Technische Universität München

Department of Informatics

Bachelor's Thesis in Informatics

A multi-tenant and privacy-preserving information processing system

Thomas Mauerer

Technische Universität München

Department of Informatics

Bachelor's Thesis in Informatics

A multi-tenant and privacy-preserving information processing system

Ein mandantenfähiges und Privatheit achtendes Informationsverarbeitungssystem

Author	Thomas Mauerer
Supervisor	Prof. Dr.-Ing. Georg Carle
Advisor	Marcel von Maltitz, M.Sc., Dr. Holger Kinkelin, Dipl.-Inf. Johann Schlamp
Date	January 15, 2016

Informatik VIII
Chair for Network Architectures and Services

I confirm that this thesis is my own work and I have documented all sources and material used.

Garching b. München, January 15, 2016

Signature

Abstract

In today's world smartphones have become indispensable. Most people use smartphones for surfing, chatting, email, photos, playing games, etc. However, another possible application is to use the available sensors (hardware and software) in order to measure several aspects. Collected data can be used for research studies or to compile statistics, for instance. For the process of collecting and governing sensor data of Android devices the *MeasrDroid* framework can be used. However, it is currently not possible to use the capabilities of *MeasrDroid* for individual situations, but only to support research by donating data.

This thesis is part of a bigger project which has the main goal to introduce multi-tenancy in *MeasrDroid* in a privacy-preserving way. Therefore, the overall structure and architecture of *MeasrDroid* has changed. Virtual machines are introduced as the individual data sink for each tenant which can be used in order to store collected data confidentially.

Apart from encryption, privacy preservation requires also an authenticated and integrity-checked connection for every data transmission. Therefore, the *MeasrDroid Pairing Protocol* has been introduced which enables the establishment of such a connection. This can either be achieved by the use of self-signed certificates or by the use of a PKI. The protocol needs to be performed before data is sent from one entity to another one and can therefore be seen as the initial pairing. Because of the protocol clients can also be allowed to directly push data to the VM where it is stored. This enables real-time data collection which has not been possible in the existing *MeasrDroid* project.

Apart from providing the theory to the *MeasrDroid Pairing Protocol* and the new architecture in general, this thesis includes also the implementation of the aforementioned protocol for the pairing between clients and VMs. Security plays a major role in the whole project because security is closely related with privacy which is the essential requirement. Therefore, the security model is investigated in detail in order to prove that privacy is obtained as best as possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Smartphones are omnipresent all over the world. Figure 1.1 shows a prediction of the number of people using a smartphone. At the time of this writing 7.32 billion people are living in the world [2]. This means that approximately 26 percent of all people in the world own a smartphone. According to figure 1.1 the number of smartphone users is continually increasing on a linear basis. This may have two main reasons: Firstly, because the number of all people living in the world is increasing, too. Secondly, because smartphones have a vast amount of capabilities and can be useful in a huge variety of situations. A smartphone is not just a device for telephony or text messaging [3]. However, it is more like a lightweight version of a desktop computer that can be taken everywhere and has become more and more powerful over time. Hence, developing applications for smartphones is a good idea in general because many people are addressed this way.

Depending on the age and the manufacturer, smartphones are usually equipped with a multitude of hardware sensors. This includes sensors for motion, acceleration or environmental properties, like temperature or pressure. These data sources can be complimented with software-based sensors, e.g. based on the wireless modules in order to measure network characteristics like roundtrip times or signal strength. Gathered data can then be exported in order to conduct research, compile statistics or to be used otherwise.

For the process of collecting and governing sensor data the *MeasrDroid* framework was developed at the chair of *Network Architectures and Services* which is explained in detail in section 2.2.

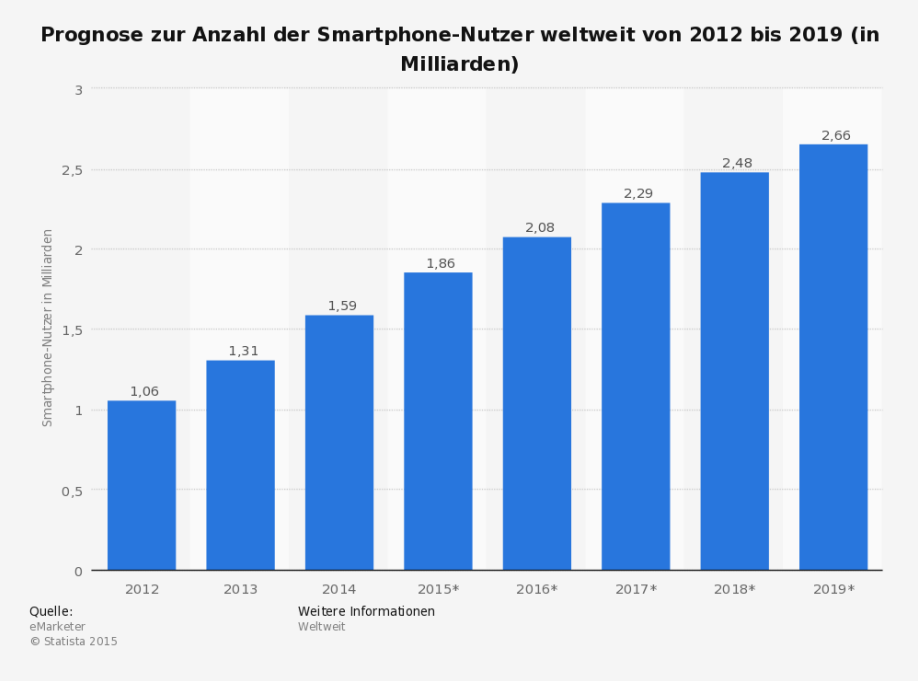


Figure 1.1: Number of Smartphone Users [1]

1.2 Problem

As explained in section 2.2 the main goal of *MeasrDroid* is to conduct network research. Users can therefore participate in the project by donating data. However, the individual user can not really take advantage of this because he has no access to the database where all measurements are stored. By implication, this means that individual users can currently not use the capabilities of *MeasrDroid* for their personal situations.

A second problem is that currently no real-time data collection is possible. This is due to the security model which hinders clients to directly push data to the server where it is stored. However, this reduces the application possibilities of *MeasrDroid* tremendously. One could think of using *MeasrDroid* as a communication infrastructure for a feedback system, for instance. In this case it may be problematic if feedback is not given instantaneously.

1.3 Goals and Research Questions

The main goal is to redesign *MeasrDroid* in order to offer the capabilities for individual situations. Therefore, multi-tenancy must be introduced in a privacy-preserving way. This basically means that the system is able to deal with individual tenants in the way that each tenant is the only party that has access to the measurements taken.

