireGuard offers are the *persistent keepalive* and *preshared key (PSK)* features. As explained before, persistent keepalive means that the peer sends a keep-alive message, in this case, every 15 seconds. This option allows us to establish a connection without actual data being exchanged. In this thesis, we will use this feature to establish connections between peers without needing to send data explicitly. The PSK is critical because it allows us to integrate other security features like a hardware-backed cryptographic operation into the WireGuard protocol without the need to change it.

```
[Interface]
PrivateKey = cCOQQtFONv3OY4GBivclWXN67cK7OM6X67CWFGIekXY=
Address = 10.0.0.1
ListenPort = 60766

[Peer]
PublicKey = ygO2CLJvcjzmELDWtlcvuBZQaCmzsl+ElKyCjKTstwI=
AllowedIPs = 10.0.0.2
Endpoint = 192.168.1.78:60766
PersistentKeepalive = 15
PresharedKey = EvL1/FfX6Oy4xsaY9CXkAk2nqggwRVzyL4uvOpXnDXk=
```

Listing 2.1: WireGuard Configuration File.

**WireGuard on Android.** The WireGuard implementation on Android is the application we aim to modify in this thesis. As such, a quick overview of its structure is prudent. WireGuard on Android is made up of three essential parts. The first one is the *user interface (UI)* implemented inside its own package and written in the programming language *Kotlin*. This package contains all the activities and fragments necessary for the WireGuard UI to run. It handles the user input, saves the configurations, and provides users with settings they can alter. The second part is the so-called *tunnel* package written in Java. Herein lie all classes necessary for the backend, such as the data constructs Interface and Peer. Furthermore, it contains the class *Backend*, which is necessary to interact with the WireGuardGo backend implementation. The WireGuardGo implementation is the third part and is controlled by the Backend class, which offers a *Java Native Interface (JNI)* to control WireGuardGo. These controls include starting and stopping tunnels, getting statistics from WireGuardGo, like the number of attempted handshakes, and setting new configuration files. The WireGuardGo backend is the actual WireGuard implementation.

# 3 Designing a Hardware-Backed WireGuard on Android

This chapter illustrates the building and structuring of a hardware-backed WireGuard (HWWireGuard) implementation. Section 3.1 explores the reasons why WireGuard is uniquely qualified for the integration of hardware-backed security devices. Afterward, Section 3.2 describes the user interface of HWWireGuard, and Section 3.3 presents the actual design of the HWWireGuard backend with its intermediate development versions. By structuring the project into these development versions, the amount of additional innovations per version is reduced, and each version can build upon the previous one.

HWWireGurd will then allow a proper performance evaluation in Chapter 5 with a real-world application and not just the isolated operations on the hardware devices.

## 3.1 Why WireGuard?

Before describing HWWireGuard, it is crucial to reiterate the reasons for choosing WireGuard as a real-world example. Next to WireGuard other apps like *FreeOTP*[1], *Briar*[2], and *Element*[3] were also considered. These are different kinds of applications like a two-factor authentication application or a peer-to-peer messenger. Compared to these apps, WireGuard possesses several qualities making integration easier. The following describes these qualities in more detail.

1. WireGuard is an **open-source project**: An essential quality of WireGuard for this project is the accessibility and modifiability of the application. The open-source nature allows the necessary changes to the WireGuard code to integrate the hardware devices.
2. WireGuard has a **small code base**: As discussed in Chapter 2, WireGuard was created to keep the code base as small as possible. As a result, it only has about 4,000 lines of code, and any modifications necessary to the code are easier

[1]`https://github.com/freeotp/freeotp-android`, as of July 11, 2022
[2]`https://github.com/briar/briar`, as of July 11, 2022
[3]`https://github.com/vector-im/element-android`, as of July 11, 2022

to identify and implement. For instance, the apps *Briar* and *Element* are far bigger projects than WireGuard's Android implementation.

3. WireGuard's key storage is **not protected by a hardware device**: WireGuard on Android saves all interface and peer configurations on files. Only the OS file permissions protect these files, including the private key of the interface and the preshared key. This safeguard is sufficient against most threats, but against a rooted attacker, this is not enough. Security measures like HSMs or TEEs protect against a fully compromised device. Ideally, a rooted attacker is not able to breach such devices. For this reason, integrating them inside the encryption key generation makes an attack on the established tunnel more difficult. The attacker needs authenticated access to the hardware device protecting the connection.

4. WireGuard offers an **easy integration**: While other apps meet the above expectations, the integration aims to preserve the cryptographic backend. WireGuard makes this easy because of the preshared key feature. As a result, most security considerations of WireGuard translate directly to HWWireGuard. A modification of *FreeOTP* would require an alteration of the actual cryptographic backend, which is against the goal of minimizing the changes to the underlying protocol of the application.

5. WireGuard is a **peer-to-peer implementation**: A peer-to-peer application is easier to modify because the developer only needs to change and understand one side. In a server-client model, the workload could double because any modifications to cryptographic components would need to be mirrored on the other side.

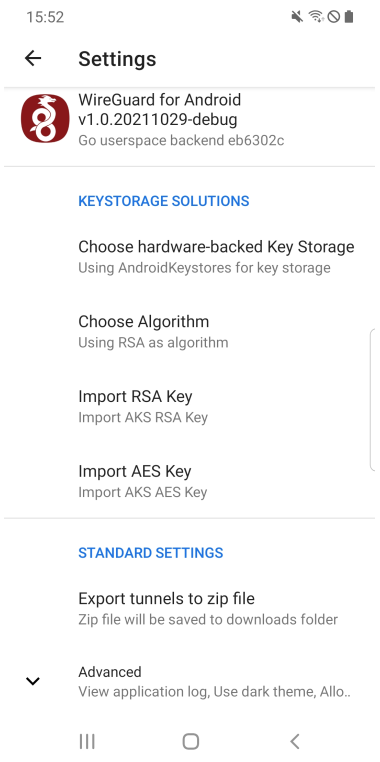
## 3.2 User Interface (UI)

The user interface is the part of the app with which the user interacts. The goal is to keep the changes to WireGuard UI as minimal as possible, which is also the goal for the whole project. Nonetheless, HWWireGuard needs to add several options for the user to utilize the app effectively. These changes include a selection of the hardware device the user wants to select or the user's authentication on these devices. The following sections present the necessary design changes to the UI of HWWireGuard.

### 3.2.1 Selecting the Hardware Device

One decision taken reasonably early in the development process was to make a WireGuard app that allows the user to select between three options. These options include no hardware-backed device at all, the SmartCard-HSM or Android KeyStores. This selection opportunity allows the user to decide which manufacturer should be trusted. The user can select whether to run HWWireGuard as the standard WireGuard, WireGuardHSM, or WireGuardKeyStore.

A suitable place for this selection is in the settings of WireGuard. To visually separate the HWWireGuard-specific settings from the standard WireGuard settings, HWWireGuard can use Android's Preference Categories to isolate the preferences. The actual implementation of this will be explained in Chapter 4. All additional settings for HWWireGuard are grouped within a preference category with the heading KEYSTORE SOLUTIONS, as shown in Figure 3.1. The additional tabs within the preference category like *"Import RSA Key"* or *"Chose Algorithm"* are relevant for Section 3.2.2. Figure 3.2 shows the different hardware devices a user can select.



15:52

Settings

WireGuard for Android
v1.0.20211029-debug
Go userspace backend eb6302c

KEYSTORAGE SOLUTIONS

Choose hardware-backed Key Storage
Using AndroidKeystores for key storage

Choose Algorithm
Using RSA as algorithm

Import RSA Key
Import AKS RSA Key

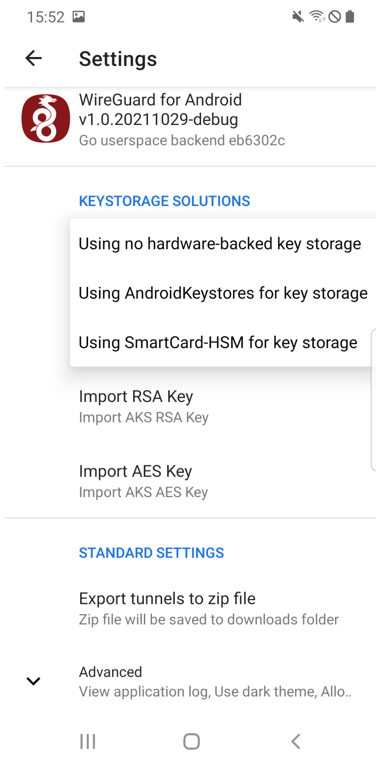
Import AES Key
Import AKS AES Key

STANDARD SETTINGS

Export tunnels to zip file
Zip file will be saved to downloads folder

Advanced
View application log, Use dark theme, Allo..

Figure 3.1: Preference Category.



15:52

Settings

WireGuard for Android
v1.0.20211029-debug
Go userspace backend eb6302c

KEYSTORAGE SOLUTIONS

Using no hardware-backed key storage

Using AndroidKeystores for key storage

Using SmartCard-HSM for key storage

Import RSA Key
Import AKS RSA Key

Import AES Key
Import AKS AES Key

STANDARD SETTINGS

Export tunnels to zip file
Zip file will be saved to downloads folder

Advanced
View application log, Use dark theme, Allo..

Figure 3.2: Hardware Device Selection.

Figure 3.3: HWWireGuard Preference Activity.

### 3.2.2 Importing and Selecting Keys

Another critical topic is the importation of keys for the hardware devices and how HWWireGuard identifies the key it is supposed to use for the operations. HWWireGuard must be able to perform hardware-backed cryptographic operations, which require a key on the hardware device and correct identification of that key. Furthermore, peer devices must have the same key on their hardware device.

The answer for importing keys depends on whether the SmartCard-HSM or Android KeyStores are used. Because this feature is unimportant to the overall comparison of the hardware devices, the most straightforward and practical solution was selected.

Furthermore, HWWireGuard needs to perform some key-management tasks and should at least be able to select between RSA and AES keys. In the appropriate paragraph, we describe how HWWireGuard handles the key selection and additionally give an outlook on what a more sophisticated key management could be.

**SmartCard-HSM.** The SmartCard-HSM is mainly controlled via the *Smart Card Shell 3 (SCS3)* offered by CardContact System GmbH. The SCS3 allows developers to generate, export, and import keys across several SmartCard-HSMs on Windows and Linux computers. HWWireGuard has specific slot numbers for the RSA and AES keys hard-coded and expects the user to ensure that these slots contain the correct key types. Therefore, HWWireGuard expects the user to generate the correct keys inside the correct slots.

Doing this allows more time in the development process to focus on the security-critical implementations relevant to this thesis. The SCS3 also allows the export and import of keys which is necessary to ensure that the HSMs for both peers possess the same key. The exchange is protected by a *Device Key Encryption Key (DKEK)*, which secures the key transfer from one device to the other. As a result, the key is never transferred unprotected, contrary to how the Android KeyStores are transported.

**Android KeyStore.** Unfortunately, the same solution does not work for Android KeyStores. As a result, HWWireGuard must offer the possibility to import RSA and AES keys. Figure 3.1 shows the tabs *"Import RSA Key"* and *"Import AES Key"* a user can employ to import new keys. The device's *Download* folder is where both functions expect the necessary key files. AES requires a *key.txt* file with the AES key encoded in *Base64*. RSA needs a *crt.pem* file with a PEM- or DER-encoded certificate and a *private_key.der* file with a PKCS#8 DER encoded private key. AES key generators can quickly generate the AES key but generating the correct format for the RSA key files is more complicated. Listing 3.1 illustrates how *OpenSSL* on Linux can generate the files needed. Unlike the HSM, the implemented import function for the Android KeyStore keys does not support protection like the DKEK. Although the Android documentation for Android KeyStores [Andb] mentions a possibility to import encrypted keys more securely, we could not implement this in HWWireGuard because one of the test devices used did not have the required Android version 9.

```
openssl genrsa -out rsakeypair.pem 2048
openssl pkcs8 -topk8 -inform PEM -outform DER -in rsakeypair.pem -out
    private_key.der -nocrypt
openssl req -new -x509 -key rsakeypair.pem -out crt.pem -outform pem
```

Listing 3.1: Terminal Commands to create RSA key files.

**Key Management.** For the key selection, HWWireGuard provides a drop-down menu similar to the hardware device selection, as shown in Figure 3.1. The user can either select the RSA or AES key.

In this thesis, only one key is available for HWWireGuard to use for all tunnels and peers. It would be far more interesting to have multiple keys on the hardware devices that can be selected for specific tunnels. A business using HWWireGuard could define multiple security domains corresponding to these keys. Another design for utilizing multiple keys could be exclusively using one key for one connection between peers. Unfortunately, the 60,000-byte *EEPROM* [Sma] storage space on the SmartCard-HSM limits the number of saved keys. Android KeyStore does not have the same restrictions because the memory used to store the keyblobs is the mobile phone memory.

The feature of multiple security domains would make further changes to the UI necessary, where users can select the key used for a specific tunnel. This implementation would allow the tunnel to operate in the security domain of its base key. As a result, administrators could define security domains and give certain employees access to them by loading the corresponding key onto the hardware device.

### 3.2.3 User Authentication

The last design alteration to the HWWireGuard UI are prompts for user authentication. Different prompts have to be used depending on the hardware device selected, as shown in Figure 3.6.

**SmartCard-HSM.** To access the SmartCard-HSM, the user must use a personal identification number (PIN) previously set during the initialization with the SCS3. The PIN needs to be entered once at the start of the tunnel. HWWireGuard provides a dialog prompt with a field for password entry. The characters entered get hidden by dots, as standard in Android password prompts. Figure 3.4 illustrates where the user can enter the PIN of the SmartCard-HSM.

