Politecnico di Torino

Cybersecurity for Embedded Systems
01UDNOV

Master's Degree in Computer Engineering

Track 9 - Techniques for authentication of mobile devices
Project Report

Candidates:
Andrea Perno - 296185
Lorenzo Sciara - 303462
Omar Scicolone - 296492
Basilio Spinello - 292856

Referee:
Prof. Paolo Prinetto

Contents

List of Figures

Abstract

The project addresses the challenge of authenticating data originating from a smartphone device, aiming to provide a methodology to prove the origin of data to an external verifier. The unique identification of a device is a crucial aspect with multiple implications. In addition to ensuring security and authentication in IT operations, it allows you to track the use of devices, simplifying resource management and improving the user experience. Furthermore, this unique identification can prevent counterfeiting and protect sensitive data from unauthorized access. In summary, the ability to uniquely identify each device is critical to creating a trusted, efficient, and personalized computing environment. In particular, we want to have sure proof that a generic data was actually produced by a certain device.

Leveraging a secure area of the main processor of Android smartphones as a hardware-backed root of trust, the Trusted Execution Environment, applications perform cryptographic operations that lead to the creation and attestation of cryptographic keys. This process yields a unique and reliable identifier, associated with the device, that can be used in the verifying phase to effectively guaranteeing and proving the authenticity of the sent data. Furthermore, we considered the use of SIM card information as alternative identifiers, but they have not been considered for implementation purposes as it did not properly address the project goals. Rather they could be used to strengthen the verification process in future developments. Our progress was guided by official Android documentation and hands-on experimentation using Android Studio.

CHAPTER 1

Introduction

The project aims to address the challenge of authenticating data originating from a smartphone device, by leveraging unique features specific to that device. The problem can be described as follows: Suppose there is a device, on which you have no physical control however, you can execute software that sends you data, which are purportedly generated on the device, such as a screenshot. The question then arises: How can you prove that the data indeed originated from the device and were not generated by a malicious third party?

The goal of this project is to define a methodology to prove to an external verifier that the data in question was indeed generated on the specific device. We have achieved this objective on Android smartphones leveraging the TEE (Trusted Execution Environment) of the device as Hardware-Backed root of trust for the correct usage of the Keystore module and the associated APIs provided by Android.

In practice, our software is able to generate an Asymmetric key-pair to perform some cryptographic operations, retrieve the associated certificate chain with which the key-pair is hierarchically signed, and perform the **Key Attestation operation** to attest the *trustworthiness* of the certificates up to the root of trust represented by the Google certificate. This process allows us to obtain a **trusty and unique identifier** associated with the device, which is the TEE Serial Number of the attested device, that can efficiently guarantee and prove the authenticity of the sent data.

The project's approach involves several steps:

- Firstly, we conducted a research to identify all possible identifiers present on an Android device, classifying them according to two main criteria: accessibility and uniqueness of the identifier.
- Unfortunately, the most interesting identifiers have proved impossible to use due to numerous limitations introduced in the latest versions of Android
- Ultimately, we concluded that the most effective way to identify the device was through its serial number of the Trusted Execution Environment (TEE). The correctness of the TEE serial number is given by the verification of the certificate chain provided by the keystore.

An alternative approach to achieving our objective was to design software capable of retrieving various information related to the device's SIM card (or SIM cards in the case of multiple SIMs present in the same device), including the ICCID (Integrated Circuit Card ID). Although these pieces of information do not directly identify the device, they can be used as identifiers for the SIM card itself, thus potentially increasing the accuracy of identification when multiple identifiers are available.

The most effective way to proceed through each phase was to follow the documentation available on the official Android web page and simultaneously hands-on experimentation with each new feature using Android Studio.

CHAPTER 2

Background

In this section we will introduce some concepts that will be useful for understanding the operations done and described later in the report.

2.1 Asymmetric cryptography

The Asymmetric cryptography, also known as public-key cryptography, is a cryptographic system that uses a pair of keys to perform cryptographic operations: a **public key** and a **private key**. These keys are mathematically related, but while the public key can be freely distributed and disclosed, the private key must be kept secret. This technique can be used for several purposes, including Secure Data Transmission, Key Exchange, Secure Login and Authentication, but what we are interested in analyzing is the **digital signature**. It is possible to use asymmetric cryptography features in Android through the Android Keystore system and various cryptographic libraries available for Android development like Java Cryptography Architecture (JCA), which provides important classes like KeyPairGenerator, Signature, and Cipher.

2.2 Digital signature

A Digital Signature is a cryptographic technique used to provide authenticity, integrity, and non-repudiation to digital messages or documents. It is generated using asymmetric cryptography, where a pair of keys (public key and private key) is used. The objective is to bound the data to the sender (the signer), in such way that is possible to verify that only the real signer could have signed the document, and that the data have not been manipulated in any way after the signature. It involves the use of an Hash function that generates a digest of the data to be signed and the key pair to encrypt/decrypt the digest. The following properties are guaranteed:

- **Integrity of the data**: if some part of the data change, the digest will be different from the one encrypted, and the verifier can detect any modification;
- **Authenticity of the data**: the signature is strictly related to the signed data (e.g. same signer with different data results in different signatures, on the contrary to the paper handmade signature which can be copied and pasted on different documents) so they are exactly the data that the signer wanted to sign;
- **Authentication of the signer**: this can be obtained thanks to the usage of the key pair, where the private key is used only by the owner of the key pair, without sharing it to anyone.

- **Non-repudiation**: if the key pair is certified by means of a certificate, for example an X509 certificate, that bounds the key pair with the legal proprietary, emitted by a trusted Certification Authority, this feature is added to the signature which means: the proprietary can not deny a signature on a document made with its key pair, with legal consequences. This is optional.

The two main phases of this operation are therefore the **signature**, using the private key (sender side), and the **verification**, using the public key (receiver side).

2.3 Hash functions

A hash function is a one-way mathematical function that takes an input (or message) of any length and produces a fixed-size output, called a hash or **digest**. The hash function processes the input in a way that the output (digest) is unique and seemingly random, even for a small change in the input. A well-designed hash function should be deterministic, meaning the same input will always produce the same hash value. The choice of the hash function to use depends on the specific use and security requirements of your system, but to Digital signature purposes it is possible to use SHA family hash functions: **SHA-256**, SHA-384 and SHA-512, which produce digests of 256 bits, 384 bits and 512 bits respectively.