To obtain this goal the overall structure and architecture of *MeasrDroid* has changed which is explained in detail in chapter 3. The main innovation are individual data sinks for every tenant in the form of virtual machines with no third-party access. There are two situations where data is sent from one entity of *MeasrDroid* to another one: Firstly, when a smartphone uploads gathered data to the virtual machine. Secondly, when a tenant wants to share data with another tenant and therefore data is sent from one virtual machine to another one. Beside from having no third-party access to measured data, privacy preservation also requires an authenticated and integrity-checked connection in these two situations. For the establishment of such a connection a protocol has been introduced which manages the initial pairing between the two entities. This protocol is the main part of this thesis.

In the same context the security model is investigated. This is necessary because the current security model has changed due to the new structure and architecture of *MeasrDroid* (see chapter 3).

As a summary the research questions that will be answered in this thesis are as follows:

- **RQ1**: How can smartphones and virtual machines be paired to enable an authenticated and integrity-checked connection for uploading data?
- **RQ2**: How can two virtual machines be paired to enable an authenticated and integrity-checked connection for sharing data?
- **RQ3**: How can real-time data collection be enabled without undermining the common security concept?

This thesis also includes the implementation of the aforementioned pairing protocol in *MeasrDroid*.

1.4 Structure of this Document

Chapter 2 describes fundamentals that are needed in order to understand the main theory part of this thesis. As even being part of the title, privacy preservation is an important requirement. Therefore, chapter 2 starts with a definition of privacy preservation. Also, the *MeasrDroid* project is described as it is at the moment and aspects of cryptography are explained.

Chapter 3 introduces the theory part of this thesis. It starts with the description of the new structure of *MeasrDroid* and architecture. Afterwards the use cases are explained and the attacker model is defined which leads to the central aspect of this thesis: the *MeasrDroid Pairing Protocol*.

Chapter 4 describes two possible solutions of the aforementioned *MeasrDroid Pairing Protocol*. The first solution is based on self-signed certificates, whereas the second

solution uses a *Public Key Infrastructure* to achieve the same goals.

In chapter 5 the practical part of this thesis is documented. This is the implementation of the *MeasrDroid Pairing Protocol*.

Chapter 6 evaluates both the theoretical and practical part of this thesis under certain aspects. This chapter refers to the research questions and states how they are achieved. Also, a comparison of the two solutions described in chapter 4 is given here.

Chapters 7 and 8 finally introduce to related work, conclude this thesis and provide an outlook on further work that has to be done in order to have a privacy-preserving multi-tenant *MeasrDroid*.

Chapter 2

Background

This chapter provides the reader with some basics and background knowledge needed to follow the discussions in chapters 3 and 4. A larger part of this section are aspects of cryptography.

2.1 Definition: Privacy Preservation

According to [4] privacy is a term for which no consistent definitions exist. Prof. Dr. Lutz Prechelt tries to define it as the area in which a person can decide to whom, when and why information is accessible [4]. This can be justified by the so called *Golden Rule* (ethic of reciprocity) which prescribes to always treat others in the way one would also expect others to treat oneself [4]. As a general rule, this means that every kind of information related to a person has to be kept private until the particular person decides to share this information with others. In this thesis information basically means collected sensor data.

Privacy preservation then means methods to obtain privacy which can be achieved by mainly two possibilities: [4]

1. **Physical Blockade**: By taking appropriate measures of security it is not possible for an unauthorized person to get access to stored data. Furthermore, *Physical Blockade* means secrecy of sensitive data which is a crucial aspect of many security concepts.

2. **Encryption**: Data is never sent or stored in plain format but always encrypted and can only be decrypted by the authorized person.

This definition is subject of the overall new concept of *MeasrDroid*. Derived from the two aspects, privacy preservation requires also the authentication of communication

partners and verification of the integrity of transmitted messages. This confirms that messages are sent only to valid entities and are not manipulated during the transport.

2.2 MeasrDroid Project

This section describes the existing *MeasrDroid* project as it is at the moment and also states why changes are necessary in order to fulfil the research questions mentioned in 1.3.

MeasrDroid is an Android framework for collecting and governing sensor data of Android devices [5]. The project was started in March 2011 by members of the chair of *Network Architectures and Services* and more than 10 students [6]. The lead developer is Dipl.-Inf. Johann Schlamp who is also advisor of this thesis. The main goal is to run network research [6]. *MeasrDroid* consists of a client (see 2.2.1) which is installed on the device in order to measure sensor data and a backend infrastructure for storing collected data (see 2.2.2).

2.2.1 MeasrDroid Core

The *MeasrDroid Core* is an Android application that has to be installed on the device. It is responsible for measuring sensor data. Collected data is always encrypted on the device before it is pushed to the backend server. At the first start a wizard opens which lets the user configure several aspects, like how much data he wants to donate per day or which sensors should be activated or deactivated [6]. The core itself does not consist of any activities. However, every Android application that wants to use the features of *MeasrDroid* only has to include the core into the project, let the main class inherit from the core and mark it with the special *@MeasrDroidSetup* tag. The most famous Android application that uses *MeasrDroid* is the *MeasrDroid Application* which is available on the *Google Play Store* for free. [7] It lets the user visualize all taken measurements.

2.2.2 Backend Infrastructure

The aim of the *MeasrDroid* backend is to store collected data. The data can be made available for research projects later on. An overview of the backend components is given in figure 2.1.

The *MeasrDroid* backend consists of the following components:

- **C3PO:** The *C3PO* is the main server of *MeasrDroid*. It is responsible for decrypting collected data and storing it in the database. The *C3PO* itself is not connected to

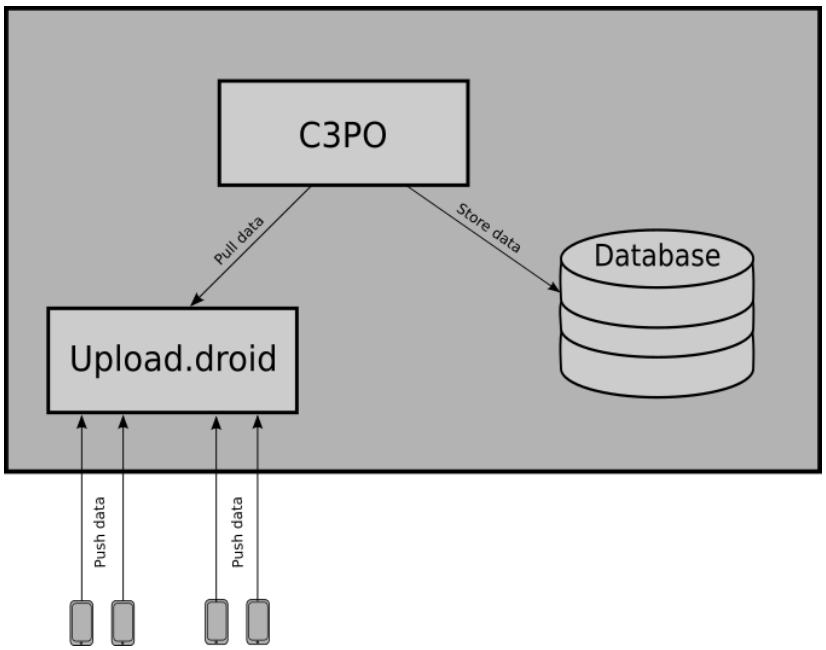


Figure 2.1: Overview of the Backend Components

the internet, due to security reasons. Therefore, it is not possible to directly push data to the *C3PO*.