**Android KeyStore.** The authentication for Android KeyStores is possible via Android built-in biometric prompts. The user can either authenticate using the biometric authentication of the mobile phone, like a fingerprint, or use the system PIN. Figure 3.5 shows the biometric dialog prompt.

## 3.3 HWWireGuard Backend

There are many different ways in which one can integrate hardware-backed cryptography into WireGuard. In this thesis, we will examine three such possibilities, each one building on the features of the previous version and addressing further security-critical considerations not taken into account in the previous ones. Unlike the hardware device selection, the user cannot select which backend to use from inside the app. Instead, there are three different Github branches with the corresponding backend.

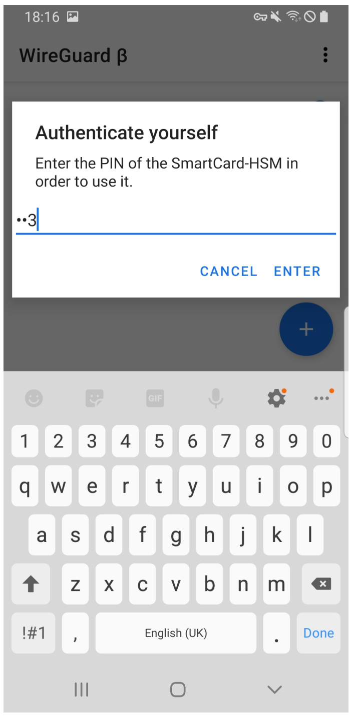


Figure 3.4: SmartCard-HSM.

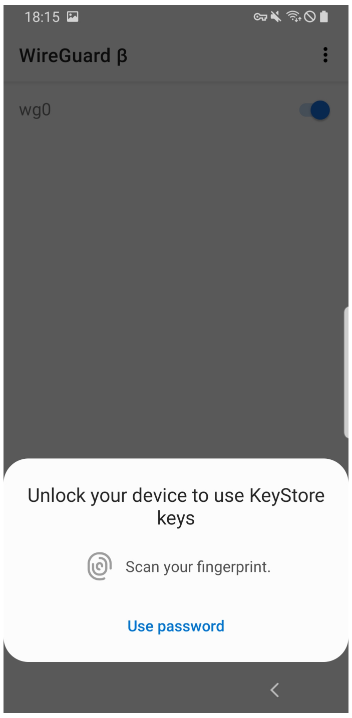


Figure 3.5: Android KeyStore.

Figure 3.6: HWWireGuard User Authentication Dialog.

### 3.3.1 Version 1: HWWireGuard with Static PSK

The first version of HWWireGuard is also the most fundamental and straightforward to design. The idea is to use either the SmartCard-HSM or Android KeyStores to generate a PSK. By doing this, the PSK depends on a secret from a hardware device, and an attacker can only calculate it by either extracting the key or stealing the devices and forging the authentication. The calculated PSK needs to be the same for both communication peers.

Figure 3.7 shows the structure of HWWireGuard and illustrates how to solve the previous problem. On the left-hand side is *HWWireGuard*, the usual WireGuard process with a few additional functions such as *startMonitor()*. This function starts a parallel process called *HWMonitor*, mainly needed to manage the PSKs. The final element is the *Hardware Device* able to perform the hardware-backed cryptographic operations. The key used for this is imported and selected, as explained in the previous section.

When the user starts a tunnel in HWWireGuard, it calls the *startMonitor()* function in (1.1), and starts HWMonitor. The monitor now ensures that the PSK updates with every changing timestamp. Timestamps have the format *"YYYY:mm:DD:hh"* and are therefore accurate to the hour. If the timestamp changes, it is used as input for *hwOp* in (1.2). The hwOp function will be explained more closely in Chapter 4. In (1.3), the newly generated PSK is loaded into HWWireGuard and, more specifically, the WireGuardGo backend. All handshakes happening from now on will use this PSK, until the timestamp changes. The function *stopMonitor* stops HWMonitor.

Within the same hour, the timestamp and, as such, the PSK stays the same and is static. An improvement would be reducing the intervals between changing PSKs further. Otherwise, an attacker with access to the device can crack the encryption by stealing the PSK used for a reasonably long time. To minimize this risk, we developed HWWireGuard Version 2.

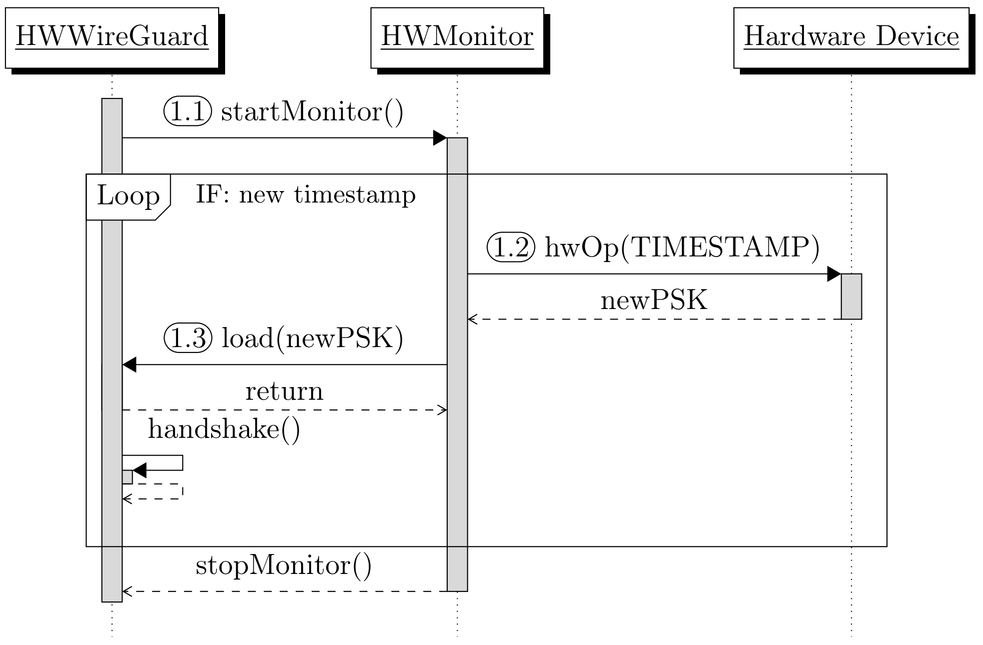


Figure 3.7: HWWireGuard Version 1.

## 3.3.2 Version 2: HWWireGuard with Dynamic PSK

Version 2 improves by changing the PSK with every successful handshake. In practice, this means that the PSK changes every two minutes. As shown in Figure 3.8, the start and the actions for a changing timestamp are identical to Version 1. However, after the first handshake, HWWireGuard starts behaving differently. Instead of using the PSK from the previous handshake, HWWireGuard calculates a new PSK for the next handshake. The *newPSK* can either be generated with a simple hash function, as in (2.1), or by using the hardware device, as in (2.2). The hash function is far less costly from a performance standpoint, but from a security viewpoint using *hwOp* is preferable. Option 1 still enables an attacker who gets hold of a PSK to calculate all newPSKs afterward as long as the timestamp does not change. Option 2 prevents this by using the hardware device. HWMonitor loads newPSK into HWWireGuard, which can perform the next handshake. This loop runs until either the timestamp is changed or *stopMonitor()* is called.

An important function that will need to be established here is a *reset* function if the PSKs between the peers become asynchronous. Either by one peer closing and then opening the tunnel again or some other network interference. HWMonitor needs to recognize if the other peer probably does not share the same PSK anymore. A possible way to implement this is by counting the failed handshake attempts. If they exceed a specific number, HWMonitor resets to the PSK calculated by the current timestamp.

### 3.3.3 Version 3: HWWireGuard with Handshake Primitives

A big problem the previous versions have is that all PSKs are deterministic. In Version 1, the same PSK is used for the whole hour and possibly for multiple peer connections if they share the same hardware key. In Version 2, the PSK changes but is again deterministic and the same for all HWWireGuard connections with the same hardware key.

A unique PSK generated from a unique parameter of the handshake is much more preferable. Such a parameter could be the *ephemeral key* of the initiator. For both communication peers, this parameter is the same. HWMonitor only can generate the resulting PSK more than once if the same ephemeral key is part of the handshake, which is not something the attacker can influence. The only problem with this approach is that WireGuard generates the ephemeral key right before the handshake. While one could try to use the ephemeral key for the current PSK, there is a risk that the time frame is too small to get the ephemeral key out of the WireGuardGo backend to the hardware device, perform the hardware-backed operation and then load it into the same handshake as PSK. As a result, for this thesis, we will use the ephemeral key of the previous successful handshake. Unfortunately, this means that the first handshake still needs to be performed deterministically and with the timestamp.

Figure 3.9 shows what this looks like. The beginning is again identical to the other versions. The second loop, on the other hand, gets the ephemeral key of the last handshake in $\textcircled{3.1}$ and uses it as input for *hwOp* in $\textcircled{3.2}$. The resulting PSK is unique and only occurs again if WireGuard generates the ephemeral key twice. By still using the timestamp for the initial PSK, the reset operation can work exactly like in Version 2.

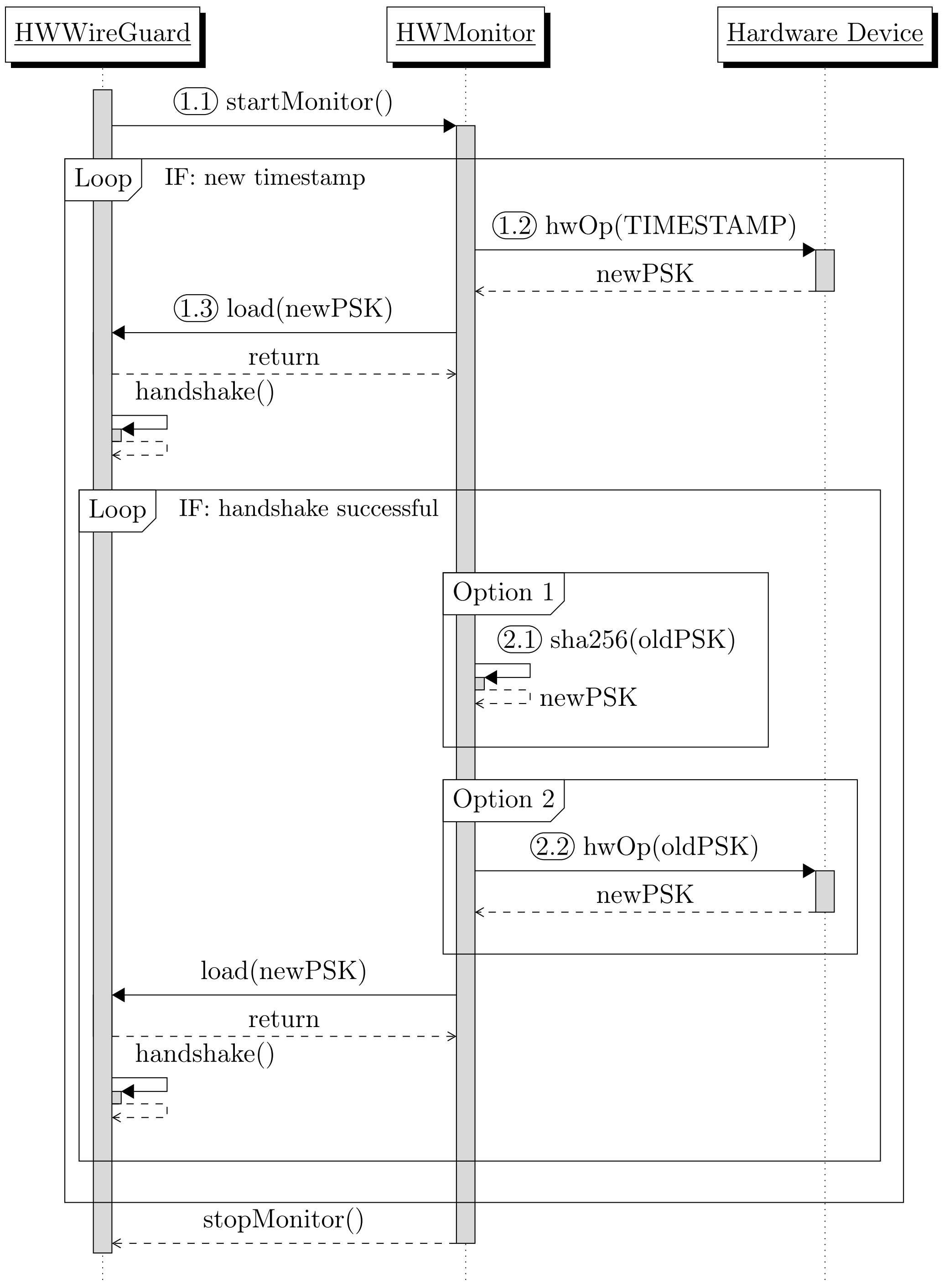


Figure 3.8: HWWireGuard Version 2.