2.4 X.509 certificate

An X.509 certificate is a digital document that is used to bind a public key to an entity, such as a person, organization, or device. It is commonly used in the context of public-key infrastructure (PKI) to establish trust and security in online communication. X.509 certificates are widely used in SSL/TLS for secure website connections, digital signatures, and other security applications. It contains various information, including the public key, **issuer** name and serial number (the entity that issued the certificate), **subject** (the entity to whom the certificate is issued) name and serial number, validity period, digital signature, and other attributes. The certificate is digitally signed by the issuer's private key to ensure its authenticity.

2.4.1 X.509 certificate chain

In public-key infrastructure the authenticity of an X.509 certificate is provided through a certificate chain, that is a sequence of X.509 certificates that establishes a trust path from an end-entity certificate (e.g., device or user certificate) to a trusted root certificate authority (CA). There could be intermediate Certification authorities and each certificate in the chain is digitally signed by the issuer's private key, forming a hierarchical structure of trust.

2.4.2 X.509 certificate verification

To verify a digital certificate, two commonly used methods are Certificate Revocation Lists (CRLs) and Online Certificate Status Protocol (OCSP).

- **Certificate Revocation Lists (CRLs)**: CRLs are lists issued by the Certificate Authority (CA) containing the serial numbers of revoked certificates. To verify a certificate, the system checks if the certificate's serial number is present in the CRL. If it is, the certificate is considered revoked, and the verification fails.

- **Online Certificate Status Protocol (OCSP)**: OCSP is a real-time method to check the status of a certificate's revocation. When a certificate is presented, the system sends a request to the CA's OCSP responder to check if the certificate is valid. The responder sends back a response, indicating if the certificate is valid, revoked, or unknown.

For the verification of X.509 certificates, in our implementation the CRLs mechanism will be used.

2.5 Root of trust

The Root of Trust (RoT) is the foundational component in a security system that is inherently trusted.

- In the context of **generating cryptographic material**, the RoT is a secure element or process that provides a trusted environment for generating and storing cryptographic keys. It ensures that the keys are generated in a secure manner and protected from unauthorized access.
- In the context of **certificate chain generation and verification**, the Root of Trust refers to the trusted anchor in the chain of trust. It is the highest-level certificate authority (CA) that is inherently trusted by all parties involved.

The Root of trust is crucial since it ensures the integrity and trustworthiness of cryptographic operations and enables secure communication and identification within a system or network.

2.6 Trusted Execution Environment (TEE)

A Trusted Execution Environment (TEE) is a secure area of a main processor that provides **hardware-based isolation**. It helps code and data loaded inside it to be protected with respect to confidentiality and integrity. Data integrity prevents unauthorized entities from outside the TEE from altering data, while code integrity prevents code in the TEE from being replaced or modified by unauthorized entities. This secure environment can be used as a secure starting point (**root of trust**) for critical operations such as cryptographic key generation, so it can used to certify any keys or cryptographic material generated by exploiting it as a root. In this case it would play the role of the *issuer* of the digital certificate that certify the generated cryptographic material.

The most common TEE implementations in mobile devices are **ARM TrustZone**, Intel Software Guard Extensions (SGX), Qualcomm Trusted Execution Environment. Also, it is important to note that implementations of TEE can vary between different mobile devices and chip manufacturers. Some devices may use a combination of TEE technologies and architecture to provide enhanced security and isolation features. The latest android devices (smartphones and tablet) for example, use a **TEE with hierarchical architecture**, in which there are **two levels of TEE**, one more trusted than the other, capable of offering different security features.

2.7 Trusted Platform Module (TPM)

A Trusted Platform Module (TPM) is a hardware-based security component that provides a **secure environment for cryptographic operations** like cryptographic key generation and secure storing, secure boot, random number generation and hardware encryption. To access TPM capabilities it is possible to use APIs made available by the Operating System; thay may vary depending on the operating system and device platform. Usually the TPM consists of a dedicated chip, like in computers or laptops. However, in many cases, such as in smartphones not equipped with Hardware Backed security module, it translates into the presence of a **software module** that performs the operations described above. In the case of Android smartphones, this is the Android Keystore system.

2.8 Android Keystore system

The Android Keystore system lets you store cryptographic keys in a container to make them more difficult to extract from the device. Once keys are in the keystore, you can use them for cryptographic operations, with the key material remaining non-exportable. Also, the keystore system lets you restrict when and how keys can be used, such as requiring user authentication for key use or restricting keys to use only in certain cryptographic modes. In the KeyStore, the cryptographic material can be generated and manipulated leveraging two possible component:

- **TEE**
- **StrongBox Keymaster**: it is an enhancement to the Android Keystore system introduced in Android 9 (API level 28). It is designed to provide even higher levels of security for cryptographic keys and operations by leveraging **dedicated hardware security modules** known as StrongBox Keymaster HALs.

In any case, exploiting the TEE as root of trust to generate cryptographic material and performing operations always offers higher security features than using StrongBox. It is important to underline that in Android, whenever applications perform operations that require the use of a key, this is retrieved directly from the Keystore where it is securely stored.

CHAPTER 3

Implementation Overview

3.1 Starting point

It is needed to develop a methodology to prove the origin of some data, originating from a mobile device, to an external verifier. So, two apps have been developed:

- one that plays the role of the client i.e. the device that produces a data, signs and sends it to the server app.
- one that plays the role of the server that receives the data and must ascertain the client's identity performing some operations that are able to securely link the data received to the device that sent it.

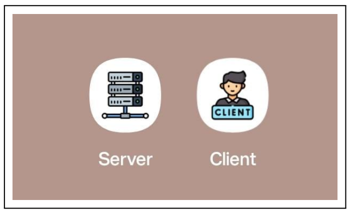


Figure 3.1: Icons of the two apps

The two apps are put in communication with each other through the use of TCP/IP Sockets in a local network context. The sockets endpoints are identified by an IP address and a port number.