- **Upload.droid:** The *Upload.droid* is the server where clients push their measured data to. The *Upload.droid* is therefore accessible on the Internet. The *C3PO* periodically pulls the temporary stored data from the *Upload.droid* and stores it in the database.
- **Database:** All taken measurements are stored in a centralized database which is only accessible by the *C3PO*.

2.2.3 Necessity of Changes

In general there are two aspects in the backend infrastructure which make it impossible to fulfil the research questions (see 1.3) and general goals of this thesis without any changes:

1. **Centralized database**: A single database for all taken measurements is not compatible with a privacy-preserving multi-tenancy concept. That is because the operator of *MeasrDroid* can learn all data collected from all clients and tenants, respectively. Instead, this should only be possible for the particular tenant.
2. **Polling**: Storing collected data in the database is currently not event-based, but done by polling. This means that the *C3PO* periodically asks the *Upload.droid*

whether data is available. If this is the case, data gets pulled by the *C3PO* and is stored in the database. However, the time steps between two polls can not be made arbitrary small because of two main reasons: Firstly, due to the amount of processed data, depending on how many Android devices participate. Secondly, due to performance reasons. In general, it is not recommended to set the time step as small as possible because this means that work is often in vain when no data is available. However, this makes real-time data collection impossible.

2.3 Cryptography

This section describes basics of cryptography and protocols that are used in the *Measr-Droid Pairing Protocol* described in section 3.5.

2.3.1 Public-Key Encryption

Public-key encryption was invented in the mid-70s by M. Hellman, W. Diffie and R. Merkle. The idea behind public-key encryption is that every participant has a key pair consisting of a public key K_E and a private key K_D. The public key can be made publicly available, for example on a website, whereas the private key always has to be kept private. Data is then encrypted with somebody's public key and can only be decrypted with the dedicated private key. The advantage of public-key encryption is that there is no need for a shared secret between sender and receiver. The most famous public-key encryption scheme is *RSA* [8] invented by R. Rivest, A. Shamir and L. Adleman. [9]

The most important properties of a public-key cryptosystem are as follows (a detailed definition can be found in [9]):

1. (K_E, K_D) can be created efficiently and there is a relation between K_E and K_D.
2. $\forall m \in M : D(E(m, K_E), K_D) = m$, where M is the set of all plaintexts, E is the encryption function and D is the decryption function.

The security in public-key encryption lies in the fact that there is a relation f between the public key K_E and the private key K_D but the private key can not be calculated efficiently by just knowing the dedicated public key. That is because f is a so called *one way function*: [9]

$$f : X \longrightarrow Y$$

1. $\forall x \in X : f(x)$ can be calculated efficiently.
2. $x = f^{-1}(y)$ can not be calculated efficiently.

In general, it is not proven that *one way functions* exist at all [9]. However, one assumes that every function is suitable if no efficient algorithms are known to revert the function.

An example is the multiplication of two big primes p and q, so that $n = p \times q$. This is quite easy to achieve. However, to revert this, one would have to solve the mathematical problem of factorizing a number n in its primes p and q, for which no efficient algorithms are known if n is big enough. This is used in *RSA*, for instance. [9]

Encryption and decryption is also based on a *one way function*. However, this would mean that even an authorized person is not able to decrypt a message efficiently. Therefore, a special form of a *one way function* is applied in this context, a so called *trapdoor one way function*. The definition is nearly the same, only aspect 2. changes to the following: [9]

2. $x = f^{-1}(y)$ can only be calculated efficiently with the knowledge of an additional information.

The additional information is the trapdoor which is the private key [9].

2.3.2 Message Authentication Code

A cryptographic hash function H is a non-injective function that maps every word $x \in X$ of arbitrary size to a value $H(x) \in Y$ of a fixed size k

$$H : X^* \longrightarrow Y^k$$

and also has the following properties: [9]

1. H is a *one way function*:
   - $\forall x \in X : H(x)$ can be calculated efficiently.
   - $x = H^{-1}(y)$ can not be calculated efficiently.
2. Given $x \in X$ and the dedicated hash value $h = H(x)$. It is not efficiently possible to find $x' \in X$ with $x \neq x'$ that produces the same hash value, $H(x) = H(x') = h$.

These two properties define a cryptographic hash function with weak collision resistance. For a cryptographic hash function with strong collision resistance also the following property must be fulfilled: [9]

3. It is not efficiently possible to find two different words $x, x' \in X$ that produce the same hash value, $H(x) = H(x')$.

Cryptographic hash functions can be used to verify the integrity of a message m, for instance. The calculated hash value $h = H(m)$ is a fingerprint for m. Any transmission errors can be uncovered by re-calculating the hash value of the received message m'. If the re-calculated hash value $h' = H(m')$ does not equal h, there were changes to m during the transmission. That is because any change to m leads to a completely different hash value due to the aforementioned properties. However, one can not use

cryptographic hash functions to protect against deliberate manipulation. That is because an attacker can also re-calculate the hash value for the manipulated message $h' = H(m')$ and replace the original h by h'.

The most famous and widely used cryptographic hash functions are the *SHA-#*-functions (where # means 1,2 or 3) [10] and the *MD5* [11]. These are so-called *dedicated hash functions* [9].

Sometimes it is also necessary to authenticate the sender of a message beside from having an integrity-protection. This is where MACs come into play: A MAC is a cryptographic hash function combined with a secret key K only known to the sender and receiver of a message. A usual communication between Alice and Bob looks as follows: [9]

1. Alice calculates a MAC for a message m with the key K.

$$MAC(m, K) = mac$$

2. Alice sends both, the message m and the value mac to Bob.

3. Bob verifies the integrity of the received message m^* by calculating the MAC of m^* and comparing it with the received value mac.

$$MAC(m^*, K) = mac^* \stackrel{?}{=} mac = MAC(m, K)$$

Usually the key K is concatenated to the message m, so $m' = m||K$ or $m' = K||m$. This means that calculating the MAC of a message m is nothing more than performing a cryptographic hash function H on m':

$$MAC(m, K) = H(m')$$

Depending on the taken hash function, this can be problematic in the way that an attacker can modify the message m and also calculate a valid MAC without even knowing the secret key. He just has to concatenate his message m'' with m' and perform the cryptographic hash function. To prevent this attack one should always use the HMAC procedure in order to create a MAC. [9]

HMAC is a procedure that uses the hash function H two times. The key K is not only concatenated with the message m but also with the result of the first run of the hash function:

$$HMAC(m, K) = H(K||(H(K||m)))$$

Using a hash function two times, increases collision resistance significantly, too. Until today, the HMAC procedure is considered to be secure. [9]

2.3.3 Public Key Infrastructure

As described in section 2.3.1 in public-key cryptography every participant has a key pair consisting of a public and a private key. Without any further means several problems may occur: [12]

1. **Authenticity of public keys:** If Alice wants to send an encrypted message to Bob, she takes Bob's public key for encryption. However, when Bob's public key is spread on the Internet, Alice can not be sure that the key really belongs to Bob. An attacker may have replaced Bob's key with his own key without being recognized by Alice.