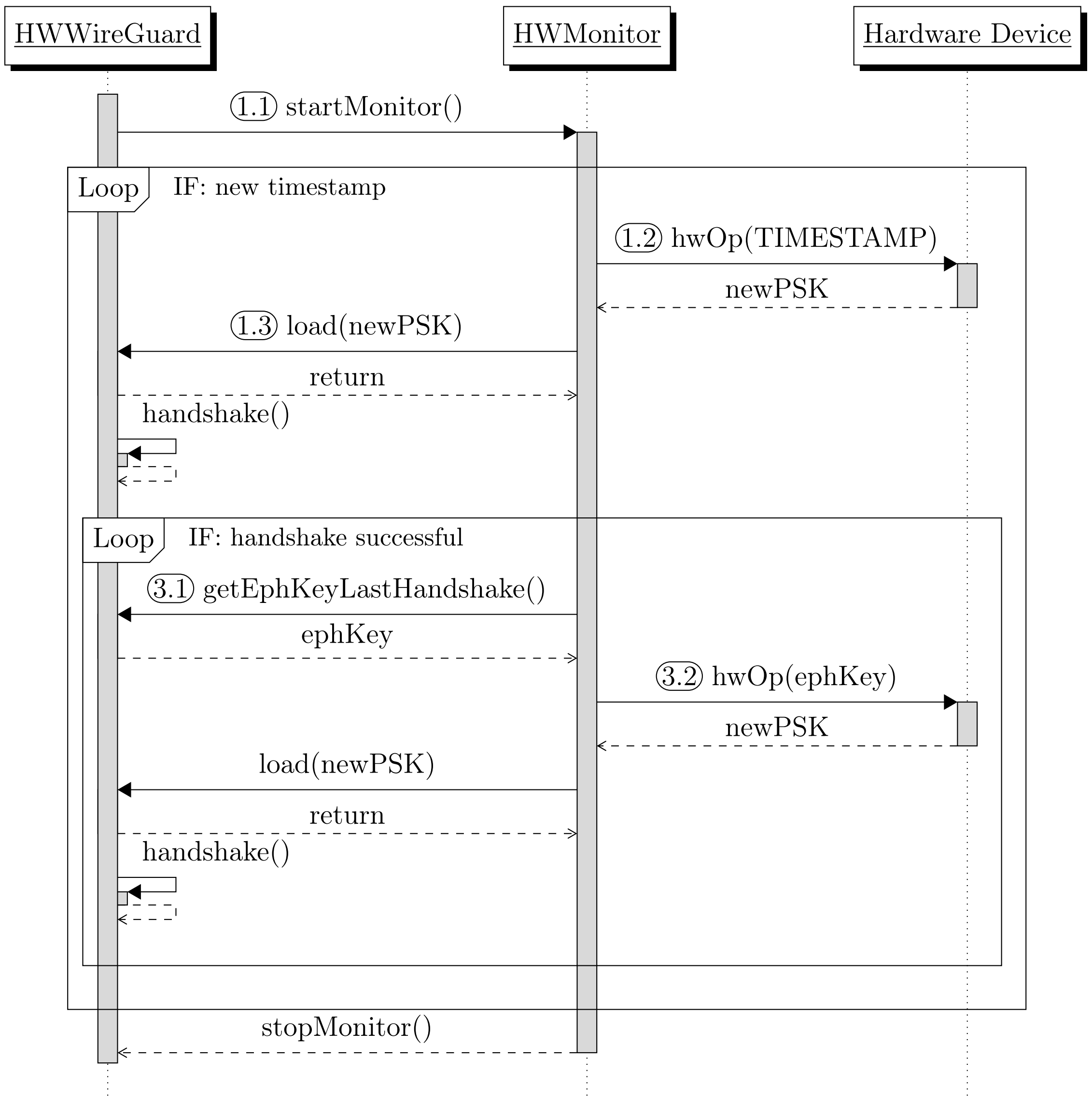


Figure 3.9: HWWireGuard Version 3.

# 4 Implementing a Hardware-Backed WireGuard on Android

This chapter examines the implementation of HWWireGuard on Android[1]. This thesis is not able to examine all changes made to the original WireGuard due to their scope. As a result, we can not go into every alteration especially because they are not located in one package or file but rather spread throughout the project.Furthermore, this thesis does not aim to discuss every change in detail but rather aspires to give insight into the overall structure of the HWWireGuard implementation and its most important classes.

To allow other developers to examine HWWireGuard more easily, all changes to the original WireGuard are surrounded by the tags *"Custom change begin"* and *"Custom change end"*. Altogether new classes are marked with the prefix *"HW"*. The following sections will present the most crucial source code sections and explain their function. By examining them in regards to the development version, each version can build upon the information already discussed. Version 2 and 3 mainly adopt the code changes of the previous version.

## 4.1 Implementing Version 1

Version 1 of HWWireGuard is the most fundamental and easiest to conceptualize. It takes the current timestamp with an accuracy of an hour and performs a hardware-backed operation with it. The resulting output is used as the new PSK for the current HWWireGuard connection. If the timestamp changes, the same process is performed again, keeping the PSK up to date.

Because Version 1 is the first version, much fundamental work is necessary to create the first working HWWireGuard prototype. The necessary changes include UI changes, hardware-backed operations, and the HWMonitor process. The implementation requires sizeable source code modifications examined in the following sections.

[1] `https://github.com/mike111-droid/HWWireGuard-android`, as of July 11, 2022

### 4.1.1 UI Changes

The first parts to implement are the UI changes for the user interaction. Using the infrastructure already provided by WireGuard allows us to achieve this with relatively few changes. The number of alterations is low because all UI changes are limited to the settings screen.

**Changing *preferences.xml*.** Android provides developers with the possibility to set layouts via *XML* files. WireGuard uses the file *preferences.xml* in the *ui* package under *ui/src/main/res/xml* to define the layout of the `SettingsActivity` responsible for displaying the settings. Here we can use *Preferences Categories* to visually separate the HWWireGuard exclusive settings from the standard WireGuard settings. Preference Categories in Android allow developers to group their preferences into groups with titles.

```xml
<com.wireguard.android.preference.VersionPreference/>
    <!-- Custom change begin -->
    <PreferenceCategory
        android:title="KEYSTORAGE SOLUTIONS">
        <DropDownPreference
            android:key="dropdown"
            android:title="Choose hardware-backed Key Storage"
            android:entryValues="@array/hwbacked"
            android:entries="@array/hwbackedlonger"
            app:useSimpleSummaryProvider="true" />
        <DropDownPreference
            android:key="dropdownAlgorithms"
            ... />
        <Preference
            android:key="import_rsa"
            android:summary="@string/import_key_summary_rsa"
            android:title="@string/import_key_rsa" />
        <Preference
            android:key="import_aes"
            ... />
    </PreferenceCategory>
    <!-- Custom change end -->
    <PreferenceCategory
        android:title="STANDARD SETTINGS">
        [STANDARD SETTINGS]
    </PreferenceCategory>
```

Listing 4.1: Preference layout in *ui/src/main/res/xml/preferences.xml*.

Listing 4.1 shows the changed parts of *preferences.xml*. The tags `<Preference-Category>` surround the new and old preference contents. The new HWWireGuard settings include two drop-down menus to select the hardware solution and algorithm and two buttons to import RSA and AES keys respectively. The referenced *@array* and

*@string* variables are stored in *ui/src/main/res/values/strings.xml*. The selected content of the drop-down fields can be looked up via the `HWApplication.getPreferencesDataStore()` function from anywhere in the code where the `HWApplication` class is accessible. The resulting UI can be seen in Figure 3.3.

**Changing the SettingsActivity Class.** As already mentioned, the `SettingsActivity` is launched when the user navigates to settings via the typical three vertical dots on the top right corner of the screen. Here we can define the RSA and AES key import for Android KeyStores, as shown in Listing 4.2. The settings activity uses the newly created Android KeyStore management class called `HWKeyStoreManager`. The `HWKeyStoreManager` class also provides the RSA and AES operations necessary for HWWireGuard. Before adding a new key, HWWireGuard removes any existing keys with the same alias. Afterward, HWWireGuard displays on the screen whether the import was successful or not via the `Toast.makeText()` function.

```kotlin
val keyStoreManager = HWKeyStoreManager()
preferenceManager.findPreference<Preference>("import_rsa")?
    .setOnPreferenceClickListener {
   keyStoreManager.deleteKey("rsa_key")
   if(keyStoreManager.addKeyStoreKeyRSA("rsa_key", "crt.pem",
       "private_key.der")) {
      Toast.makeText(activity, "Import was successful.",
          Toast.LENGTH_LONG).show()
    }else{
      Toast.makeText(activity, "Import was not successful.",
          Toast.LENGTH_LONG).show()
    }
}
preferenceManager.findPreference<Preference>("import_aes")?
    .setOnPreferenceClickListener {
   keyStoreManager.deleteKey("aes_key")
   if(keyStoreManager.addKeyStoreKeyAES("aes_key", "key.txt")) {
      ...
   }
}
```

Listing 4.2: *SettingsActivity.kt* in *ui/src/main/java/com/wireguard/android/activity*.

The functions `addKeyStoreKeyRSA` and `addKeyStoreKeyAES` read out the files *crt.pem* and *private_key.der*, or *key.txt* in the Download folder and create a key under the alias *rsa_key* or *aes_key*. The keys are both created with the options `setUserAuthenticationRequired(true)` and `setUserAuthenticationValidityDurationSeconds(6*60*60)`. The former requires biometric authentication of the user in order to use the key. The latter sets the time after the biometric authentication in which an operation is allowed to six hours. If the user has not authenticated themselves for six hours, they can not use the key. The unlocking of the phone also counts as an authentication. If the user needs to authenticate again and the app is in the background, HWWireGuard displays a notification. When HWWireGuard has opened again, the app removes the notification, displays a biometric prompt for authentication, and performs the necessary operation. The implementation of this feature is quite extensive, and the explanation exceeds the scope of this thesis which is why this section does not examine it in more detail.

**Creating functions for user authentication.** In order to use the hardware devices securely, they need to be able to authenticate the user before the tunnel is allowed to be started. For this, the `HWMonitor` class has functions implementing the authentication. Listing 4.3 shows the function `authenticate()` called at the start of the tunnel. Lines 2 and 6 show that HWMonitor selects different authentication functions depending on the selected hardware device.

```
1 private fun authenticate() {
2      if (mHWBackend == "SmartCardHSM") {
3        val hsmManager = HWHSMManager(mContext)
4        smartCardService = hsmManager.smartCardHSMCardService
5        smartCardService?.let { authenticateHSM(it, hsmManager) }
6      }else if(mHWBackend == "AndroidKeyStore") {
7        authenticateKeyStore()
8      }
9   }
```

Listing 4.3: Function for authentication in *ui/src/main/java/com/wireguard/android/hwwireguard/HWMonitor.kt*.

If Android KeyStores is selected, the monitor calls the function `authenticateKeyStore`, as shown in Listing 4.4. At first, `authenticateKeyStore` creates an authentication callback for possible results. For now, only the successful case is of interest. The callback for successful authentication only has to set the *run* variable to `TRUE` because the authentication is handled by Android internally. The authentication will last for six hours. Afterward, the callback object is used to create a `BiometricPrompt` object called *prompt*. Finally, the function uses the object *prompt* to call `authenticate` with the *promptInfo* defining options like title or `setConfirmationRequired(true)`. `authenticate` displays the biometric prompt to the user.

If the SmartCard-HSM is selected, `HWMonitor` creates a `hsmManager` and gets a `SmartCardHSMCardService` object to authenticate. After the authentication, this object performs all operations with the HSM. If it does not get adequately authenticated, it can not perform the necessary operations, and the user has to restart the tunnel to enter the correct PIN. Listing 4.5 illustrates parts of the called function `authenticateHSM`. Here `HWMonitor` constructs the alert dialog where the user enters the PIN. When the user presses the button *Enter*, the function uses the inputted PIN from the edit text field to check the PIN and authenticate the session. The variable *run* set to `TRUE` tells the `HWMonitor` process to start monitoring for changing timestamps.

```kotlin
private fun authenticateKeyStore() {
   val authCallback = object : BiometricPrompt.AuthenticationCallback() {
      ...
      override fun onAuthenticationSucceeded(result:
          BiometricPrompt.AuthenticationResult) {
         super.onAuthenticationSucceeded(result)
         run.set(true)
      }
      ...
   }
   val prompt = BiometricPrompt(mFragment,
       ContextCompat.getMainExecutor(mContext), authCallback)
   ...
   prompt.authenticate(promptInfo)
}
```

Listing 4.4: Function for authentication of Android KeyStores in *ui/src/main/java/com/wireguard/android/hwwireguard/HWMonitor.kt*.

```kotlin
private fun authenticateHSM(schsmcs: SmartCardHSMCardService, hsmManager:
    HWHSMManager) {
   ...
   val edittext = EditText(mContext)
   val alertDialogBuilder: AlertDialog.Builder =
       AlertDialog.Builder(mContext)
   ...
   alertDialogBuilder.setPositiveButton("Enter") { _, _ ->
      val pin = edittext.text.toString()
      hsmManager.checkPin(pin, schsmcs) {
         /* Defined in HWHSMManager class */
         if(!schsmcs.verifyPassword(null, 0, pin.getBytes())) {
            Log.i(TAG, "PIN is incorrect.")
         }
      }
      run.set(true)
      ...
   }
   ...
}
```

Listing 4.5: Function for authentication of SmartCard-HSM in *ui/src/main/java/com/wireguard/android/hwwireguard/HWMonitor.kt*.

### 4.1.2 Creating the HWMonitor Class

A class mentioned multiple times is `HWMonitor`. The prefix *"HW"* of this class shows that it is a newly implemented class for HWWireGuard. `HWMonitor` is mainly responsible for monitoring the HWWireGuard connection and performing the necessary PSK changes. Furthermore, it provides several attributes necessary for multiple functions in `HWMonitor` and functions in other places in the code. For instance, it offers several variables to a function in the class `HWBiometricAuthenticator`, called if the authentication for the Android KeyStore expires. There the function `keyStoreOperation` needs variables like *mContext* or *mFragment*.