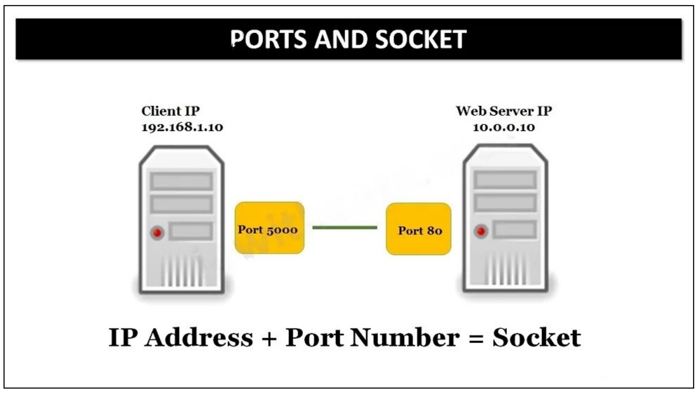


Figure 3.2: Socket is identified by its ip address and port number

3.2 Client-side

The client asks the Android Keystore to generate an asymmetric key pair, taking advantage of the methods it provides for the generation of a key pair. Since the Keystore uses the TEE as root of trust to perform its operation, the created key is automatically associated with an X.509 certificate whose issuer is the TEE of the device. This first certificate is signed by another one in a certificates chain of four that hierarchically sign each other until the fourth and last that is the self-signed Google CA certificate.

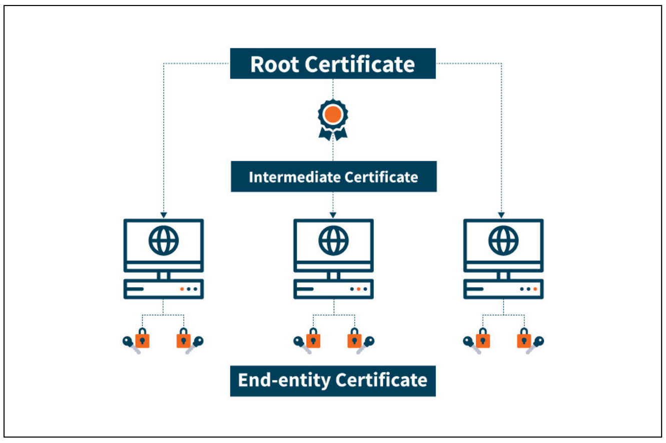


Figure 3.3: The certificate chian

It is fundamental to note that, in our solution, the use of the TEE serial number (TEE-SN) as issuer of the first certificate is the first step to uniquely associate a given data to the device that produced it, since the TEE-SN **uniquely identifies the device**.

At this point, the client takes care of producing data, computing a digest, encrypting it with its private key and sending it to the server as a digital signature object, together with its Certificate Chain. Finally, the client will wait for a feedback from the server, which will tell it if he was able to identify it and successfully complete the data-client binding operation.

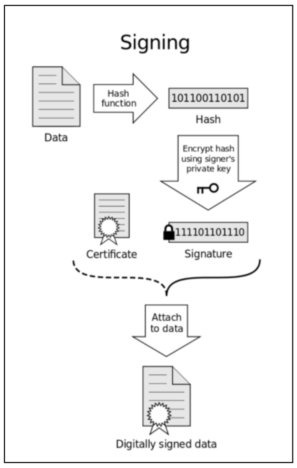


Figure 3.4: The client produces the data, processes a digest, encrypts it with its own private key and sends it to the server as a digital signature

3.3 Server-side

The first step of the server, after receiving the signature and the certificate chain from the client, is to check the validity of each certificate. To do so, for each certificate the following verifications are performed:

- verification of the validity date;
- verify that the certificate was successfully signed with the private key of the certificate at the upper level in the chain, until the Google self-signed certificate, for which its public key is also verified. Note that the Google public-key for these purposes is publicly available and therefore usable for verification
- online verification of the certificate status. The application makes an http request, which downloads the Google CRL at the official CRL distribution point. At this point, it can check the status of the certificate, i.e. whether it has been revoked/suspended or not.

The next step for the server is to extract the client's public key from the certificate chain, which is nothing more than the corresponding public key of the private key with which the client performed the signature. With the public key, the server decrypts the signature received. The result will then be compared with the data digest. If there is a match, the signature has been verified.

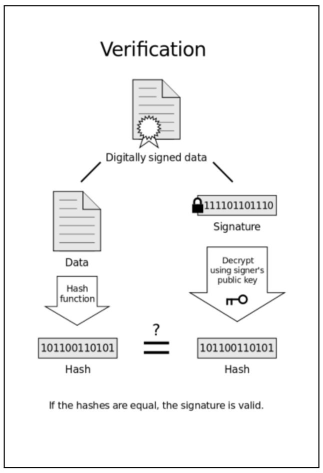


Figure 3.5: The server receives the signature and verifies it, calculating the digest on the data and decrypting the signature with the public key of the client

Then, the server verifies that the TEE serial number of the client device matches the expected one. To perform this check, the server compares the known TEE-SN to the TEE-SN that can be extracted from the first certificate in the chain sent by the client (the TEE is the issuer of that certificate). It's important to note that, this phase works under the assumption that the server already knows the TEE serial number information, which may be obtained, for instance, through prior physical access to the client device.

3.4 Final phase

Once these checks have been performed, the server will send back the results to the client which will display the outcome of the checks on the screen. So it is possible to understand if the data-sender binding verification was successful and, if not, distinguish which phase went wrong (certificate chain verification, digital signature verification, TEE-SN verification).

CHAPTER 4

Implementation Details

In this chapter, we will explore the implementation of our unique device identification technique in details. We will see into the technical aspects, the design choices and the key steps that made the creation of this system possible. From the use of the Trusted Execution Environment (TEE) and the Android Keystore system to the cryptographic mechanisms adopted, we will analyze the fundamental components of the project and how they work. We will also look at the challenges encountered during development and the strategies adopted to overcome them. This chapter is designed to provide a broad understanding of the inner workings of the project, preparing the reader to appreciate the complexity and effectiveness of the implemented solution.

4.1 Keystore

4.1.1 Keystore security features

The Android Keystore system protects key material from unauthorized use in two ways. First, it reduces the risk of unauthorized use of key material from outside the Android device by preventing the extraction of the key material from application processes and from the Android device as a whole; key material never enters the application process, and also key material can be **bound** to the secure hardware of the Android device, such as the Trusted Execution Environment (**TEE**). Second, the Keystore system reduces the risk of unauthorized use of key material within the Android device by making apps specify the authorized uses of their keys and then enforcing those restrictions outside of the apps' processes. Authorizations are then enforced by the Android Keystore whenever the key is used; this is an advanced security feature that is generally useful only if your requirements are that a compromise of your application process after key generation/import (but not before or during) can't lead to unauthorized uses of the key.