2. **Revocation of public keys:** If Bobs's private key got lost or stolen, it is not secure anymore to use this key pair for encryption. The key pair should therefore be revoked and replaced by a new key pair. However, there are no ways for Alice to recognize that Bob's public key is not valid anymore.

3. **Non-repudiation:** If Alice signs a message with her private key, it should not be possible to deny this signature. However, this is quite easy to achieve by just saying that Alice's public key does not belong to her.

The solution for the mentioned problems are digital certificates. A certificate is a signed piece of data that maps somebody's identity to the public key [12]. The common format for certificates is the *X.509* format described in table 2.1.

Field	**Explanation**
Version	The version field describes the version number of X.509. The common version is 3.
Serial Number	The serial number is a unique id for a signing party to unambiguously recognize the signed certificate.
Signature	The signature field describes the algorithm and additional parameters used for signing the certificate.
Issuer	The issuer is the name of the signing party. This has to be a *X.500* conformable name.
Validity	The validity field describes the start and end date for the certificate to be valid.
Subject	The subject is the name of the owner of the certificate. This has to be a *X.500* conformable name.
Public Key	This is the public key that belongs to the owner of the certificate.

Table 2.1: Format of X.509

A certificate can either be self-signed or signed by an independent, so called *Certificate Authority* (CA). Self-signed means that the subject is equal to the issuer.

In a common scenario there are multiple CAs and they are ordered hierarchically. This means that there is one root CA who has a self-signed certificate which signs certificates for underlying CAs. Those CAs may sign certificates for other participants in turn, for example for Alice and Bob or other CAs. This ends up in a certificate chain. The idea behind this is: if one trusts the root CA, one can also trust everybody else in the chain. The entirety of all participating parties is called a PKI. [12] An example PKI is illustrated in figure 2.2.

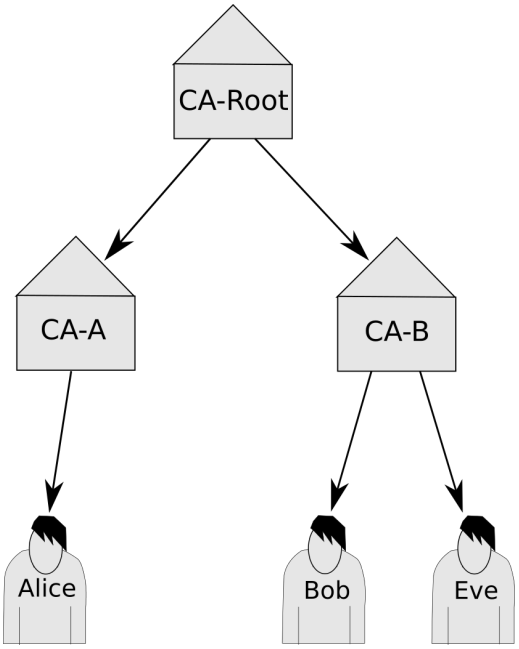


Figure 2.2: Hierarchical PKI

The example shown in figure 2.2 looks as follows: If Alice wants to send an encrypted message to Bob, she first verifies Bob's certificate. Bob's certificate is signed by CA-B which is in turn signed by CA-Root. Due to the fact that Alice trusts CA-Root, she can be sure that Bob's public key really belongs to Bob.

The solution for problem 2 is a *Certificate Revocation List* (CRL). This is simply a list provided by a CA which contains all serial numbers of revoked certificates. If a certificate is part of this list, it should not be trusted anymore. [12]

The creation of a certificate is usually split into two parts. At first, a so called *Certificate Signing Request* (CSR) is created. This contains the subject and the public key. The CSR is then sent to the signing party (e.g. CA) which creates the actual certificate and signs it in the second step.

2.3.4 SSL/TLS

SSL (Secure Sockets Layer) and TLS (Transport Layer Security) are protocols located above the transport layer in the *OSI model* [9]. All aforementioned aspects of cryptography, like *Public-Key Encryption, Message Authentication Codes* and *Digital Certificates* are part of these protocols. SSL was invented by *Netscape* with the goal of secure HTTP connections. This is called HTTPS and means simply a secure SSL channel which can be used to transfer HTTP data [9]. However, SSL can not only be used for HTTP but a huge variety of protocols. An overview of these protocols can be found in [9]. TLS is the improvement of SSL and is currently the international industry standard. However, the core aspects of TLS and SSL are the same [9].

The tasks of SSL/TLS can basically be divided into three parts: [9]

1. **Authentication of Communication Partner**: This is achieved by the use of public-key encryption and digital certificates. The idea behind this is that a sender can be sure that he is connected to the desired receiver.
2. **Confidential end-to-end transmission**: This is achieved by the use of encryption using a session key that is only known to the sender and the receiver.
3. **Verification of Integrity**: This is achieved by the use of *Message Authentication Codes*. TLS supports only HMAC which is considered to be secure.

It is not predefined what algorithms of public-key encryption, symmetric encryption or MACs should be used. This is negotiated in every connection. The procedure of exchanging all necessary information and key material is not a trivial task. Hence, this is not further explained in this thesis but can be found in detail in [9].

The security of SSL/TLS lies mainly in the used algorithms for cryptography. If only secure algorithms are used with recommended key lengths, the protocol is considered to be secure. [9]

However, one known security leak is the management of digital certificates. If an attacker manages to get a valid certificate signed by a globally verifiable, trusted CA, he can redirect a client to a malicious server which authenticates itself with the valid certificate. In this case a client can not recognize that he is not connected to the desired server but to the malicious one. [9]

Chapter 3

Analysis

This chapter is the main theory part of this thesis. The reader gets information about the new structure, as well as an overview of the new architecture of *MeasrDroid*. Also, the privacy model is defined in this chapter. This basically means at which layer we are going to obtain privacy preservation. The thoughts on this are required, due to the new structure. Apart from that, use cases and the attacker model are presented which lead to the central aspect of this thesis: the *MeasrDroid Pairing Protocol*.

3.1 Structure and Privacy Model

A multi-tenancy concept requires a new structuring of *MeasrDroid*. Prior, every person that participated in the project was an individual user of *MeasrDroid* per definition. Therefore, it was only necessary to install the application on the client device (smartphone) and start donating data. This changes in the new concept. Figure 3.1 shows the new structure of *MeasrDroid*.

According to figure 3.1 a completely new layer is introduced: the layer of tenants. Per definition, every tenant is an individual user of *MeasrDroid* now and every tenant can have an arbitrary number of clients as data sources. Each client belongs to exactly one tenant. A client is a smartphone, whereas a tenant is meant to be a person or even a group of people. In this case there is no distinction between the individual persons of the group. They are all on an equal footing.