The first thing to discuss is how the `HWMonitor` process starts. When the switch button activating the tunnel in WireGuard is toggled, WireGuard calls the `setTunnelState` function in the `BaseFragment` class. Because WireGuard does not support multiple tunnels on a non-rooted device, we do not have to consider the use case of several running tunnels. We will employ this function as a mounting point to start `HWMonitor` and turn WireGuard into HWWireGuard. The `HWMonitor` object called *monitor* is created in the `onCreate` function of the fragment, as seen in Listing 4.6. As already mentioned, `HWMonitor` requires *mContext*, *mActivity*, and *mFragment* to function. These variables are all parameters of the constructor.

```
private lateinit var monitor: HWMonitor
override fun onCreate(savedInstanceState: Bundle?) {
   monitor = HWMonitor(requireContext(), requireActivity(), this)
   super.onCreate(savedInstanceState)
}
```

Listing 4.6: *onCreate* in *ui/src/main/java/com/wireguard/android/fragment/BaseFragment.kt*.

Now the variable *monitor* can be used in `setTunnelState`. Listing 4.7 shows that `HWMonitor` only starts if either hardware device is selected. The variable *checked* is `TRUE` if the toggle switch is turned on and `FALSE` if it is turned off. If *checked* is `TRUE`, the member variable *mTunnel* is set to the current tunnel, and the `HWMonitor` process starts. In the latter case, the tunnel turns off, and `HWMonitor` stops.

Listing 4.8 illustrates the function `startMonitor`. Line 2 shows how `HWMonitor` runs in a coroutine, allowing everything inside of it to execute inside its own process. Afterward, the user authenticates using the functions explained in Section 4.1.1. The function `startMonitor` now pauses until the variable *run* is set to `TRUE` by the authentication functions. The variable type is an atomic boolean, allowing the variable to contain the same value across different processes. This consistency across processes is essential for stopping the monitor process because, as Listing 4.7 shows, `setTunnelState` calls the function `stopMonitor` from a different process. `stopMonitor` sets *run* to `FALSE` breaking the loop in Line 6 of Listing 4.8. In the end, the `finally` block ensures that the SmartCard-HSM is properly shut down.

The function `monitor`, called repeatedly in Line 6 of Listing 4.8, checks whether the timestamp has changed. It uses the member variable *mOldTimestamp* to save the previous timestamp. The class `HWTimestamp` in *hwwireguard/crypto* generates the timestamp. If the timestamp changed, `monitor` calculates the new PSK and loads it into the WireGuardGo backend.

```kotlin
fun setTunnelState(view: View, checked: Boolean) {
   ...
      /* Custom change begin */
      if(monitor.mHWBackend != "none") {
         if(checked) {
            monitor.setTunnel(tunnel)
            monitor.startMonitor()
         } else {
            monitor.stopMonitor()
         }
      }
      ...
      /* Custom change end */
      setTunnelStateWithPermissionsResult(tunnel, checked)
   }
```

Listing 4.7: *setTunnelState* in *ui/src/main/java/com/wireguard/android/fragment/BaseFragment.kt*.

```kotlin
fun startMonitor() {
   mActivity.applicationScope.launch {
      try {
         authenticate()
         while(!run.get()) { delay(1000) }
         while(run.get()) { monitor() }
      } catch (e: Exception) {
         Log.e(TAG, Log.getStackTraceString(e))
      } finally {
         if (mHWBackend == "SmartCardHSM") {
            try {
               SmartCard.shutdown()
            } catch (e: Exception) { ... }
      ...
}
```

Listing 4.8: *startMonitor* function in *ui/src/main/java/com/wireguard/android/hwwireguard/HWMonitor.kt*.

The function used for this is `loadNewPSK` and is fairly easy to implement in Version 1. By getting the configuration as a `Config` object from *mTunnel*, `loadNewPSK` can iterate through all peers and sets the new PSK within the `Config` object. The class `GoBackend` requires a new function `loadConfig` that allows it to load this new configuration into the WireGuardGo backend. `loadConfig` can use the already existing `IpcSet` function within the WireGuardGo backend in *device/uapi.go*. The function call of `IpcSet` is then performed in the function `loadConfig` from *tunnel/tools/libwg-go/api-android.go*, as shown in Listing 4.9. The parameter *settings* is the configuration of the WireGuard tunnel and *tunnelHandle* identifies the tunnel so the device can be accessed to call `IpcSet`. The function `loadConfig` inside *android-api.go* is integrated into the GoBackend class over Java's Native Interface in *tunnel/tools/libwg-go/jni.c*. As a result, the function loadConfig in *android-api.go* can be called with the Java class GoBackend.

```go
//export loadConfig
func loadConfig(tunnelHandle int32, settings string) int32 {
    handle, ok := tunnelHandles[tunnelHandle]
    if !ok {return -1}
    handle.device.IpcSet(settings)
    return 0
}
```

Listing 4.9: *loadConfig* function in *tunnel/tools/libwg-go/api-android.go*.

### 4.1.3 Hardware-Backed Operations

The last significant change implemented for HWWireGuard Version 1 is the hardware-backed operation called hwOp from Chapter 3. Depending on the hardware device selected, the function hwOp can be different. The classes HWHSMManager and HWKeyStoreManager implement all the functions necessary to interact with the selected device.

In both cases, the function's structure is similar. In practice, this means that in both hwOps the input and the resulting output of the hardware device are hashed in a SHA256 function. The hashing of the input allows developers to use these functions in cases where the input is bigger than the supported block sizes of the devices. The hashing of the output is somewhat redundant, but it further disguises the clean output of the device and ensures that the conversion function from bytes to the data type Key works without exception. The following sections explain the difference between the two functions in more detail.

#### SmartCard-HSM Operations

Listing 4.10 illustrates the function hsmOperation called out of HWMonitor. The function parameters are the *KeyType*, the string used as input, and a byte for the *keyID*. Furthermore, it requires the SmartCardHSMCardService object to interact with the HSM, which already received the PIN during user authentication in Section 4.1.1. In Lines 3 to 9, hsmOperation then selects the suitable operation for the selected algorithm. The functions hsmOperationRSA and hsmOperationAES show how the SmartCardHSMCardService is used to perform the two operations. For RSA, HWWireGuard uses the *"RSA-v1-5-SHA-256"* algorithm. The algorithm for the AES is identified by the *0x10* byte representing AES encryption in CBC mode. This function's output is static, meaning the SmartCard-HSM uses a static initialization vector (IV).

### Android KeyStore Operations

For Android KeyStores, the structure is similar to the HSM. The function called in `HWMonitor` is `keyStoreOperation`, and it similarly separates into two possible operations, either RSA or AES. Listing 4.11 presents these functions. In both functions, HWWireGuard must acquire an instance of either the cipher or the signature with the algorithm. Unfortunately, HWWireGuard can not use the same AES algorithm for Android KeyStores and SmartCard-HSM. Android KeyStores does not support AES encryption in CBC mode with repeating IVs, which is why ECB was selected here. Although Android KeyStore allows developers to choose the IV for a given cipher, it prevents them from repeating the IV. In order for both peers in HWWireGuard to create the same PSK, we would need to create the same IV independently on both devices. This solution would introduce another layer of complexity, where both peers also need to have a synchronous IV generation. ECB bypasses this problem because no IV is necessary.

In Listing 4.11, the *cipher* and *signature* objects need to be initialized with the corresponding key. Afterward, `sign` and `doFinal` perform the corresponding operations, after which the result is again hashed and transformed to the data type Key.

```
public Key hsmOperation(final KeyType keyType, final
    SmartCardHSMCardService schsmcs, final String init, final byte keyID) {
   ...
   if(keyType == KeyType.AES) {
      /* AES on HSM */
      res = hsmOperationAES(schsmcs, digest, keyID);
   }else if(keyType == KeyType.RSA){
      /* RSA on HSM */
      res = hsmOperationRSA(schsmcs, digest, keyID);
   }
   ...
}
public byte[] hsmOperationRSA(SmartCardHSMCardService schsmcs, byte[]
    digest, byte keyID) {
   SmartCardHSMRSAKey rsa2048Key = new SmartCardHSMRSAKey(keyID,
       "RSA-v1-5-SHA-256", (short) 2048);
   return schsmcs.signHash(rsa2048Key, "SHA256withRSA", "PKCS1_V15",
       digest);
}
private byte[] hsmOperationAES(SmartCardHSMCardService schsmcs, byte[]
    digest, byte keyID) {
   return schsmcs.deriveSymmetricKey(keyID, (byte) 0x10, digest);
}
```

Listing 4.10: Functions for hsmOperation *ui/src/main/com/wireguard/android/hw-wireguard/crypto/HWHSMManager.kt.*

```
private Key rsaOperation(final KeyStore.Entry keyEntry, final byte[]
    initBytes) {
  Signature sig = Signature.getInstance("SHA256WithRSA");
  sig.initSign(((KeyStore.PrivateKeyEntry) keyEntry).getPrivateKey());
  sig.update(initBytes);
  byte[] signature = sig.sign();
  return bytesToKey(sha256(signature));
}
private Key aesOperation(final KeyStore.Entry keyEntry, final byte[]
    initBytes) {
  Cipher cipher = Cipher.getInstance("AES/ECB/NoPadding");
  cipher.init(Cipher.ENCRYPT_MODE, ((KeyStore.SecretKeyEntry)
      keyEntry).getSecretKey());
  byte[] digest = sha256(initBytes);
  return bytesToKey(sha256(cipher.doFinal(digest)));
}
```

Listing 4.11: Functions for keyStoreOperation *ui/src/main/com/wireguard/android/hwwireguard/crypto/HWKeyStoreManager.kt.*

## 4.2 Implementing Version 2

Recall from Section 3.3.2 that in Version 2, we aim to change the PSK with every handshake by using the old PSK as input for either a hash function or hardware operation, depending on the option selected. With this, Version 2 introduces a few challenges that need to be solved to implement it correctly. Among the necessary changes are an expansion of the information provided by the WireGuardGo backend over the `getStatistics` function, the ability to load a new PSK for a specific peer, and the mechanism that creates a new PSK with every handshake. All these changes are needed to expand the function `monitor` in `HWMonitor`.

### 4.2.1 Function *monitorV2Extension*

The function `monitorV2Extension` contains the main changes to this HWWireGuard version. The function executes at the end of the original `monitor` function. Listing 4.12 shows how `monitorV2Extension` gets the necessary WireGuardGo backend data from the `getStatistics` function in the `GoBackend` class. By using `getStatistics`, `monitorV2Extension` can access the *lastHandshakeTime*. If WireGuard performed a successful handshake *lastHandshakeTime* changes. By saving the previous *lastHandshakeTime* in *mLastHandshakeTime*, `HWMonitor` can identify which peer had a successful handshake. In Line 7 of Listing 4.12, we iterate through all peers of the configuration, checking each one. In Line 11, the corresponding ratchet algorithm is called, depending on which option from 3.3.2 is preferred. The function `ratchetOptionX` needs to be able to perform the *ratchet* for just one peer, which is why it gets *peer* as a parameter. The following Section 4.2.2 explains the necessary implementation changes for this peer-specific function.

Finally, in Lines 13 and 14, `HWMonitor` checks whether the amount of failed handshakes indicates that the PSKs for both peers have become asynchronous, and a reset is necessary. If `handleFailedHandshake` detects six failed handshakes, the saved *initPSK* loads into the WireGuardGo backend. The variable *initPSK* contains the PSK for the current timestamp to prevent unnecessary hardware device access. If `HWMonitor` detects 13 failed handshakes, it calculates *newPSK* with the hardware device and current timestamp.

```
 1  private suspend fun monitorV2Extension() {
 2     val config = mTunnel!!.config ?: return
 3     ...
 4     val stats = HWApplication.getBackend().getStatistics(mTunnel)
 5     ...
 6     val lastHandshakeTime = stats.lastHandshakeTime
 7     for (peer in config.peers) {
 8        ...
 9        if (lastHandshakeTime[peer.publicKey] !=
              mLastHandshakeTime[peer.publicKey]) {
10           lastHandshakeTime[peer.publicKey]?.let {
                 mLastHandshakeTime.put(peer.publicKey, it) }
11           ratchetOptionX(config, peer)
12        }
13        val handshakeAttempts = stats.handshakeAttempts[peer.publicKey]
14        handleFailedHandshake(handshakeAttempts, config, peer)
15     }
16  }
```

Listing 4.12: Function *monitorV2Extension* in *ui/src/main/com/wireguard/android/hwwireguard/HWMonitor.kt.*

### 4.2.2 Peer-Specific Functions

Section 4.2.1 introduced the necessity of changing PSKs for specific peers. Mainly three functions rely on this feature: `ratchetOptionPSK`, `hsmOperation` and `keyStoreOperation`. The function used by all those functions is `loadNewPSK`. The following paragraphs give a short explanation of how they each work.

To prevent additional functions from becoming necessary, Version 2 alters the existing functions by adding the parameter `peer`. If the caller of these functions sets peer to null, it operates for all peers. Otherwise, the functions use the given peer and only change this specific peer.

***ratchetOptionX*.** Listing 4.13 shows the structure of both `ratchetOption1` and `ratchetOption2`. At first, the functions iterate through all peers to find the one selected over the parameter *peer*. When found, it gets the PSK from *config* and then performs a *ratchet*, which is either a simple SHA256 operation or a hardware-backed cryptographic operation. Lines 6 to 8 show the code from `ratchetOption1`, where SHA256 is used. Option 2 is shown in Lines 9 to 14.

```
1  private fun ratchetOptionX(config: Config, peer: Peer) {
2     ...
3     for((counter, peerIteration) in config.peers.withIndex()) {
4        if(peerIteration == peer) {
5           val psk = config.peers[counter].preSharedKey.get()
6           /* Option 1 */
7           val newPSK = ratchetManager.ratchet(psk)
8           loadNewPSK(config, newPSK, peer)
9           /* Option 2 */
10          if(mHWBackend == "SmartCardHSM") {
11             hsmOperation(psk.toBase64(), peerIteration)
12          }else if(mHWBackend == "AndroidKeyStore") {
13             keyStoreOperation(psk.toBase64(), peerIteration)
14          }
15       }
16    }
17 }
```

Listing 4.13: Function *ratchetOptionX* in *ui/src/main/com/wireguard/android/hw-wireguard/HWMonitor.kt*.