4.1.2 Useful API

Here are the Keystore System classes and methods we used, which can be found in the **java.security** library.

- **public class KeyStore**: this class represents a storage facility for cryptographic keys and certificates. A KeyStore manages different types of entry, each one identified by an "alias" string. In the case of private keys and their associated certificate chains, these strings distinguish among the different ways in which the entity may authenticate itself. For example, the entity may authenticate itself using different certificate authorities, like the key-pair we generate.

- **public static KeyStore getInstance(String type)**: returns a keystore object of the specified type. This method traverses the list of registered security Providers, starting with the most preferred Provider. A new KeyStore object encapsulating the KeyStoreSpi implementation from the first Provider that supports the specified type is returned.
- **public final void load(KeyStore.LoadStoreParameter param)**: loads this keystore using the given LoadStoreParameter. Note that if this KeyStore has already been loaded, it is reinitialized and loaded again from the given parameter.
- **java.security.cert.Certificate[] getCertificateChain(String alias)**: this method returns the certificate chain for the requested alias if it exists. The certificate chain represents the bound the key-pair has with the device since the first one, the one that certifies and signs the key-pair, is issued from the unique TEE of the smartphone.

4.2 Verifying key-pairs with Key Attestation

4.2.1 Key Attestation

Key attestation allows the server to verify that the requested key lives in secure hardware, such as the attestation signing key is protected by secure hardware like TEEs and signing is performed in the secure hardware. It also allows servers to verify that each use of the key is gated by user verification, preventing unauthorized uses of the key. In **Android**, the attestation statement is signed by an attestation key injected into the secure hardware (TEE) at the factory. Attestation statements are produced in the form of an X.509 certificate. Google provides the root CA and certifies attestation keys to each vendor.

4.2.2 Retrieve and verify a hardware-backed key pair

During key attestation, is possible to use the Android Keystore system specifying the alias of the key pair we want to attest. It, in return, provides a certificate chain, which can be used to verify the properties of that key pair. If the device supports hardware-level key attestation, the root certificate within this chain is signed using an attestation root key, which the device manufacturer injects into the device's hardware-backed Keystore at the factory.

4.2.3 Examples

To implement key attestation, complete the following steps:

- Use a KeyStore object's *getCertificateChain()* method to get a reference to the chain of X.509 certificates associated with the hardware-backed Keystore.
- Check each certificate's validity using an X509Certificate object's *checkValidity()* methods. Also verify that the root certificate is trustworthy with *verify()* method.
- Verify the status of each certificate (valid/revoked/suspended) downloading the Google CRL from the official CRL distribution point and looking fot the certificate serial number in the list.
- Also, it is possible to extract extension data from the X.509 certificate using a parser (e.g. ASN.1 parser) and compare them with the set of values that you expect the hardware-backed key to contain.

The following example consists of two certificates of the four in the certificate chain, since they are the most important to understand. The certificate chain is extracted from a sample device, and they are standard X.509 certificates with optional extensions. Certificate 0 is the certificate of the public key to attest that was generated in the sample device, whereas Certificate 3 is the root certificate that represent the Google CA that self signs. Certificates 1 and 2 represents respectively the second level of the device TEE that signs the first one (cert0) and the Google CA that signs the second level of the TEE (cert1).

Here are decoded X.509 certificates:

- **Certificate 0**:

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 1 (0x1)
    Signature Algorithm: ecdsa-with-SHA256
        Issuer:
            Title=TEE, serialNumber="c6047571d8f0d17c"
        Validity
            Not Before: Jan 1 00:00:00 1970 GMT
            Not After : Jan 19 03:14:07 2038 GMT
        Subject:
            commonName = Android Keystore Key
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (256 bit)
                pub:
                    04:21:01:97:84:c5:06:91:99:f7:f0:cc:33:ee:fd:
                    4a:4e:fd:e8:78:2f:b2:b1:6b:f4:bc:12:64:57:60:
                    fa:2c:80:e5:a0:aa:01:16:a4:c8:98:65:2e:64:48:
                    a0:91:43:8a:ce:4d:f4:0f:89:93:7a:b0:27:7e:66:
                    67:d9:69:aa:c2
                ASN1 OID: prime256v1
            X509v3 extensions:
    X509v3 Key Usage: critical
        Digital Signature
        1.3.6.1.4.1.11129.2.1.17:
    Signature Algorithm: ecdsa-with-SHA256
        30:45:02:20:0e:15:a8:83:3c:f2:9a:d9:a7:54:2f:d1:eb:de:
        f6:db:c9:61:28:07:46:d9:ce:f1:af:b7:d2:50:9e:da:de:84:
        02:21:00:9a:18:dc:a8:00:79:80:87:ac:23:f0:79:74:a5:46:
        47:26:45:2c:55:73:69:64:4a:e3:79:65:db:6e:53:9d:68
```

- **Certificate 3**:

```
Certificate:
Data:
    Version: 3 (0x2)
    Serial Number: 1 (0x1)
Signature Algorithm: ecdsa-with-SHA256
```

```
        Issuer:
            serialNumber="f92009e853b6b045"
        Validity
            Not Before: May 26 16:28:52 2016 GMT
            Not After : May 24 16:28:52 2026 GMT
        Subject:
            serialNumber="f92009e853b6b045"

        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (4096 bit)
                Modulus:
                    00:af:b6:c7:82:2b:b1:a7:01:ec:2b:b4:2e:8b:cc:
                    54:16:63:ab:ef:98:2f:32:c7:7f:75:31:03:0c:97:
                    [...]
                    96:43:ef:0f:4d:69:b6:42:00:51:fd:b9:30:49:67:
                    3e:36:95:05:80:d3:cd:f4:fb:d0:8b:c5:84:83:95:
                    26:00:63
                Exponent: 65537 (0x10001)

        X509v3 extensions:
            X509v3 Subject Key Identifier:
            36:61:E1:00:7C:88:05:09:51:8B:44:6C:47:FF:1A:4C:C9:EA:4F:12
                X509v3 Authority Key Identifier:
            keyid:36:61:E1:00:7C:88:05:09:51:8B:44:6C:47:FF:1A:4C:C9:EA