Privacy preservation is only applied to the layer of tenants. This means that collected data from one of the clients is only accessible by the respective tenant. By implication, this means that there is no privacy preservation at the layer of clients. When a client participates as a data source for a tenant, he has to accept that the respective tenant has access to all of his collected data.

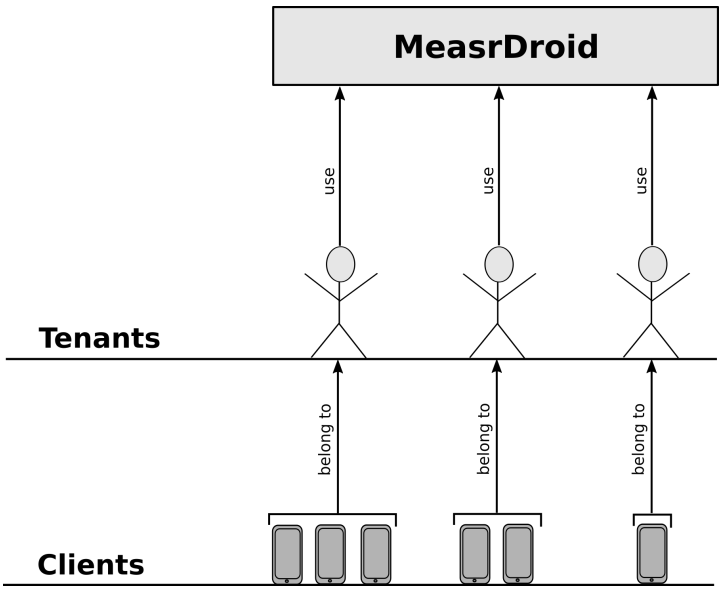


Figure 3.1: Structure of MeasrDroid

However, this definition is acceptable because we assume that all clients that belong to one tenant have the same interests. If one client does not accept this decision, he can still use the capabilities of *MeasrDroid* by becoming his own tenant.

3.2 Overall Architecture

In comparison to the architecture described in section 2.2.2 several aspects changed in the new architecture of *MeasrDroid* in order to fulfil the research questions and satisfy the new structure explained in section 3.1. Figure 3.2 illustrates the new architecture. All components are described in the following subsections.

3.2.1 Individual Data Sinks

In general, multi-tenancy can be achieved by different ways. Due to privacy preservation as the central requirement of this work, it is necessary to separate data - owned by individual tenants - from each other. One possible solution are individual data sinks for every tenant with no third-party access. This can be reached by the use of containers. Here, a container simply means an isolated environment that offers the possibility of storing data. On a technical point of view it does not matter whether the containers are isolated by physical aspects (e.g. different servers) or because of special software. One possibility are virtual machines (VMs) in order to reach the goal of isolation. In this thesis we are always talking about VMs.

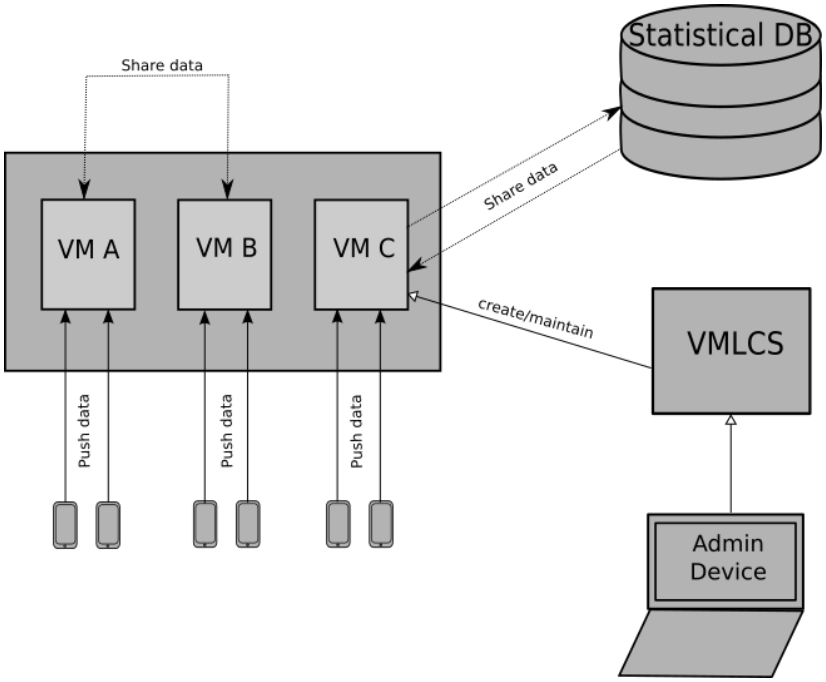


Figure 3.2: Overall Architecture

Each tenant is assigned a VM which he can use to store collected data. In general, two interfaces exist on all VMs for the exchange of data: One in order to receive data from clients and one in order to share data with other VMs. Both interfaces demand the presence of a secure and authenticated connection, due to the requirement of privacy preservation: In the first case between a client and a VM, in the second case between two VMs. In chapter 4 a protocol is presented that enables the establishment of such a connection.

Furthermore, privacy preservation requires that data is never sent or stored in plain-format, but always encrypted. This ensures that reading some tenant's private data is not possible at any time - and even not for the provider of *MeasrDroid* who may have access to the VM because of administrative tasks.

In figure 3.2 three VMs are illustrated exemplary with two clients, respectively.

3.2.2 Statistical Database

The statistical database is kind of a remnant of the centralized database of the old architecture. Its main purpose is to offer the original objectives of *MeasrDroid*, e.g. network research if tenants are willing to donate data. Also, it enables an easy possibility of compiling statistics because it is a central database with which all tenants can share data. On a technical point of view the statistical database is nothing more than a VM

apart from not being assigned to any tenant. Therefore, the same interface for sharing data between two VMs can be used for sharing and receiving data from the database.

3.2.3 Trusted Tenant Device

The *Admin Device* is the tenant's device that has direct access to the particular VM in the form of a TLS-protected connection. It is meant to be the control center of *MeasrDroid* for the respective tenant. It is necessary in order to register a new client or to inspect and evaluate collected data. This is only possible on the *Admin Device* because the private key, which is needed for decryption, is stored here. Apart from that, a tenant can select parts of collected data that he wants to share with others on the *Admin Device*. Due to the fact that the *Admin Device* has direct access to the VM, it must be secure and trusted which it is per definition. We assume that only the respective tenant has access to the *Admin Device*.

3.2.4 Virtual Machine Management

The *VMLCS* (Virtual Machine Lifecycle Service) as it is called in figure 3.2 is a service that is responsible for the creation, maintenance and deletion of VMs and the assignment to tenants. The *VMLCS* is part of a different work and therefore it is not investigated any further in this thesis.

3.3 Use Cases

Based on the new structure and architecture of *MeasrDroid* several use cases are conceivable.