**`hsmOperation` and `keyStoreOperation`.** The functions `hsmOpertion` and `keyStore-Operation` have a straightforward structure. As in Version 1, they call the suitable functions from `HWHSMManager` and `HWKeyStoreManager`. The only real difference is that the `loadNewPSK` function now also takes the parameter *peer*, which is directly passed on to `loadNewPSK`. Furthermore, `hsmOperation` and `keyStoreOperation` need to save the PSK with the current timestamp into the member variable *initPSK*.

**`loadNewPSK`.** Listing 4.14 shows the critical parts of the function `loadNewPSK`. The function starts a separate process to ensure that the central monitoring process is not blocked. Line 3 shows how `loadNewPSK` iterates through all peers. If the caller sets the parameter *peer* to null, the function must load the PSK to all peers. Otherwise, the function only changes the PSK for the specified peer. Within this function, there are also quite a few shutdown locks missing from Listing 4.14. These locks prevent the WireGuard tunnel from shutting down if the monitor is currently calling or about to call `HWApplication.getBackend()`. If this happens for a shutdown tunnel, the app crashes.

A new function in `loadNewPSK` is the `GoBackend` member function `loadPSK`. This function is necessary because the function `loadConfig` triggers a new handshake with the peers automatically. So `HWMonitor` needs a new function that allows updating the PSK right after a handshake without triggering a new one. As a result, `loadNewPSK` can not use the function `IpcSet` from the WireGuardGo implementation. Instead, HWWireGuard needs a new function called `IpcSetPSK` implemented for this thesis in the custom WireGuardGo backend[2]. IpcSetPSK prevents the automatic handshake by loading only the PSK directly into the backend. The call of IpcSetPSK works the same as IpcSet by creating the altered Config data object and giving it as a parameter to the function. The function loadNewPSK can again call IpcSetPSK over the file in *tunnel/tools/libwg-go/api-android.go* incorporated over Java's Native Interface into the GoBackend class.

```
fun loadNewPSK(config: Config, newPSK: Key, peer: Peer?) {
   mActivity.applicationScope.launch {
      for((counter, peerIterate) in config.peers.withIndex()) {
         ...
         if(peer == null) {
            config.peers[counter].setPreSharedKey(newPSK)
            HWApplication.getBackend().loadConfig(config)
         }
         if(peer == peerIterate) {
            config.peers[counter].setPreSharedKey(newPSK)
            HWApplication.getBackend().loadPSK(config)
         }
         ...
      }
   }
}
```

Listing 4.14: Function *loadNewPSK* in *ui/src/main/com/wireguard/android/hwwireguard/HWMonitor.kt.*

### 4.2.3 Extending the *getStatistics* function

For monitorV2Extension to work, the function needs several variables from inside the WireGuardGo backend. WireGuard already offers a function for some of these variables. For instance, in Section 4.2.1, HWMonitor requires access to the last handshake time, which is provided by HWApplication.getBackend().getStatistics. The number of handshake attempts is a variable the WireGuardGo backend does not provide with getStatistics. As a result, we need to count the number of incoming or outgoing handshake initiations to supply this to HWMonitor.

**Adding variables to wgGetConfig output.** wgGetConfig is a function in *tunnel/tools/libwg-go/api-android.go* which returns a string containing several key-value pairs. This string is constructed inside the function IpcGetOperation inside the WireGuardGo backend in *device/uapi.go*. The function already writes the key-value pair for the last handshake time into this configuration string. However, the handshake attempts still need to be added, which is possible with just one line of code, as shown in Listing 4.15. Finally, we must add the member variable *actualHandshakeAttempts*

[2]https://github.com/mike111-droid/HWWireGuard-androidGoBackendmod, as of July 11, 2022

to the `timers` data struct and ensure it is increased or reset in the correct places of the WireGuardGo backend code.

```go
func (device *Device) IpcGetOperation(w io.Writer) error {
   ...
   /* Custom change begin */
   sendf("handshakeAttempts=%d", peer.timers.actualHandshakeAttempts)
   /* Custom change end */
   ...
}
```

Listing 4.15: Function *IpcGetOperation* from the WireGuardGo backend in *device/uapi.go*.

The first place to do this is in *device/peer.go* where the `Peer` struct with its subsequent `timers` struct is defined by adding the variable *actualHandshakeAttempts*, as seen in Listing 4.16. Then we need to increase this new variable when new handshake initiation messages are either sent or received or reset it back to zero if a handshake was successful.

In *device/send.go*, there is a function called `SendHandshakeInitiation`. At the end of this function, *actualHandshakeAttempts* must be increased by one. The same change needs to be added in *device/receive.go* in the function `RoutineHandshake` within the case `MessageInitiationType`. Lastly, we need to ensure that *actualHandshakeAttempts* is reset to zero if a handshake was successful, which is possible with the function `timersHandshakeComplete` in *device/timers.go*. WireGuardGo calls this function for successful handshakes.

```go
type Peer struct {
   ...
   timers struct {
      ...
      /* Custom change begin */
      actualHandshakeAttempts uint32
      /* Custom change end */
   }
   ...
}
```

Listing 4.16: Adding *actualHandshakeAttempts* to Peer data structure in the WireGuardGo backend *device/peer.go*.

**Adding variables to *Statistics* class.** The `Statistics` class in *tunnel/src/main/java/com/wireguard/android/backend* is responsible for storing the output from `wgGetConfig`. Therefore, we need to add two *HashMaps* for *handshakeAttempts* and *lastHandshakeTime*. The HashMaps take key-value pairs as inputs where the key is the peer's public key for identification. Additionally, the class `Statistics` needs two functions to add key-value pairs to the object.

Finally, the last alteration happens in the `GoBackend` class in the same folder. The function `wgGetConfig` is called inside `getStatistics` which works as the parser from configuration string to `Statistics` object. Here we have to add two if-clauses looking for a line starting with the key values *"handshakeAttempts"* or *"last_handshake_time_sec"* and add the corresponding key-value pair via the functions implemented in `Statistics`. Now `monitorV2Extension` has all the necessary values from the WireGuardGo backend.

## 4.3 Implementing Version 3

As already discussed in Section 3.3.3, Version 3 aims to replace the deterministic chain of PSKs from Version 2 with a PSK chain that is non-deterministic. To achieve this goal, HWWireGuard includes the ephemeral key from the WireGuardGo backend in calculating the new PSK. For this, we modify the `ratchet` function described in Section 4.3.1. Furthermore, Section 4.3.2 discusses an implementation solution to minimize the time where a deterministic PSK is used.

### 4.3.1 Function *ratchetV3*

All changes for `ratchetV3` use methods already used in the versions before. Similar to Section 4.2.3, Version 3 must extend `getStatistics` to include the ephemeral key. For this, WireGuardGo needs a new variable inside the `handshake` structure in *device/noise-protocol.go* and write the ephemeral key into this variable. Then `ratchetV3` replaces the PSK in Listing 4.13 with the ephemeral key.

### 4.3.2 Minimize the First-Handshake-Problem

As discussed in Section 3.3.3, the problem of Version 3 is the PSK of the first handshake, which is deterministic. While there is no straightforward solution to this problem, there is a simple way to minimize the time the encryption keys generated from this handshake are used. The function to load the new PSK into the backend used right after starting the tunnel is `loadConfig`. As a byproduct, this function automatically triggers a handshake renegotiation. This renegotiation was acceptable when starting the tunnel for the first time because a handshake needed to be performed. This behavior was problematic for later operations because the PSK was loaded into the backend after a successful handshake. An automatically triggered handshake renegotiation was not helpful. However, this feature could help shorten the time when an encryption key derived from a timestamp-PSK is used.

To implement this, `HWMonitor` needs an additional member variable because the object needs to save whether the current PSK depends on the timestamp for each peer. The HashMap *timestampPSKInUse* allows this. At the start of the monitor, `startMonitor` fills it with `FALSE` for all peers. As soon as `HWMonitor` loads a new timestamp-PSK, *timestampPSKInUse* for this peer is set to TRUE. In the function `monitorV2Extension`, `HWMonitor` usually checks for successful handshakes. By simply adding a function `fixTimestampPSK` with the if-check whether *timestampPSKInUse* is set to `TRUE`, `HWMonitor` now can replace the timestamp-PSK right after the handshake and trigger a new handshake in kind. The first handshake is diverted from its intended use to a negotiation for a shared input necessary to create a non-deterministic PSK on both sides.

# 5 Performance Evaluation

In this chapter, we examine the performance aspects of both Android KeyStores and SmartCard-HSM outside and inside of HWWireGuard. All performance measurements are done on a Samsung Galaxy S8 with the specifications shown in Table 5.1.

| |
|---|
| **Module Number**: SM-G950F |
| **Operating System**: Android 9.0 Pie - Samsung One UI 1.0 |
| **Kernel Version**: 4.4 |
| **Processor**: Exynos Octa-core (4×2.3 GHz M2 Mongoose & 4×1.7 GHz) Cortex-A53 (GTS) |
| **Battery**: 3000 mAh |

Table 5.1: Samsung Galaxy S8 specifications.

**Approach.** For short-term analysis of the CPU, we will use the *Android Studio Profiler* [Pro]. The Android Profiler is a good tool for examining the CPU usage and execution time of specific functions in short time frames. Furthermore, the recording can be triggered directly inside the source code, making it highly accurate. Unfortunately, the Android Profilers battery tool is inadequate. It scales the amount of energy consumption with terms like *Light* or *Medium*, making it challenging to compare the SmartCard-HSM and Android KeyStore operations, which do not differ much in energy consumption.

For this reason, we will use *Android's Battery Historian* [Bat] in combination with the *Android Debug Bridge (adb)* and its bugreports. Here we have access to the devices' estimated power use, the CPU's battery voltage, and temperature. Contrary to the Android Profiler, the Battery Historian offers the opportunity to examine performance statistics over a longer time frame.

## 5.1 SmartCard-HSM vs. Android KeyStores

Before directly comparing WireGuardHSM with WireGuardKeyStore, this thesis first examines the two different hardware devices outside of WireGuard. Here we determine the actual performance delta between the SmartCard-HSM and Android KeyStores.

### 5.1.1 CPU Usage

The Android Profiler measured the CPU usage in an application called *PerformanceComp*[1] that offered two buttons, each starting a benchmarking function for both the SmartCard-HSM and Android KeyStore. The following paragraphs discuss how the app PerformanceComp was structured, how the measurements were obtained, what the measurements are, and how they can be assessed.

**PerformanceComp.** As already mentioned, Android Profiler benchmarked the CPU usage, allowing us to start traces from inside the code, as seen in Listing 5.1. During the test, ten RSA and ten AES operations were executed. With a higher number of iterations, the Android Profiler reaches a limit of traces it can collect and suddenly stops recording. The test was performed five times for higher accuracy, and the results averaged.

Listing 5.1 illustrate the generalized benchmark function started by PerformanceComp. The function exists both for Android KeyStores and the SmartCard-HSM. The *preprocess* function differs depending on the hardware backend used. These are all functions that can be excluded from the performance analysis because they only need to be performed once and, as such, become negligible with an ever-increasing number of encryption or signature operations. As 5.1 shows, the Android Profiler trace is started in Line 4 and finished in Line 11. The trace captures the execution time or wall duration of the *operationRSA* and *operationAES* functions and all functions they call. Furthermore, it records the CPU duration a function has and the standard deviation. Unfortunately, the functions operationAES use different algorithms. The reasons behind this decision are discussed in Section 4.1.3.

As input PerformanceComp uses a constant 32-byte input. For the legal input sizes of 16 and 64 bytes, the performance values did not change significantly and, as such, are not relevant.

```
1  private void test() {
2     ...
3     preprocess();
4     Debug.startMethodTracing("test.trace");
5     for(int idx = 0; idx < 10; idx++) {
6        operationRSA(..., (byte[]) input);
7     }
8     for(int idx = 0; idx < 10; idx++) {
9        operationAES(..., (byte[]) input);
10    }
11    Debug.stopMethodTracing();
12     ...
13 }
```

Listing 5.1: Generalized Benchmark Test.

[1]`https://github.com/mike111-droid/PerformanceComp`, as of July 11, 2022

**Analyzing the measurements.** As expected, the SmartCard-HSM is much less performant than the Android KeyStore operations, as seen in Table 5.2. The wall duration measures the actual execution time of the function. CPU duration indicates how long the function was occupying CPU resources. We can calculate the average CPU usage from these values and break it down to a device with eight cores by dividing it by eight. The average standard deviation indicates how reliable the average wall duration is. Unfortunately for the SmartCard-HSM, the standard deviation for processes with longer execution times tends to be higher. To illustrate this, we examine how significant the share of the standard deviation is, compared to the wall duration.

Table 5.2 shows that the wall time of the RSA operation on the SmartCard-HSM is about ten times longer than with Android Keystores. Similarly, the AES operation is about five times longer. The CPU usage for SmartCard-HSM operations is around 40%. This CPU usage is about 5% of the device's CPU resources on a device with eight cores. With the Android KeyStore operations, we can observe quite a difference in CPU duration between AES and RSA operations. An AES operation requires almost 50%, while a RSA operation only needs about 20%. Surprisingly the more calculation-intensive RSA operations require far fewer CPU resources than AES operations. On the SmartCard-HSM, the difference between RSA and AES is not significant. Nonetheless, RSA operations on both hardware devices take longer than AES operations.