            X509v3 Basic Constraints: critical
                CA:TRUE
            X509v3 Key Usage: critical
                Digital Signature, Certificate Sign, CRL Sign
            X509v3 CRL Distribution Points:

                Full Name:
                    URI:https://android.googleapis.com/attestation/crl/

    Signature Algorithm: sha256WithRSAEncryption
        20:c8:c3:8d:4b:dc:a9:57:1b:46:8c:89:2f:ff:72:aa:c6:f8:
        44:a1:1d:41:a8:f0:73:6c:c3:7d:16:d6:42:6d:8e:7e:94:07:
        [...]
        06:06:9a:2f:8d:65:f8:40:e1:44:52:87:be:d8:77:ab:ae:24:
        e2:44:35:16:8d:55:3c:e4
```

Note that:

- The issuer of certificate 0 is the **TEE of the smartphone**, uniquely identified by its serial number. This is a fundamental information at the base of our attestation mechanism.
- Certificate 3 is the one of Google that, as CA, self signs itself. Among the certificate extensions we can find **CA:TRUE** and the official **CRL Distribution point**. The **serialNumber="f92009e853b6b045"** is publicly known to be the Google CA serial number for operation of this purpose.

4.3 Client-Server Architecture

In order to create a client-server architecture, an approach with TCP/IP socket is used, running within a local area network. The Server has its IP address and port hardcoded and on them it listens for incoming connections, whereas the Client has to manually insert the IP address and the port of the Server to which it wants to send the signed data for verification. To create the socket we have used the Java *Socket* class and the Java *ServerSocket* class with the following methods:

- **Server-side**: *new ServerSocket(serverPort)*: creates a server socket, bound to the specified port.
- **Client-side**: *new Socket(serverIp, serverPort)*: creates a stream socket and connects it to the specified Server IP address and port number

Communication through the socket takes place thanks to Java streams, that are sequence of data. An input stream, defined by the Java *InputStream* class, is used to read data from the source. An output stream, defined by the Java *OutputStream* class, is used to write data to the destination.
Futhermore, the *BufferedReader* and *PrintWriter* classes were also used for the communication with the following method. This classes, which are generated from OutputStream and InputStream objects, offer specific methods for writing and reading string, character or numeric data, ensuring that the data is sent and received in an appropriate format. It was decided to use different stream classes for communication because client and server exchange different types of data, some that are better to stay in bytes (for example: signature, certificate chain) and others that are more convenient to send typed (for example: the final results of the verification which are of type String).

4.4 Client Implementation

4.4.1 Key Pair Generation

First of all, we need to create an instance of the KeyStore using the *getInstance()* method of the *KeyStore* class, then using the *KeyPairGenerator* class we can initialize the builder method with all the parameters that we want to control, for example: **key alias**, key purpose (encryption, signature etc.), key padding, key length, etc. One of the most curious parameter is the *setUserAuthenticationRequired* which force the device to have done the user authentication through the screen unlock of the device, since a limited amount of time, otherwise no operation can be performed.
After that the key pair can finally be generated and automatically inserted inside the KeyStore system. When you want to extract the key-pair, you need to refer to it through the *getEntry()* method specifying the key alias. In this way it is possible to retrieve the securely stored entry and then also obtain the secret and public key with the methods *getPublicKey()* and *getPrivateKey()*.

4.4.2 Signature Generation

In order to generate the signature we need to extract the key pair from the KeyStore and distinct the private and the public key; compute the digest over the data to be signed, using an hash algorithm previously decided with the server (in our case we decided to use the SHA-256 hash algorithm), and the performing the signature, over the computed digest, using the *Signature* class and the Private Key previously generated.

4.4.3 Connection to the Server

At this point we send the signature value to the server. We can now extract the certificate chain associated to the public key of the key-pair using the Keystore method *getCertificateChain()* providing

the alias of the key. So we start sending the Certificate Chain, sending one certificate at a time preceded by its length, in order to tell the server how many bytes it have to read to separate the certificates.
After having sent all the information, the client waits for the responses from the server computation, and updates the graphics in order to display them.

4.5 Server Implementation

The Server creates the Socket object generating its endpoint and assigns its IP address and the port. It can now opens all the input and output streams, that allow it to receive and send data, and can starts waiting for incoming connections and sent data.
When the client connects, first data the server receives is the signature value, and after that, it start receiving the certificate chain of four certificates. For each certificate it receives also its length in order to know how many bytes read before the start of the next one.
After having received all the data, the Server has the duty to perform three kind of verifications: the Certificate chain validitation, the signature validation and the check on the TEE serial number of the sender. This last step is the fundamental check in our implementations, since it evaluates if the TEE serial number extracted from the first certificate of the chain (the one that certifies the public key of sender) is actually the one of the sender. In this way we can be certain that the data has been actually sent from that client.

It is important to note that the checks are done in the specified order given that first of all it is necessary to verify the authenticity of the public key by validating the certificate chain tracing it back to the legitimate sender; then, using the public key, it is possible to verify the received digital signature object; finally the most important check is carried out.