3.3.1 Measurement of a Private Area

In this use case all capabilities of the original *MeasrDroid* are used for somebody's personal purposes. A tenant, which can also be a group of people in this use case (maybe a company), wants to measure a private area. However, collected data should never be accessible by anybody apart from the tenant. Therefore, a VM is rented to which all clients upload their measured data. Data is stored confidentially on the VM and can only be inspected and evaluated on the particular *Admin Device* which is the entity that has direct access to the VM.

3.3.2 Support Research

This use case is the original goal of *MeasrDroid*. Due to the fact that a centralized database still exists, which is basically a VM managed by the service provider, supporting research by donating data is still possible. Only the dataflow of collected data changes. A tenant rents a VM and starts measuring data with his smartphone. The collected data is uploaded to the VM where it is stored confidentially. The tenant can then inspect the collected data on his *Admin Device* and select the parts he wants to donate and therefore share with the centralized database.

3.3.3 Comparison of Measurements

This use case is the situation where two persons want to compare several aspects of their private area. Therefore, both rent a particular VM, measure their private area and upload the collected data to their VMs where it is stored confidentially. Data must then be shared with each other in order to enable a comparison. Collected data is transferred from one VM to the other one. However, if one person wants to keep some of the collected data confidential, he can select only parts before sharing. The selection can be done on the *Admin Device* where data is inspected.

3.4 Attacker Model

Privacy as it is defined in section 2.1 mainly depends on security. Hence, it is necessary to define the attacker model in order to figure out which means of security are required in order to achieve the goal of privacy preservation. According to section 3.3 there are mainly three forms of data flow in the new concept of *MeasrDroid*.

1. The data flow from a client to a VM.
2. The data flow from a VM to the particular *Admin Device*.
3. The data flow from one VM to another one.

Privacy preservation requires the presence of means of security in the situations when data is sent from one entity to another one. In the following only the aspects 1 and 3 are taken into the definition of the attacker model because aspect 2 is part of a different work.

A1: Steal Private Data
The first kind of attacker tries to gain insights into some tenant's private data. Basically, there are three possible targets to perform this attack:

1. **Access to VM**: All collected data is stored on the VM. If the attacker manages to get access to the VM, he has also access to all of the data. Therefore, it is required to have common security aspects in place like *Access-Control* mechanisms, *Intrusion Detection Systems, Firewalls*, etc. which make it hard to ever get access to the VM. Apart from that, it is necessary to store data only in encrypted format. In this case an attacker has no chance of stealing private data even if he manages to get access to the VM.

2. **Access to Client Device**: The client is the device that is responsible for measuring data. If an attacker manages to get access to the client device, he has also access to the measurements collected from the specific client. Therefore, it is required to not store any data on the client device but only upload it to the VM. If the VM is not reachable at any moment, the data needs to be stored temporary on the client device and uploaded later. In this case it is necessary to encrypt the data before it is stored.

3. **Intercepting the Transmission/Spoofing**: If the attacker manages to intercept the transmission of some data, he can get access to the sent data. Therefore, it is required to never send data in plain format but always encrypted. This means that both a client and a VM must always encrypt data before it is sent. Apart from that, an attacker could also redirect the transmission in order to let the client or the VM send their data to a malicious VM. Therefore, it is required to authenticate the receiver and have a confidential end-to-end data transmission.

A2: Distort Data
The second kind of attacker tries to distort collected data by foisting senseless data on some tenant. This would be possible if the VM accepts and stores incoming data from all clients or VMs, respectively. Therefore, it is required to authenticate the sender of a message and only establish a connection if the sender is valid.

A3: General Attacks
The third aspect encompasses all general attacks against an IT system, in this case basically against a VM. This concludes cripples of a whole system, viruses, worms, etc. To prevent this kind of attacks, common security aspects like *Intrusion Detection Systems, Firewalls*, etc. are necessary.

3.5 MeasrDroid Pairing Protocol

We assume there are two entities of *MeasrDroid*, called *A* and *B*. *A* can either be a client or a VM, whereas *B* is always a VM. According to the use cases (3.3) and the attacker model (3.4) there are situations when data is sent from *A* to *B*. In these situations it is required to have an authenticated and integrity-checked connection for the transmission

of collected data in order to prevent attacks of the type **A1** and **A2**.

This can both be achieved by a TLS-protected connection with mutual authentication. Therefore, it is necessary to exchange cryptographic material that can be used in order to establish such a connection. The *MeasrDroid Pairing Protocol* has been developed to reach exactly this goal. The protocol does not depend on which kind of data should be transmitted. As a side effect, it enables real-time data collection which is also part of the research questions. This is explained in section 6.5. The protocol is used in two situations:

1. **Upload Data to VM**: In this case the protocol must be performed between a client and a VM. After the point when a client has performed the protocol successfully, he is a valid client and can therefore always establish the TLS-protected connection with the VM in order to upload data. Due to this aspect, the *MeasrDroid Pairing Protocol* can also be seen as the registration of a new client for an existing VM.
2. **Share Data**: In this case the protocol must be performed between two VMs. After the point when a VM has performed the protocol successfully, the particular tenant can always establish the TLS-protected connection with the second VM in order to share data.

Chapter 4 describes two different possible solutions for the protocol.

Chapter 4

Design

This chapter describes two possible solutions for the *MeasrDroid Pairing Protocol* introduced in section 3.5. The first solution is based on self-signed certificates, whereas the second solution is based on a PKI. Each solution is split into two parts: the pairing between a client and a VM and the pairing between two VMs. In section 6.4 the security model of both solutions is investigated to prove that both solutions are secure.

4.1 Pairing based on Self-Signed Certificates

This solution is based on self-signed certificates. The goal is to exchange self-signed certificates and mark them as trusted. This enables the establishment of a TLS-protected connection for sending/sharing sensor data later on. That is because both the client and the VM or two VMs can use the self-signed certificates for authentication then.

4.1.1 Client ↔ VM

Participating entities are the VM, the client and the *Admin Device*. In this solution, both the client and the VM generate self-signed certificates and exchange them. At the end of the protocol the exchanged certificates are stored and marked as trusted.

Problematic is that there is no secured connection between the VM and the client at the start of the protocol. Hence, sending the certificates via a non-confidential channel is vulnerable for *Man-in-the-Middle* attacks. An attacker may either replace the client certificate with a self-signed certificate of himself in order to gain the rights of being an authorized client. Or he could also set up an own VM and replace the transmitted VM certificate with the certificate of his VM. In this case he could manage to let the client send his collected data to the attacker's VM.

The solution in both cases is the verification of the integrity of the transmitted message. If the integrity is correct, the receiver can be sure that the message was not manipulated during the transmission. Apart from that, authentication of the sender is necessary. This way the receiver can be sure that the message was really sent from the valid sender but not from an attacker. Integrity verification and authentication is both achieved by a *Message Authentication Code* based on a shared secret only known to the VM and the client. The protocol is illustrated in figure 4.1. The creation of the VM certificate is not part of the illustrated protocol because we assume that the VM already possesses a valid certificate.