The standard deviation for the SmartCard-HSM seems to be much higher than for the Android KeyStores. To put this into perspective, we must examine how great the deviation is compared to the execution time. Here we can see that the SmartCard-HSM AES operation standard deviation only makes up about 4% of the wall duration, while the Android KeyStores can achieve standard deviations of up to 13%. The same thing can be observed in AES operations.

Overall, the measurements show that Android KeyStore operations are much cheaper, as expected. The integrated form allows shorter communication paths and access to more considerable resources. Nonetheless, the SmartCard-HSM operations have not disqualified themselves, especially in an application where the number of calls is limited, as in HWWireGuard.

| | **SmartCard-HSM** | | **Android KeyStore** | |
|---|---|---|---|---|
| | RSA | AES | RSA | AES |
| Average Wall Duration | 399.71ms | 103.18ms | 46.51ms | 21.1ms |
| Average CPU Duration | 158.07ms | 51.13ms | 8.02ms | 9.65ms |
| Average Std. Dev. of Wall Duration | 15.59ms | 12.66ms | 6.06ms | 4.82ms |
| Average CPU Usage | 39.42% | 41.51% | 17.02% | 47.23% |
| Estimated average CPU Usage for 8 cores | 4.93% | 5.19% | 2.13% | 5.90% |
| Std. Dev. / Wall Duration | 3.9% | 12.27% | 13.03% | 22.84% |

Table 5.2: Performance of hardware-backed cryptographic operations.

### 5.1.2 Battery Usage

The battery consumption is measured via bugreports and displayed with Android's Battery Historian. Unfortunately, the Battery Historian does not allow to start the traces from inside the source code. However, luckily bugreport saves the starting point of the displayed application. In order to create general tests, the aim is to reduce the user interaction to a minimum so that other processes do not pollute the traces and the testing environment stays the same. All tests were performed with a running screen and in airplane mode to ensure this uniformity over multiple tests. The goal of minimizing user interaction was achieved by creating specific apps called *AKS15000*, *AKS30MIN*, *HSM15000*, and *HSM30MIN*, which started the benchmark right after launch. Each app can be classified into a specific group, as explained in the following paragraphs.

1. **Time-fixed**: In this group, the app runs for 30 minutes. During the 30 minutes, it either performs AES or RSA operations. The resulting battery consumption shows the battery required for each hardware device. It is expected that Android KeyStores can perform far more operations within 30 minutes which is why we also need to examine the operation-fixed mode.
2. **Operation-fixed**: In this group, the app runs only as long as is needed to perform 15,000 operations. As a result, the hardware device with longer execution times is at a disadvantage because other devices like the screen also use more battery.

In Figures 5.1 and 5.3, AKS15000 and HSM15000 indicate operation-fixed apps with 15,000 operation executions. More than 15,000 operations were unfeasible because the runtime for HSM15000 approached one hour. Equally tricky was choosing a lower number because AKS15000 did not register a battery change for lower execution numbers. AKS30MIN and HSM30MIN illustrate the results for the time-fixed apps with a 30-minute runtime. All tests were run five times, and the discharge was averaged.

Figure 5.1 shows the actual discharge rate of the device during execution. The difference between AKS15000 and HSM15000 is the most obvious. The difference is mainly due to the long runtimes of HSM15000 and the accompanying display discharge. To put it into perspective, the HSM15000 for RSA takes about six times longer than the Android KeyStore implementation. This same difference does not occur in the time-fixed modes. AKS30MIN and HSM30MIN are relatively similar in battery consumption.

Figure 5.3 illustrates the estimated power use of the application. Compared to Figure 5.1, the estimated power consumption of the apps and their hardware devices is reasonably low. Devices like the screen are responsible for far more significant battery usage. Nonetheless, we see the same behavior for the operation-fixed modes as before where longer running tasks need more battery. Interestingly, the difference between AKS30MIN and HSM30MIN seems to be greater in favor of HSM30MIN, which is unexpected because the overall battery discharge would suggest a relatively equal discharge rate. However, the differences are in the area of 0.03% and 0.05%, which is rather low and do not seem to have any impact on the overall discharge.

The analysis shows that the difference between the SmartCard-HSM and Android KeyStores is relatively negligible compared to other devices like the screen. Furthermore, it shows that the discharge rates between both devices are comparable and influenced mainly by execution times. As shown in Section 5.1.1, the SmartCard-HSM has some disadvantages, but the main question is whether these disadvantages translate into a resource-intensive app like HWWireGuard.

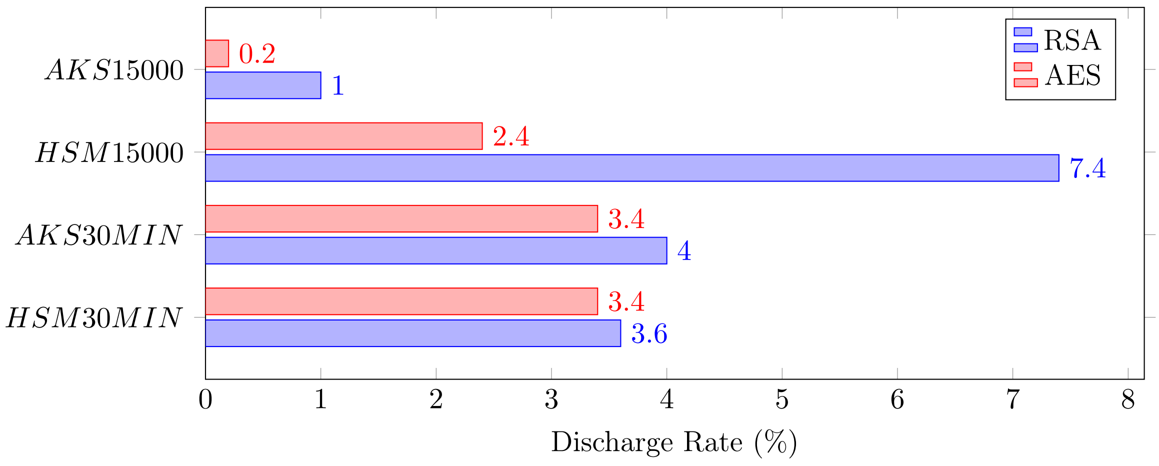


Figure 5.1: Discharge Rate during runtime.

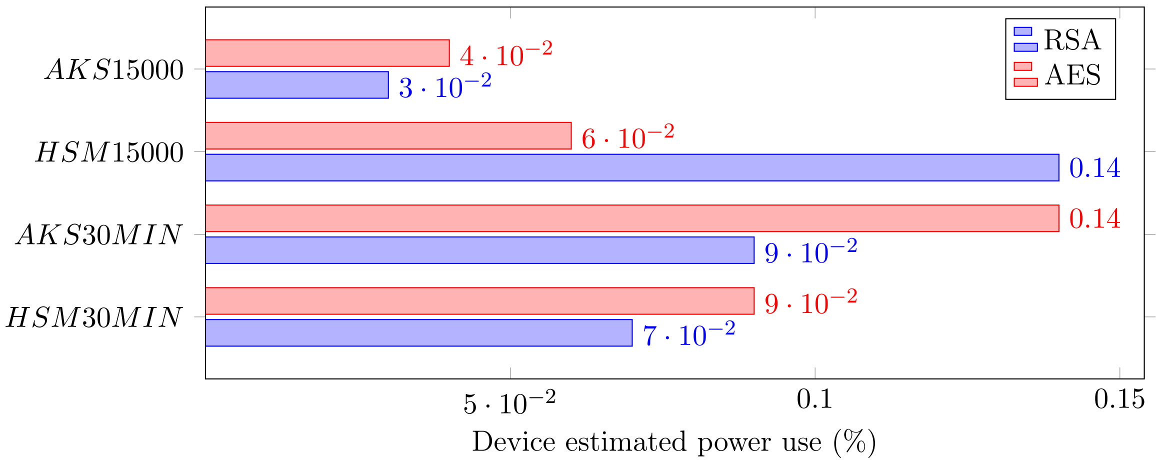


Figure 5.2: Device estimated power use of app.

## 5.2 WireGuardKeyStore vs. WireGuardHSM

After examining the performance of the hardware devices individually, this section looks at the performance of HWWireGuard and the different development versions. Section 5.2.1 assess the execution times of relevant functions in each version. Afterward, Section 5.2.2 examines the battery usage of the HWWireGuard versions with different hardware devices and algorithms. All CPU and battery usage measurements were performed in an ideal environment, meaning that no failed handshakes occurred, and no PSK resets were necessary. A more extensive analysis with induced PSK synchronization failures was not considered due to the resulting complexity of the tests.

### 5.2.1 CPU Usage

In the following, we evaluate different functions of HWWireGuard using the possible selections of hardware devices and algorithms. Apart from the first function, each function can be attributed to one of the implementation versions. The coming paragraphs describe the functions, in which versions they occur, and their execution times. All resulting measurements are illustrated in Table 5.3 and were gathered by measuring the traces ten times and averaging the results from the Android Profiler.

**Function *newTimestamp*.** The function *newTimestamp* exists in every HWWireGuard version. HWMonitor uses the function to check for changing timestamps, as explained in Section 4.1.2. This code section is called if a new timestamp has occurred, which is every hour. It contains the calculation of the new PSK with the hardware device and the start of the separate process responsible for loading the new PSK into the backend.

Table 5.3 in Part (a) shows the measured results. The average percentage of CPU usage shows that the difference between hardware devices is far less relevant than the difference between algorithms. For instance, both hardware devices have a CPU usage of around 60% for *newTimestamp* using RSA operations. Compared to the 80% when the function uses AES operations. The percentage of CPU usage is around two times higher than for the isolated hardware device operations examined in Section 5.1.1. The additional functions performed inside *newTimestamp* increased the necessary CPU resources. Regarding the algorithm selection, *AES* is the better choice for both hardware devices regarding execution times.

Furthermore, the wall duration shows that the massive advantages of Android KeyStores have not translated to the actual implementation. For instance, the RSA operation's function is only about 200ms longer, not ten times as before. The function using AES with Android KeyStores even takes longer than the SmartCard-HSM one, with about 30ms. Responsible for this are additional functions like the hashing with SHA256 and the function transforming the resulting bytes to a WireGuard Key explained in Section 4.1.3. Furthermore, Android KeyStores could not profit from calculating the *preprocess* outside of the operation, as in Section 5.1.1. The function executes the *preprocess* for both hardware devices each time an operation is called.

| | | **SmartCard-HSM** | | **Android KeyStore** | |
|---|---|---|---|---|---|
| | | RSA | AES | RSA | AES |
| **(a)** | Average Wall Duration | 614,81ms | 373,75ms | 461,56ms | 402,6ms |
| | Average CPU Duration | 369,37ms | 300,39ms | 319,99ms | 333,15ms |
| | Average CPU Usage | 60,08% | 80,37% | 69,33% | 82,75% |
| | Average CPU Usage for 8 cores | 7,51% | 10,05% | 8,67% | 10,34% |
| **(b)** | Average Wall Duration | 62,37ms | | | |
| | Average CPU Duration | 61,8ms | | | |
| | Average CPU Usage | 99,09% | | | |
| | Average CPU Usage for 8 cores | 12,39% | | | |
| **(c)** | Average Wall Duration | 638,91ms | 336,36ms | 482,87ms | 382,81ms |
| | Average CPU Duration | 396,32ms | 266,91ms | 335,24ms | 306,63ms |
| | Average CPU Usage | 62,03% | 79,35% | 69,43% | 80,10% |
| | Average CPU Usage for 8 cores | 7,75% | 9,92% | 8,68% | 10,01% |
| **(d)** | Average Wall Duration | 648,5ms | 336,32ms | 475,32ms | 374,93ms |
| | Average CPU Duration | 407,25ms | 269,37ms | 330,66ms | 301,71ms |
| | Average CPU Usage | 62,80% | 80,09% | 69,57% | 80,47% |
| | Average CPU Usage for 8 cores | 7,85% | 10,01% | 8,70% | 10,06% |

Table 5.3: Execution Times for different functions inside HWWireGuard.
**(a)** Function called when timestamp changed.
**(b)** Function *monitorV2Extension* with *ratchetOption1*.
**(c)** Function *monitorV2Extension* with *ratchetOption2*.
**(d)** Function *monitorV2Extension* with *ratchetV3*.

**Function *monitorV2Extension* with *ratchetOption1*.** The function *monitorV2Extension* uses different *ratchet* functions depending on the version. The function *ratchetOption1* uses the SHA256 hashing algorithm, as explained in Section 3.3.2. In a typical connection, this function is called with each handshake every two minutes. Alongside this *ratchet* function, *monitorV2Extension* also handles the case of too many failed handshakes. In this test scenario, all function tests were performed in an ideal environment, meaning that possible PSK resets were unnecessary, and this part of the function was never called.

The results in Table 5.3 Part (b) show that the function is independent of the hardware device due to the hashing function used as the ratchet function. These measurements are included to contrast them to Parts (c) and (d), which are far less performant. Using the hardware devices for the ratchet function increases the execution time of *monitorV2Extension* by about five to ten times.

**Function *monitorV2Extension* with *ratchetOption2*.** The function *ratchetOption2* uses the hardware device to ratchet and is implemented in HWWireGuard Version 2 with Option 2. As before, the function executes with every successful handshake.

Part (c) illustrates that all measurements are similar to Part (a). The main functional difference between Part (c) and (a) is that the latter also contains *getStatistics* calls and several checks on whether a handshake was successful or failed. As explained in the previous paragraph, the case for failed handshakes never executes because the test scenario performs in an ideal environment. These checks and function calls seem to have a far less profound impact on the execution time and CPU usage than the hardware operations.