Let's now analyze in details the three verifications.

4.5.1 Certificate Chain Validation

This part is about verifying all the certificates in the certificate chain. It is important to remember that the certificate chain is automatically generated by the client Keystore system when the key-pait is generated. Since the used device has the possibility to generate Hardware-backed keys, the public key is signed by the specific TEE of the device.
In this phase, three kinds of checks are performed on the certificates.

- a first check on the certificate temporal validity, to understand if the certificate is still valid i.e. it is not expired. The X509Certificate *checkValidity()* method is used;
- the second check uses the X509Certificate *verify()* method to verify that the certificate was correctly signed using the private key of the certificate in the upper level in the certificate chain (parent certificate, starting from the last one of Google CA);
- the last check is about the status of the certificate. The server fetches the CRL from the Google official CRL distribution point, that is the Google API *https://android.googleapis.com/attestation/status.*
  It gets a list of revoked certificates and looks in it for the analyzed certificate by comparing its serial number to those in the list in order to check if it has been revoked or suspended.

After these three checks, if all four certificates are valid, the last step is to compare the Public Key of the root self-signed certificate (certificate 3 in the chain), with the hardcoded Public Key retrieved from the Google documentation: in this way we assure that it has been emitted from Google itself, and not from someone who claims to be the Google Certificate Authority.

4.5.2 Signature Validation

Server verifies the received Digital Signature object using the same hash algorithm of the client (in this case SHA-256). The computation of the signature is equal to the one performed by the Client, but in this case there is an additional step which check if the two digests are equal (integrity of data). In this step are used the hardcoded sample data and the Public Key extracted from the first certificate of the certificate chain.

4.5.3 TEE Serial Number Validation

This last check, for which it is assumed that the previous ones have been successful, is the one that really makes it possible to satisfy the requirement of the initial problem: univocally linking the sent data to the device that generated and sent it.
The Server extracts the TEE serial number from the first certificate in the chain (the one of the key-pair used) where it (the TEE) is addressed as the "Issuer of the Certificate", since it creates the certificate for the key pair used and "signs" the key pair. Then, the server compares it to a TEE serial number that it already known (in this case we hardcoded it). In practice, this presuppose that the server already knows the Client TEE serial number, maybe because in verification phase it can have physical access to the verified device. It is guaranteed that the extracted TEE serial number is the one of the creator of the key-pair because it is inside the certificate which integrity has been previously verified.

4.6 Issues and Possible Extensions

4.6.1 Client Side

- The data signed and transmitted are a fixed sample string hardcoded in the client: can be extended by inserting an input TextField in order to allow user to insert its own text. Although feasible, this solution would bring other security-related complications. Indeed, the string entered by the client must be securely transferred to the server, and this means that it should have the properties of integrity and confidentiality.

- Secure trasmission of the TEE serial number: advanced solutions could be adopted with which the client could be able to securely communicate its TEE serial number to the server, in the case that the verification could take place remotely and the server does not have physical access to the device under analysis. In this case the transmission of it should be totally secure and complications related to its confidentiality and integrity should be added.

4.6.2 Server Side

- The IP address on which the socket is opened is hardcoded in the code. It could better if it could be retrieved from the Network interface, avoiding the need to reload a new code when there is some network change.

- Data that have been signed client side, are hardcoded also in the server, so there is no need to transmit them over the network. When Client will be capable of sending arbitrary data, server code must be modified in order to consider also the of the reception of the signed data to verify.

- In the case the client sends his TEE serial number, the server should be modified in order to have the possibility to receive securely the serial number, and/or to store a list of devices that can ask for verification.

We deliberately decided to separate the three validation responses in Certificate Chain validation, signature validation and TEE serial number validation **in order to show which part may fail**. It is clear that if, for example, the Certificate chain validation fails for some reason, all the signature validation must fail, instead continuing to check the signature value and the TEE serial number correspondence. In this "error case" the final result of the verification should be negative.

CHAPTER 5

Results

In this project we have explored the problem of authenticating data coming from a device (a smartphone) using features unique to that specific device. So, the goal was to define a methodology to demonstrate to an external verifier that that data was generated on this specific device.

5.1 Strategy adopted

To obtain this type of verification, we took advantage of the presence of the TEE module in the device, in this case a smartphone, basing our work on the fact that it is uniquely identified by a serial number. The TEE (Trusted Execution Environment) is a secure and isolated environment within the smartphone where critical operations and sensitive data are processed and protected. It also provide a higher level of security and trustworthiness for various tasks, including cryptography, authentication, and key management.