According to figure 4.1 a shared secret is generated on the *Admin Device* and delivered to the VM and the client. Apart from that, the client is also provided with the client name and the URL of the VM. The client name is used in the subject field of the certificate. The URL is necessary in order to let the client send its certificate to the VM.

The transmission medium for the client is a *QR-Code* where all information is encoded. The client then only has to scan the *QR-Code* in order to get the provided information. The reason for using a *QR-Code* is simply that it is more convenient to scan an image than typing in all the needed information. If a device is not able to scan a *QR-Code* because of a missing camera, for instance, the transmission medium can easily be replaced by a different side channel (e.g. USB flash drive).

Before the client certificate is created, a new key pair is generated because this is the main part of the certificate. Afterwards the *Message Authentication Code* is calculated using the HMAC procedure. The value of the MAC as well as the certificate itself are then sent to the VM using the HTTP protocol. The VM can then verify the integrity of the received certificate. Afterwards the MAC of the self-signed VM certificate is calculated and sent to the client together with the VM certificate itself. If the verification of the integrity fails for some reason, the protocol is aborted either by the VM or by the client.

4.1.2 VM ↔ VM

Participating entities are two VMs and the respective *Admin Devices*. Again, the goal is to exchange self-signed certificates and mark them as trusted which enables the authentication later on for the establishment of a TLS-protected connection.

The protocol is illustrated in figure 4.2. The creation of the two self-signed certificates is not part of the illustrated protocol because we assume that both VMs already possess valid certificates.

According to figure 4.2 the procedure is not much different than the one described in section 4.1.1. To prevent *Man-in-the-Middle* attacks the integrity of the sent messages must be verifiable and an authentication of the sender is required. This is again achieved by a *Message Authentication Code* based on a shared secret. The only difference now