**Function *monitorV2Extension* with *ratchetV3*.** The function *ratchetV3* is similar to *ratchetOptionV2* and only differs regarding the input used for the hardware operations. Here *ratchetV3* uses the ephemeral key called out of the WireGuardGo backend with *getStatistics*. However, as seen in the paragraph above, this function is a very low-cost call regarding CPU resources. As a result, Part (d) of Table 5.3 shows that the execution times and CPU usage is very similar to both Part (a) and (c).

Overall, the results illustrate that the hardware operations are the most resource-intensive tasks in the functions called regularly in HWWireGuard. Other checks and function calls like *getStatistics* are somewhat irrelevant. The results for Versions 2.2 and 3 also mean they use the most CPU resources because they use ratchet functions using hardware operations and are called every two minutes. To what extent the higher frequency of these calls makes using the HWWireGuard versions uneconomical regarding battery usage will be examined in the following section.

### 5.2.2 Battery Usage

As examined before, different HWWireGuard versions execute different functions with differing frequencies. This section examines how much these disparities influence battery consumption over time. We run every HWWireGuard version with each hardware device and algorithm for one hour. Within this hour, HWWireGuard only sends keep-alive messages every 15 seconds. The keep-alive messages guarantee that the tunnel established is used by a persistent package flow. The time frame of one hour ensures that HWWireGuard Version 1 executes at least one hardware operation. The measurements were performed three times, and the resulting hourly discharge rates were averaged. Another measure to provide consistency across multiple tests was to have the same battery level at the start of every recording at 100%. Additionally, the screen was turned on during the recording, and the screen brightness was the same across all recordings.

Figure 5.3 illustrates the battery discharge rate per hour. It shows that the battery consumption between the standard WireGuard and HWWireGuard in Versions 1 and 2.1 is the same. A factor not examined here is the impact of PSK resets on the

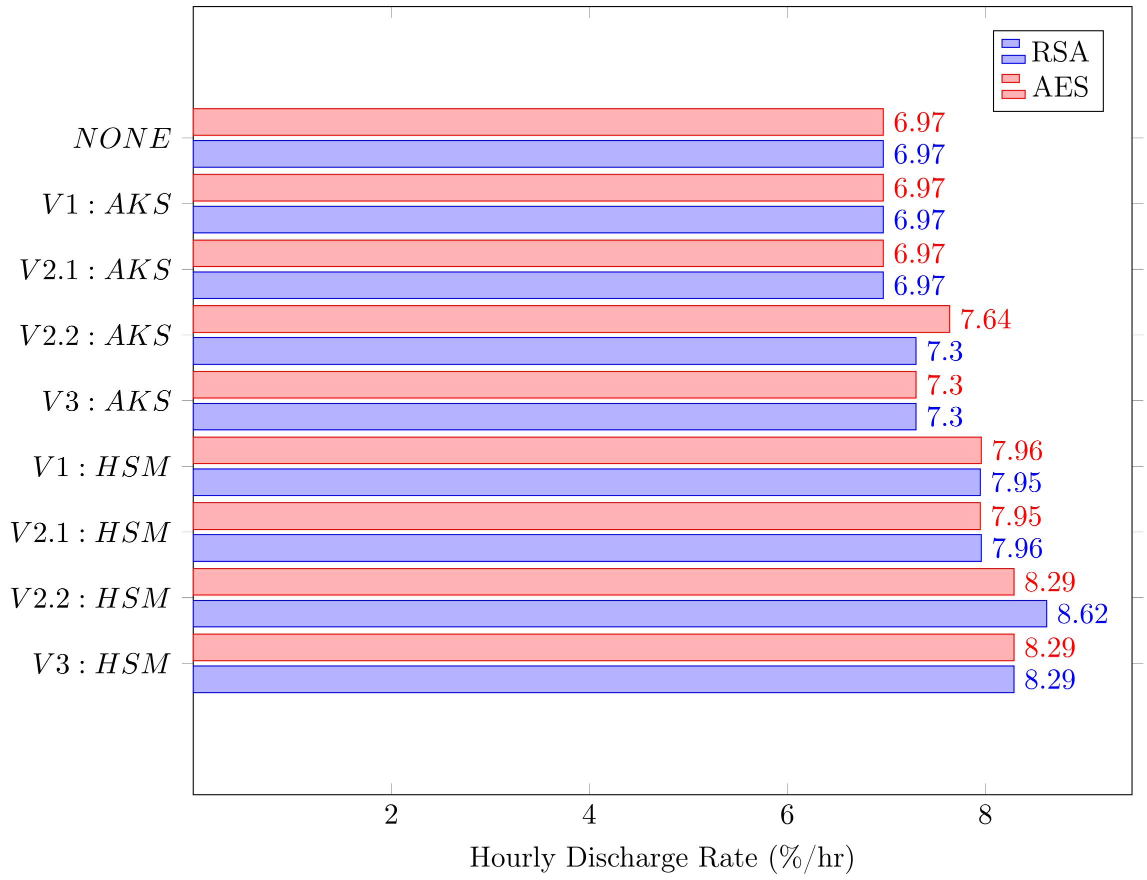


Figure 5.3: Discharge rate (%/hr) during HWWireGuard execution.

battery consumption, so the results here are inconclusive regarding the comparison of HWWireGuard and WireGuard. However, we can say that the ideal monitoring process which runs in all HWWireGuard versions seems to have a minimal impact on battery consumption. Ideal in this context means that no PSK resets were necessary. In Versions 2.2 and 3, we can see an increase of about 0.4% because HWWireGuard performs a hardware operation with every handshake every two minutes. The hash function in Version 2.1 did not have the same effect. These observations are valid for both hardware devices. Figure 5.3 also shows that the battery consumption of all versions is about 1% higher if HWWireGuard uses the SmartCard-HSM. This percentage is equivalent to a 14% increase. Contrary to the previous section's results, there does not exist a significant difference between RSA and AES. Except for Version 2.2, all tests result in the same battery consumption independent of the selected algorithm.

Overall, the measurements show a difference in battery consumption between WireGuard and HWWireGuard, except for Versions 1 and 2.1 using Android KeyStores. There is the anticipated delta between Versions 1 and 2.1 and Versions 2.2 and 3, where the number of hardware operations is much higher. Furthermore, a difference exists between the hardware devices, leading to the SmartCard-HSM needing more power. This performance delta was expected due to the external nature of the HSM. However, the measurements do not differ greatly, and the different versions allow users to decide how much battery they want to sacrifice for a more comprehensive security concept.

# 6 Discussion

This chapter will discuss the advantages and disadvantages of the hardware-backed WireGuard implementation in general and of the different hardware devices. This discussion includes the consideration of security-critical characteristics, as well as their usability. Section 6.1 compares HWWireGuard with WireGuard, while Section 6.2 discusses the SmartCard-HSM and Android KeyStores. The examination includes the identification of several overarching categories relevant to possible users. The categories classified in this thesis are *Usability*, *Performance*, and *Security*. Within these categories, we identify several characteristics where either HWWireGuard and WireGuard or the SmartCard-HSM and Android KeyStores differ. Contrasting these characteristics allows us to identify qualities in each category that are more or less desirable and discern which element is better.

Within the category *Usability*, we group general characteristics that might interest users. These include additional features the user can access, cost considerations, or concerns about user-friendliness. Separated into their own category are all characteristics related to *Security* which contain differing security features and advantages. The category *Performance* possesses qualities informed by the outcome of Chapter 5, allowing for a well-balanced discussion.

## 6.1 Hardware-Backed WireGuard vs. WireGuard

After designing and implementing HWWireGuard in this thesis, it is essential to discuss the actual advantages and disadvantages of the implementation. For this assessment, we look at several qualities either implementation possesses classified into the three categories discussed above. Table 6.1 contains these characteristics.

**Usability.** HWWireGuard retains three qualities that might lead to possible users favoring the original WireGuard.

1. **Kernel backend implementation**: On rooted devices, users can use the kernel backend of WireGuard instead of the WireGuardGo backend. This backend's chief advantage is its ability to open multiple tunnels. For some users, this could be a necessary feature. However, HWWireGuard does not support the kernel backend yet.

| | | WG | HWWG |
|---|---|---|---|
| **Usability** | Kernel backend implementation | ✓ | ✗ |
| | Easy accessibility | ✓ | ✗ |
| | Stable implementation | ✓ | ✗ |
| **Performance** | Slight performance advantages | ✓ | ✗ |
| **Security** | Protection against rooted attackers | ✗ | ✓ |
| | Selection of trusted hardware | ✗ | ✓ |
| | Upgradeable | ✗ | ✓ |

Table 6.1: Characteristics of WireGuard and HWWireGuard.

2. **Easy accessibility**: Two factors constrain the accessibility of HWWireGuard. Firstly, HWWireGuard is not accessible to users who are not technically inclined. It only exists as a Github project that users need to download and compile on their own. Secondly, HWWireGuard entails a minimal amount of key management tasks. Users have to ensure that devices that are supposed to establish a tunnel have duplicate keys on their hardware devices. This extra work can lead to possible users preferring the more straightforward solution over the possibly more secure one. As a result, HWWireGuard is more helpful to companies and organizations where technical-orientated administrators can take care of these activities. HWWireGuard is not yet practical for the everyday end-user.

3. **Stable implementation**: This last point is likely the biggest flaw of HWWireGuard. HWWireGuard still is very much a prototype. As such, the app does not run as reliably as WireGuard and crashes from time to time. Implementing features like the shutdown locks, as mentioned in Section 4.2.2, does not prevent bugs and crashes completely but only limits them. As a result, proper everyday usage of HWWireGuard still requires some work.

**Performance.** The app HWWireGuard has more processes executing. As such, the resulting app requires more CPU and battery resources. Nonetheless, the performance disparity was not observable for all versions. For instance, the battery usage of WireGuard is comparable to HWWireGuard Version 1 with Android KeyStores and RSA. The hourly battery consumption for both is about 6.97%. It is likely that the increased battery consumption, which has to exist because of additional processes, is so small that they do not show up during a one-hour test. Higher versions of HWWireGuard harm the battery consumption by increasing it by 0.4%. It is important to remember that these results were obtained in an environment that did not necessitate a PSK reset. To get a holistic analysis of the difference between WireGuard and HWWireGuard this would have to be examined further.

**Security.** In this category, HWWireGuard offers the most advantages. The main design goal of HWWireGuard was to increase security by anchoring it inside a dedicated hardware device. As a result, HWWireGuard possesses the following qualities regarding security.

1. **Protection against rooted attackers**: The idea behind both TEEs and the SmartCard-HSM is to protect the key material from attackers even if they have physical access to the device. A fully rooted attacker controls the non-secure world of the TEE completely. The Keymaster TA used by Android KeyStores is supposed to protect key material even against these attackers. The SmartCard-HSM does not care whether the device it interacts with is rooted or not. It just protects the key material. The standard WireGuard protects its connections solely with the help of OS file permissions which do not hold up to an attacker with root privileges.

2. **Selection of trusted hardware**: HWWireGuard allows users to select which hardware device to trust. With this option, users can decide which manufacturer to rely on. WireGuard does not offer such a selection. Only the OSes file permissions ensure the protection of the VPN tunnel.

3. **Upgradeable**: The structure of HWWireGuard allows developers to replace the underline hardware operation with their implementation or hardware device. The already existing framework and class infrastructure permit upgrades with limited alterations to the code.

Overall, HWWireGuard offers a possible solution to companies and organizations that want to explore a hardware-backed VPN solution with WireGuard. Especially in the category *Security*, HWWireGuard offers several advantages. Nonetheless, the app is still a prototype and, as such, still needs development work to create a stable application. Additional work will also be required if the kernel backend is supposed to be supported. The Backend class for the kernel backend uses a root shell that takes the same commands as WireGuard on Linux. With *WireGuardHSM-linux*[1], there already exists a hardware-backed WireGuard implementation with the SmartCard-HSM on Linux. The same commands for loading new PSKs into WireGuard should also work for the kernel backend.

## 6.2 Android KeyStore vs. SmartCard-HSM

One of the main questions posed in this thesis is whether the SmartCard-HSM is a possible alternative to Android KeyStores. The HWWireGuard implementation has already shown that developers can use the SmartCard-HSM instead of Android KeyStores in this particular scenario. However, which hardware device is the better solution? The following will try to answer this question. As in the section above, this question will be analyzed within the context of the three defined categories. Table 6.2 illustrates the differing characteristics and will be explained in the following paragraphs.

[1] `https://github.com/mike111-droid/WireGuardHSM-linux`

| | | HSM | AKS |
|---|---|---|---|
| **Usability** | External device | ✓ | ✗ |
| | One-time session authentication | ✓ | ✗ |
| | More supported features | ✗ | ✓ |
| **Performance** | Longer execution times | ✓ | ✗ |
| **Security** | Number of manufacturers | 1 | >1 |
| | Specialized hardware device | ✓ | ✗ |
| | Additional authentication layer | ✓ | ✗ |
| | Self-destruct feature | ✓ | ✗ |

Table 6.2: Characteristics of SmartCard-HSM and Android KeyStores.

**Usability.** When talking about usability in the context of these devices, users are also developers. They are the ones who need to decide which solution is practical for their application. As a result, not only the actual end-user of the hardware device is relevant but also the ease with which developers can build projects around these devices. Therefore, some of the qualities listed pertain to how easily a hardware device can be integrated into Android.