Indeed, in case of generation of a key-pair on a smartphone, this component is capable of certifying the generated key-pair by signing the public key. This also involves the generation of a certificate chain whose role is to certify in a chain the various certificates involved, up to that of Google which, as a CA, self-certifies itself. So if it is possible to deliver the **signed** sample data and the certificate chain to the server that perform the verification, by first validating the certificate chain and having the guarantee that it is authentic, it is possible to extract the serial number of the TEE which on the sender has certified the public key, in order to make a comparison. This is possible since the TEE module appears in the first certificate of the chain as an issuer, and its serial number is also present. The extracted TEE serial number, as mentioned, is then compared with the actual TEE serial number of the sender device (already known by the server in some way), thus managing to bind the signed data received to the device that actually produced it.

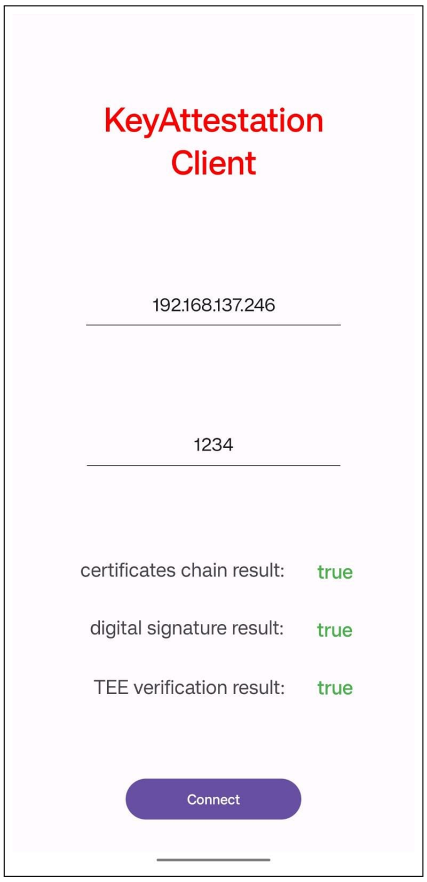
5.2 Use Cases

At the end of the verification, the server sends the client the outcome of the verification distinguishing three types of results: certificate chain verification result, digital signature verification result, TEE verification result.
We deliberately decided to separate the three validation responses so that it is possible to visualize, for educational purposes, which phase has failed. Clearly, if just one of them fails, the whole verification can be considered failed. So, if, for example, the Certificate chain validation fails for some reason, all the signature validation must fail, instead continuing to check the signature value and the TEE serial number correspondence. In this ”error case” the final result of the verification should be negative.

Instead, for the verification to be considered successful, all three results must be true. Let's see some screenshots to understand how the client and server application works.

5.3 Final verification successfull

After all the verifications are done and succesSsful, the following screen will be displayed in the client:



certificates chain result: true

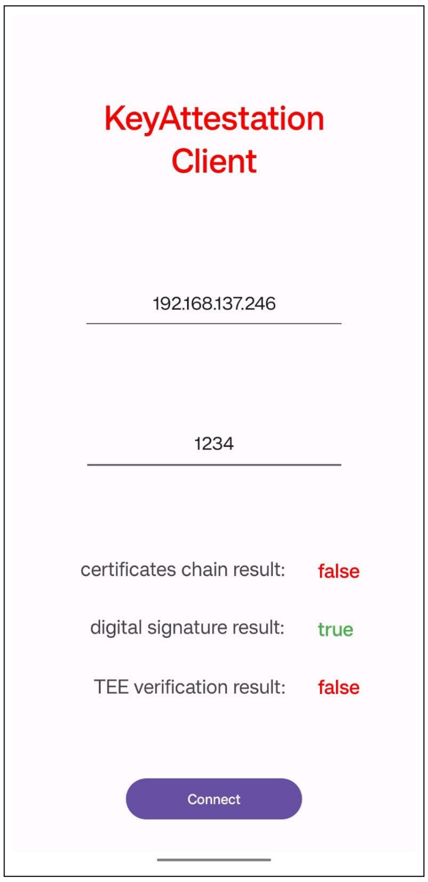
digital signature result: true

TEE verification result: true

Figure 5.1: Client has received some negative results from the verifications

5.4 Final verification failed

After all checks have been done and some have failed, the following screen will be displayed in the client:


KeyAttestation Client

192.168.137.246

1234

certificates chain result: false

digital signature result: true

TEE verification result: false

Figure 5.2: Client has received some negative results from the verifications

5.5 Known Issues

During the development of the project we faced a series of issues mainly related to security, design and communication between different apps.

5.5.1 Hardcoded data to sign

The data being signed and sent by the client is currently a static sample string that is hardcoded into the client's application. This inflexibility raises concerns, as it doesn't allow users to input their own text for signing.

While it is technically possible to address this by introducing an input TextField for user-provided text, doing so may introduce additional security challenges. Specifically, securely transferring the user-entered string to the server would become a priority, requiring measures to ensure data integrity and confidentiality during transmission. These improvements would inevitably lead to having to modify the server code in order to consider also the of the reception of the signed data to verify.

5.5.2 Secure trasmission of the TEE serial number

The TEE is manually loaded into the server to easily perform the final verification. This is not likely in possible real application of this technique.

Advanced solutions could be adopted with which the client could be able to securely communicate its TEE serial number to the server, in the case that the verification could take place remotely and the server does not have physical access to the device under analysis. In this case the transmission of it should be totally secure and complications related to its confidentiality and integrity should be added. Moreover, the server could store a list of authorized devices to verify, saving their TEE serial number maybe after the use of some service.

5.5.3 Harcoded IP address

The code presently has a fixed IP address for the socket.

A preferable approach would involve automatically fetching the IP address from the Network interface. This would ensure seamless adaptability to network changes, eliminating the need for manual code modifications or reloads. By dynamically retrieving the IP address, the application gains flexibility and user-friendliness, allowing it to function reliably in diverse network environments.

5.6 Future Work

The completion of this project marks a significant achievement, producing satisfactory outcomes. Nevertheless, it is crucial to recognize that there remain ample opportunities for further refinement and expansion.

5.6.1 Local network limit

Currently the verification system composed of two applications (client and server) only works if the devices are connected to local networks. In fact, the socket communication channel with which the two apps communicate, works with limited delays only if the client and server IP addresses are of the **192.168.x.x class**. If you try to use the applications with a public IP, such as when you are connected to the data network 3G/4G or to two different local networks, the client application fails to connect to server and crashes for a TIMEOUT exception.

In the future it is possible to investigate the problem and improve the communication system between the two apps in order to make it functional and resilient in various network conditions.

5.6.2 Google CRL fetch

The third check on the certificates in the certificate chain consists in making an HTTP request (GET) to the URL of the official Google CRL (Certificate Revocation list) distribution point