1. **External device**: The main difference between the SmartCard-HSM and Android KeyStores is that the former is an external device, while the latter is integrated internally. As a result, the features the devices offer developers and users differ. Firstly, the SmartCard-HSM is not bound to just one device but can also be used on other mobile phones. It can therefore act much more like a smart card for authentication, as the name suggests. Furthermore, the HSM is also supported on OSes like Linux or Windows and, as such, possesses cross-platform capabilities. Users can use the same HSM on different devices to secure applications such as HWWireGuard or *WireGuardHSM-linux* mentioned above. Another advantage of the external device is its replaceability. If the device breaks, users can easily replace it at a price far lower than a new mobile phone would be. The replacement can also take place if the hardware device becomes obsolete. Unfortunately, the flip side of this argument is that the external nature of the HSM also means that mobile phone users have to buy it in addition to their mobile phones. With around 90€[2], these additional costs are rather significant. Another disadvantage of an external device is the way firmware updates have to be installed by the user manually. Android devices include these security-critical firmware updates within the Android OS update pipeline.

2. **One-time session authentication**: Another difference between the hardware devices is the authentication methodology. Access to Android KeyStores is controlled via user authentication on the mobile phone. This authentication

[2]`https://www.cardomatic.de/epages/64510967.sf/de_DE/?ObjectPath=/Shops/64510967/Products/SmartCard-HSM-MicroSD`, as of July, 2022

can happen in different apps or by unlocking the screen. Depending on the key, cryptographic operations can happen within a set time after authentication. HWWireGuard sets this time frame to six hours. Authentication on the SmartCard-HSM happens only once for a particular SmartCardHSMService object, as explained in Section 4.1.1. The way the hardware devices handle authentication differently allows developers different possibilities. Android KeyStores offers an authentication across several apps, while the SmartCard-HSM only authenticates for one specific app. In the example of HWWireGuard, the SmartCard-HSM's approach to authentication fits better into the design. In HWWireGuard, the user is supposed to authenticate themselves once for the whole tunnel duration. As a result, the authentication time frame for Android KeyStores had to be set relatively high to prevent users from having to authenticate themselves more than once. Still, users may be required to authenticate themselves more than once during a tunnel runtime.

3. **More supported features**: A quality of Android KeyStores is that they are deeply integrated into the Android OS infrastructure. As a result, specific interactions with the device are easier for developers. For instance, Android offers UI support for Android KeyStores by providing biometric prompts for authentication. For the SmartCard-HSM, such an authentication prompt had to be designed and implemented. The number of supported algorithms is another area where Android KeyStores offers more features than the HSM. For instance, when it comes to different AES modes [Andc], Android KeyStores provides CBC, CTR, GCM, and ECB. The SmartCard-HSM, on the other hand, only offers AES CBC [Sma].

**Performance.** The performance evaluation in Section 5.1 has shown that SmartCard-HSM is less performant. As a result, operations take longer, and the battery consumption is higher. However, Section 5.2 also illustrates that the actual disadvantages inside HWWireGuard are not as severe as Section 5.1 suggests. Concerning battery consumption, SmartCard-HSM led to an increase in battery consumption of about 14%. The CPU usage was not significantly better for Android KeyStores. In the case of functions using the AES algorithm on the HSM, the execution times were even shorter than those using AES with Android KeyStores. By better utilizing the functions allocated in the *preprocess*, mentioned in Section 5.1, the performance of HWWireGuard with Android KeyStores can be improved further.

**Security.** The primary motivation behind this thesis comes from several characteristics the SmartCard-HSM offers as an external device compared to Android KeyStores. These qualities can be grouped within the category *Security* and circumvent some of the problems of TEEs discussed in Chapter 1.

1. **Number of manufacturers**: The first problem the SmartCard-HSM offers a solution to is the number of manufacturers. As seen in Chapter 1, there are

several market-leading mobile phone manufacturers. These manufacturers also implement the TEEs to varying degrees of success. Developers creating Android applications have no choice but to trust these manufacturers to implement a secure hardware-backed solution. Besides these manufacturers, developers also have to trust different models and, therefore, different implementations within the same manufacturer. Some businesses or government agencies could be interested in reducing this dependence. For instance, German government agencies that want to decrease their reliance on foreign companies could be interested because the SmartCard-HSM manufacturer is located in Germany, contrary to the mobile phone producers. In times when state-level actors are a threat, such solutions are practical and easy. Furthermore, developers who build their apps with the SmartCard-HSM appropriate a concept that also occurs within WireGuard. Where WireGuard is opinionated regarding algorithms or transport protocols, apps secured by the SmartCard-HSM are opinionated about their hardware-backed security.

2. **Specialized hardware device**: This point addresses an argument often made in the academic papers mentioned in Chapter 1. The TA implementing Android KeyStores is the Keymaster TA. This TA is integrated into the TEE as a whole. Within this TEE, other TAs also use the resources managed by the TEE OS. Past vulnerabilities have shown attackers can compromise the Keymaster TA by using other TAs. This exploitation is possible due to the complex structure of TEEs. The SmartCard-HSM, on the other hand, is far less complex and only runs the equivalent of the Keymaster TA on its hardware. The HSM is a specialized hardware device, contrary to the multi-tool approach of TEEs. This approach makes security audits simpler and less complex and offers fewer attack vectors to adversaries. However, we can not argue that, therefore, the HSM is more secure. In the past, HSMs have also shown insufficient software implementations. We only observe several criticized qualities of TEEs that do not exist in the SmartCard-HSM. Further security analysis of the SmartCard-HSM is required to examine this assumption.

3. **Additional authentication layer**: As discussed above, Android KeyStore's authentication is done with the same credentials used for unlocking the mobile phone. The SmartCard-HSM uses a dedicated PIN set by the user during the device's initialization. As a result, the device stays protected even if the mobile phone credentials for unlocking the device are compromised, e.g., the fingerprint or a mobile phone PIN.

4. **Self-destruct feature**: Another feature protecting the SmartCard-HSM is a limit of failed PINs allowed. If a user fails the authentication of the user PIN three times, the device is locked. Users can unlock the device with the *Security Officer PIN (SO-PIN)*. If this authentication fails 15 times, then the whole device is destroyed.

Overall, this section has shown that both hardware devices possess different qualities. The question of which hardware device is the better solution can not be answered generally. Instead, we see that both devices have different strengths and possible users have to examine their specific use-case. For use cases where security is highly prioritized, and financial costs are not as relevant, the HSM offers features that Android KeyStores can not match. If developers are looking for a fast and easy way to anchor their security within a hardware device without extra costs, Android KeyStores would be more suitable.

# 7 Conclusion

This thesis examined the SmartCard-HSM as a possible alternative to Android KeyStores using TEEs. The goals from Chapter 1 illustrate the points examined in this thesis.

Firstly, the SmartCard-HSM has shown itself capable of being integrated into an application like Android KeyStores. We therefore designed and implemented the application HWWireGuard, which offers the security-critical advantage of anchoring the encryption within specialized hardware devices responsible for storing and using the cryptographic keys. We argue that HWWireGuard is a prototype showing the possibility of integrating hardware-backed operations into WireGuard. However, to become a capable replacement for WireGuard, developers still have to overcome several bugs to make the application functional for possible end-users.

Secondly, the executed performance evaluation has shown that the SmartCard-HSM does not significantly increase the performance requirements on the mobile device.

Thirdly, the thesis argued that the HSM offers several security-critical advantages for safety-conscious users and developers that, in some use cases, can compensate for the performance disadvantages. In particular, the concept of opinionated app designs which only allow one hardware-backed device to perform cryptographic operations can be attractive to some developers and government agencies. This concept also reduces the reliance on big mobile phone manufacturers and their willingness to implement proper hardware-backed security concepts. Furthermore, it might discourage companies from utilizing concepts like security-by-obscurity, and they might start establishing more sustainable, less closed-source solutions. Nevertheless, for all its possibilities, the SmartCard-HSM will not be able to replace TEEs as a whole because it only fulfills the functions of the Keymaster TA.

**Future Work.** In this paper, we mainly examined the integration of the SmartCard-HSM inside an Android application. Here, we only had the opportunity to discuss security-related characteristics of the SmartCard-HSM. The discussion leaves room for further research analyzing the HSM with a proper security analysis. Such an analysis would provide a basis to discuss the assumption that the less complex design of the SmartCard-HSM is advantageous and enhances security. Unfortunately, vulnerable software implementations are also a problem occurring in HSMs.

Furthermore, there is the possibility of further advancing HWWireGuard. Firstly, the stability of the current application can be improved. Secondly, a proper examination of the consequences a changing PSK has on the WireGuard protocol is necessary. Thirdly, a more integrated inclusion of hardware devices can be achieved by replacing the WireGuardGo backend with a WireGuard implementation in Java, making interactions with these devices from inside the backend possible.

# List of Figures

# List of Tables

# List of Listings

# Bibliography

[AAD21] Gunnar Alendal, Stefan Axelsson, and Geir Olav Dyrkolbotn. Chip chop — smashing the mobile phone secure chip for fun and digital forensics. *Forensic Science International: Digital Investigation*, 31, 2021.

[Anda] Android Studios Developers: Vertrauenswürdiges TEE. `https://source.android.com/security/trusty`, as of July 11, 2022.

[Andb] Developers: Android keystore system. `https://developer.android.com/training/articles/keystore.html`, as of July 11, 2022.

[Andc] Developers: Android Keystore System. `https://android-doc.github.io/training/articles/keystore.html`, as of July 11, 2022.

[Bat] Android Studios Developers: Analyze power use with Battery Historian. `https://developer.android.com/topic/performance/power/battery-historian`, as of July 11, 2022.

[Cara] CardContact Systems GmbH. OpenCard Framework. `https://www.openscdp.org/ocf/`, as of July 11, 2022.

[Carb] CardContact Systems GmbH. SmartCard-HSM. `https://www.smartcard-hsm.com`, as of July 11, 2022.

[CSFP20] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems. *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

[Dava] David Wong. Hardware Solutions To Highly-Adversarial Environments Part 1: Whitebox Crypto vs Smart Cards vs Secure Elements vs Host-Card Emulation (HCE). `https://www.cryptologie.net/article/499/hardware-solutions-to-highly-adversarial-environments-whitebox-crypto-vs-tpm-vs-tee-vs-secure-enclaves-vs-secure-elements-vs-hsm-vs-cloudhsm-vs-kms-part-1/`, as of July 11, 2022.

[Davb] David Wong. Hardware Solutions To Highly-Adversarial Environments Part 2: HSM vs TPM vs Secure Enclave. `https://www.cryptologie.net/`

article/500/hardware-solutions-to-highly-adversarial-environments-part-2-hsm-vs-tpm-vs-secure-enclave/, as of July 11, 2022.

[Gala] Gal Beniamini. Exploring Qualcomm's Secure Execution Environment. https://bits-please.blogspot.com/2016/04/exploring-qualcomms-secure-execution.html, as of July 11, 2022.

[Galb] Gal Beniamini. Extracting Qualcomm's KeyMaster Keys - Breaking Android Full Disk Encryption. https://bits-please.blogspot.com/2016/06/extracting-qualcomms-keymaster-keys.html, as of July 11, 2022.

[Galc] Gal Beniamini. QSEE privilege escalation vulnerability and exploit. https://bits-please.blogspot.com/2016/05/qsee-privilege-escalation-vulnerability.html, as of July 11, 2022.

[Gald] Gal Beniamini. Trust Issues: Exploiting TrustZone TEEs. https://googleprojectzero.blogspot.com/2017/07/trust-issues-exploiting-trustzone-tees.html, as of July 11, 2022.

[Jas] Jason A. Donenfeld. WireGuard: Next Generation Kernel Network Tunnel. https://www.ndss-symposium.org/wp-content/uploads/2017/09/ndss2017_04A-3_Donenfeld_paper.pdf, as of July 11, 2022.

[LKP$^+$11] Junho Lee, Haeng-Seok Ko, SangHyun Park, Myungwon Seo, and Injung Kim. Study on Secure Mobile Communication based on the Hardware Security Module. *UBICOMM 2011 : The Fifth International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*, 2011.

[Mar] Marcel Busch and Johannes Westphal and Tilo Mueller. Unearthing the TrustedCore: A Critical Review on Huawei's Trusted Execution Environment. https://www.usenix.org/conference/woot20/presentation/busch, as of July 11, 2022.

[NoA] War of the Worlds - Hijacking the Linux Kernel from QSEE. https://bits-please.blogspot.com/2016/05/war-of-worlds-hijacking-linux-kernel.html, as of July 11, 2022.

[Pet] Peter Smirnoff. Understanding Hardware Security Modules (HSMs). https://www.cryptomathic.com/news-events/blog/understanding-hardware-security-modules-hsms, as of July 11, 2022.

[Pro] Android Studios Developers: The Android Profiler. https://developer.android.com/studio/profile/android-profiler, as of July 11, 2022.

[PS19] Sandro Pinto and Nuno Santos. Demystifying Arm TrustZone: A Comprehensive Survey. *ACM Computing Surveys*, 51(130), 2019.

[Sch21] Andreas Schwier. SmartCard-HSM: AGD User Manual for Version 3.5. 2021.

[Sin22] Rajinder Singh. An Overview of Android Operating System and Its Security Features. *Int. Journal of Engineering Research and Applications*, 2022.

[Sma] SmartCard-HSM 4K Data Sheet. `https://www.smartcard-hsm.com/docs/sc-hsm-4k-datasheet.pdf`, as of July 11, 2022.

[SRW14] Alon Shakevsky, Eyal Ronen, and Avishai Wool. Trust Dies in Darkness: Shedding Light on Samsung's TrustZone Keymaster Design. *USENIX Security Symposium*, 4(2), 2014.

[Str] Android Studios Developers: Hardware security module. `https://developer.android.com/training/articles/keystore#HardwareSecurityModule`, as of July 11, 2022.

[Tru] Chapter 3. TrustZone Hardware Architecture. `https://developer.arm.com/documentation/PRD29-GENC-009492/c/TrustZone-Hardware-Architecture?lang=en`, as of July 11, 2022.