https://android.googleapis.com/attestation/status to check the status of the certificate (revoked/suspended/valid). The HTTP response contains a limited list of certificates/revoked keys of about fifty entries.

This very limited list is not likely to represent the true amount of revoked or suspended certificates; in fact, the official documentation says that the list is not an exhaustive list of all issued keys. For this reason, in the future it would be possible to carry out further investigations on how reliable this list is and possibly integrate the control with CRLs downloaded from other distribution points.

CHAPTER 6

Conclusions

The goal of this project is to define a methodology to prove to some external verifier that those data were generated on that specific device. We started by researching what could be some ways to uniquely identify the device. Reading the android documentation we found several unique codes that could be right for us because some of them were even directly related to the hardware, such as the IMEI, the serial number of the device or the MAC address. However, we realized that the Android documentation discouraged the use of these codes for identification purposes since it is not a good practice to use non-modifiable codes for these purposes. Furthermore, starting from the Android 11 version onwards (API level >= 11) these codes are no longer accessible by third-party applications since a particular permission is required which is granted only to system apps carrier apps (ISP applications). We also explored other types of codes (Android ID, Advertising ID) but they didn't fit as they weren't persistent (resettable after a factory reset) or changed after disinstalling and reinstalling the application.

Therefore, we opted to study aspects related to cryptographic operations of the device, such as the generation of keys, their signature and the digital certificates associated with them. We also discovered that it was possible to rely on the key attestation mechanism to verify the authenticity of key, which can also be hardware-backed thanks to the use of device security modules such as the **TEE (Trusted Execution Environment)** and the **Android Keystore Module**.

So we decided to leverage these tools to find a way to uniquely bind the key (key-pair) generated and used to sign a sample data to the device that generated it (which generated the data and therefore also the key-pair).

We have exploited the fact that when a key-pair is generated on an Android device, it is signed (certified) by the TEE in order to guarantee its authenticity. In turn, the TEE is certified in a chain by other entities up to Google which is a publicly trusted Certification Authority. This certificate chain, together with the digital signature of the data, can be sent to the verifier. During the verification phase, it is possible to extract from the first certificate of the chain information on the issuer, i.e. on the entity who certified the key-pair, which would be the TEE of the sender device. By extracting this serial number, during the verification phase, it is then possible to compare it with the actual TEE serial number of the sender device that the verifier already knows in some way. In this way it is possible to uniquely identify the sender and then successfully bind the data to the device that generated and sent it.

Bibliography

[1] https://developer.android.com/training/articles/user-data-ids

[2] https://developer.android.com/training/articles/security-key-attestation

[3] https://github.com/google/android-key-attestation/tree/master

[4] https://android-developers.googleblog.com/2019/09/trust-but-verify-attestation-with.html

[5] https://developer.android.com/training/articles/keystore

[6] White Paper: *Hardware-backed Keystore Authenticators* (HKA) on Android 8.0 or Later Mobile Devices

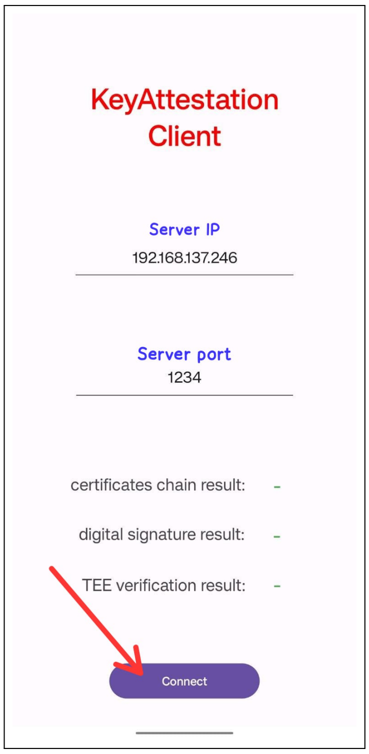
APPENDIX A

User Manual

A.1 How the applications work

A.1.1 Client

To start the client application you need to insert the correct IP address and port number of the server application. The server **must be** already active and waiting for connections (see next subsection). Now you can press the **connect button**, so your app will generate a data and send all the necessary information to the server that will perform the verification. Now the application waits for the server responses.



Server IP
192.168.137.246

Server port
1234

certificates chain result: -

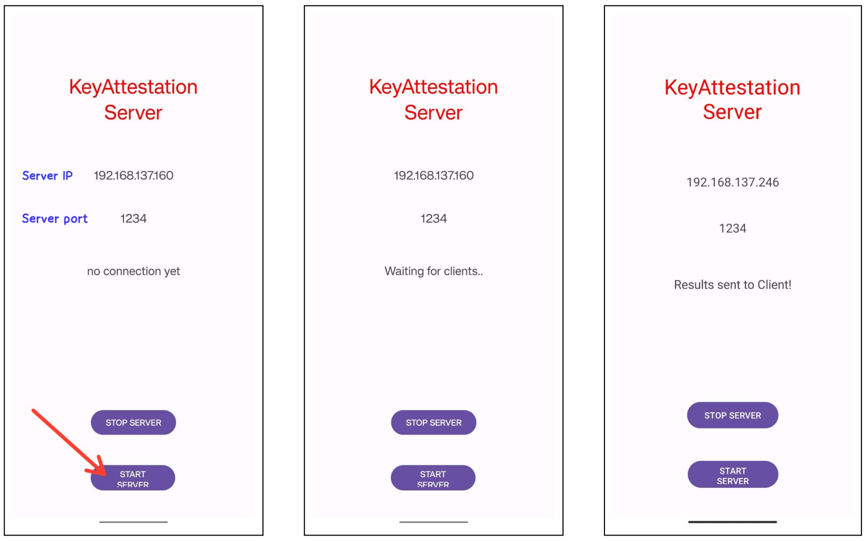
digital signature result: -

TEE verification result: -

Figure A.1: How to start the client

A.1.2 Server

To start the server application you need to press the start server button. Now the server is active and waits for connections from clients. After a client application connects, in background it will perform some computation and verification and then sends back the results to the client. You can see the success status displayed when results are sent back.



Server IP 192.168.137.160
Server port 1234
no connection yet

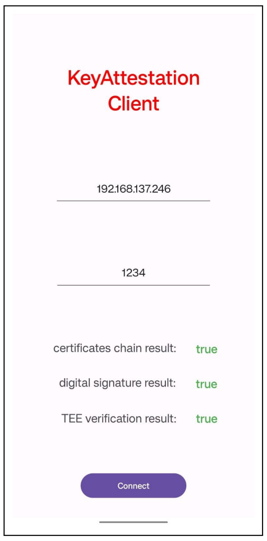
192.168.137.160
1234
Waiting for clients..

192.168.137.246
1234
Results sent to Client!

Figure A.2: Starting the server and receiving confirmation that data has been sent to the client

A.2 Final verification successfull

In case the verification is successfull, i.e. the certificate chain, the digital signature and the TEE serial number are verified correctly, the client will display the success status of each one of the verification.



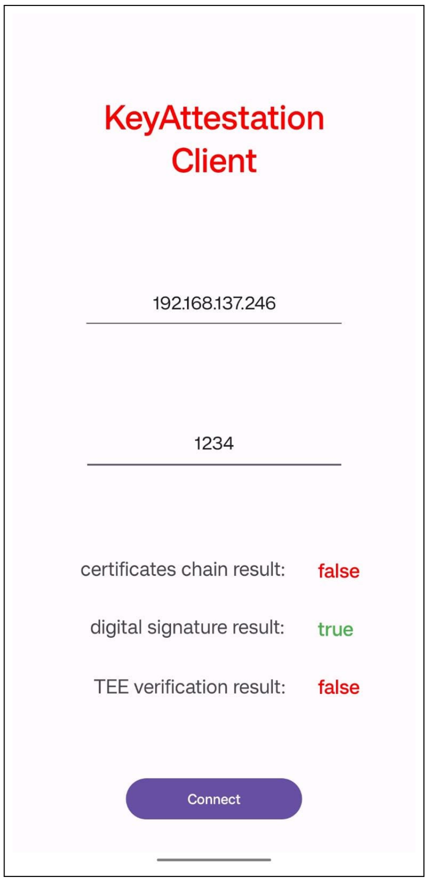
192.168.137.246
1234

certificates chain result: true
digital signature result: true
TEE verification result: true

Figure A.3: Client has received successful results from the verifications

A.3 Final verification failed

In case the verification fails, i.e. the certificate chain, the digital signature and the TEE serial number are not verified correctly (one of them or all), the client will display the fail status of verifications that have failed. Clearly, if just one of them fails, the whole verification can be considered failed.



KeyAttestation Client

192.168.137.246

1234

certificates chain result: false

digital signature result: true

TEE verification result: false

Figure A.4: Client has received some negative results from the verifications