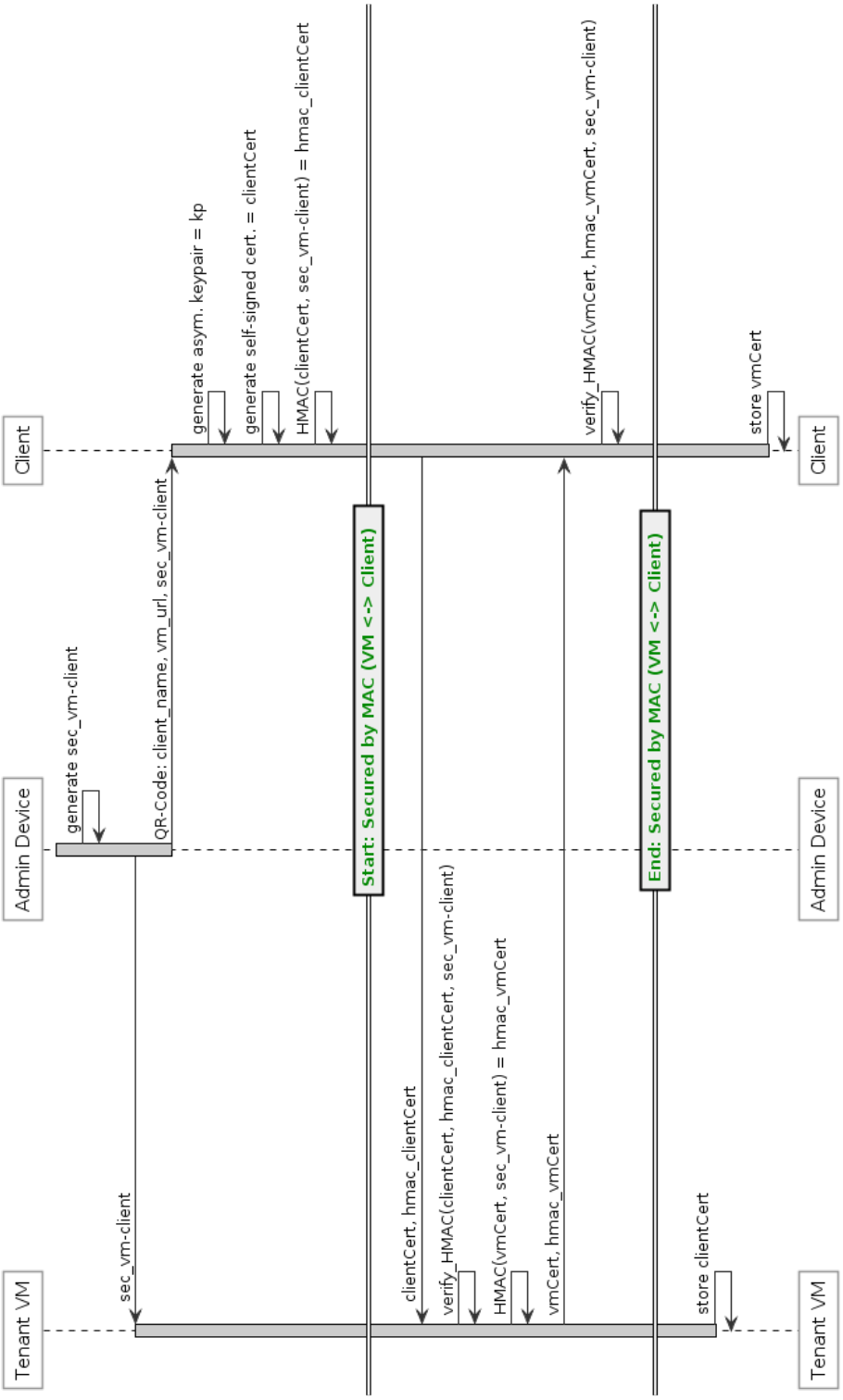


Figure 4.1: Pairing between a Client and a VM based on Self-Signed Certificates

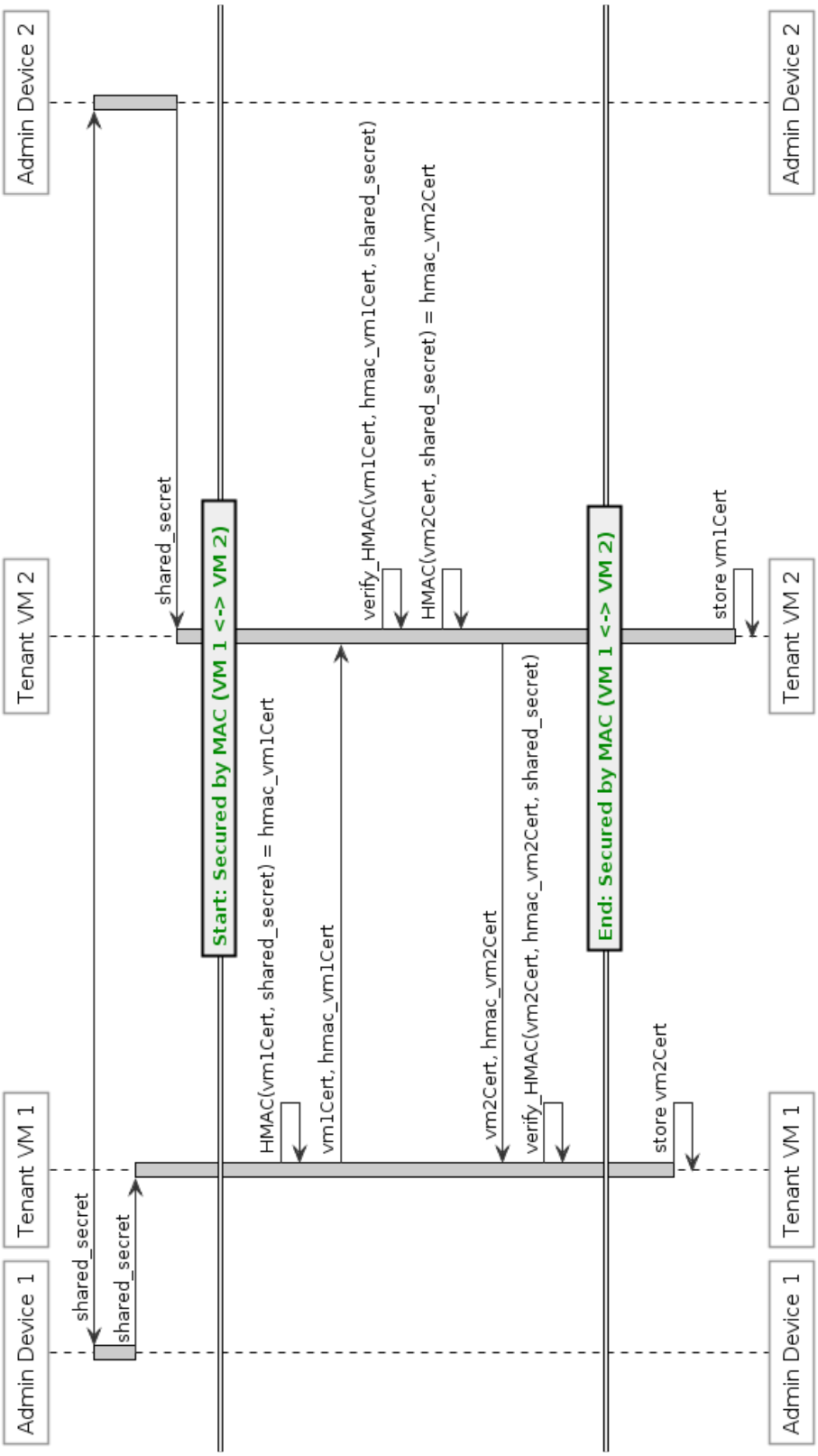


Figure 4.2: Pairing between two VMs based on Self-Signed Certificates

is that the two *Admin Devices*, more precisely the two tenants, have to agree on a shared secret. That is because the *Admin Devices* are the two entities for which a secure connection to the particular VM exists. The agreement on the shared secret must happen on a secure side channel. This can be a telephone, for instance. We assume this is acceptable because if a tenant wants to share data with another one, it must also be reasonable for him to make contact with the other person.

In analogy to the procedure described in section 4.1.1, the VM who wants to send data to the second VM must initiate the pairing by sending the certificate first. This can be justified as follows: If VM 1 (VM which wants to send data) receives the certificate of VM 2 and verifies the integrity, VM 1 can be sure that the pairing has been successful. Otherwise it would have never received a certificate from VM 2. At this point, VM 1 can open the connection in order to send its payload. In consequence, if the pairing is initiated by VM 2, VM 1 has no possibility to know whether the pairing has been successful. If it was not, due to transmission errors, for instance, opening a connection in order to send payload will fail.

4.2 Pairing based on a PKI

This solution can either be based on a local PKI or a global PKI. Using a global PKI makes the protocol a lot easier, especially the pairing between two VMs, and should therefore be preferred. Again, the goal is the establishment of a TLS-protected connection. The unique identifier that is needed in order to verify the authentication of a VM is the *domain name* per definition. The *domain name* is part of the URL of the VM.

4.2.1 Client ↔ VM

Participating entities are the VM, the client and the *Admin Device*. The following paragraphs describe the solution based on a local PKI. However, to improve this solution by the use of a global PKI only a few changes are necessary which is explained in the last paragraph of this section.

The root CA is the *Admin Device* which possesses a self-signed certificate. At the end of the protocol the client has a certificate signed by the root CA which he can use to authenticate himself against the VM later on. The main idea is that the client generates a CSR and sends it to the *Admin Device* (root CA) in order to get back a signed certificate.

Problematic in this solution is that the main conversation is between the client and the *Admin Device* (root CA) for which no bidirectional direct connection exists. Therefore, the VM is used as a means for the purpose because both the client and the *Admin Device* can connect to the VM. Due to the fact that no direct connection exist, we decided to use a *Message Authentication Code* based on a shared secret only known to the client

and the *Admin Device* as an additional security feature. This way the integrity of sent messages can be verified and authentication of the sender is possible, too. This protects against manipulation of the CSR on the VM where it is stored temporary until it is sent to the *Admin Device*. Sender authentication is also essential for the *Admin Device* in order to sign only valid CSRs. The protocol is illustrated in figure 4.3. We assume that the VM already possesses a valid certificate signed by the root CA.

According to figure 4.3 the shared secret is created on the *Admin Device* and delivered to the client. Apart from that, the client is provided with the client name, the URL of the VM and the root certificate. The client name is used in the subject field of the CSR. The URL of the VM is necessary in order to let the client send the CSR to the VM. The URL and the root certificate are also required by another reason: That is because the URL contains the *domain name* which is the unique identifier. The VM authenticates itself with a certificate signed by the root CA. Due to the fact that the client is provided with this information, he can verify the authentication.

The transmission medium is a *QR-Code* where all information is encoded. If a device is not able to scan a *QR-Code* because of a missing camera, for instance, the transmission medium can easily be replaced by a different side channel (e.g. USB flash drive).

After the storage of the root certificate a new key pair for the CSR is generated and the CSR itself. A *Message Authentication Code* is calculated using the HMAC procedure. Afterwards the value of the MAC as well as the CSR itself are sent to the VM. This is secured by TLS which is possible because the authentication of the VM can be verified by the client.

The two received information are transferred to the *Admin Device*, again secured by TLS. This is possible because we assume that this connection already exists after the point when a VM is assigned to a tenant. The *Admin Device* can then verify the integrity of the received CSR, create the certificate and sign it. Afterwards a *Message Authentication Code* for the certificate is calculated and together with the certificate itself sent back to the VM.

The VM transfers the certificate and the MAC to the client, secured by TLS. Finally, the client can verify the integrity of the received certificate and store it.

To improve this solution by the use of a global PKI, a few changes are necessary: The *Admin Device* is no longer the root CA but an intermediate CA. This means that it does not possess a self-signed certificate but a certificate signed by a globally verifiable CA. This leads to the fact that the client can authenticate the VM without being provided any certificate before. Hence, it is not necessary to encode a certificate in a *QR-Code* and store it on the client. The rest of the protocol is exactly the same.

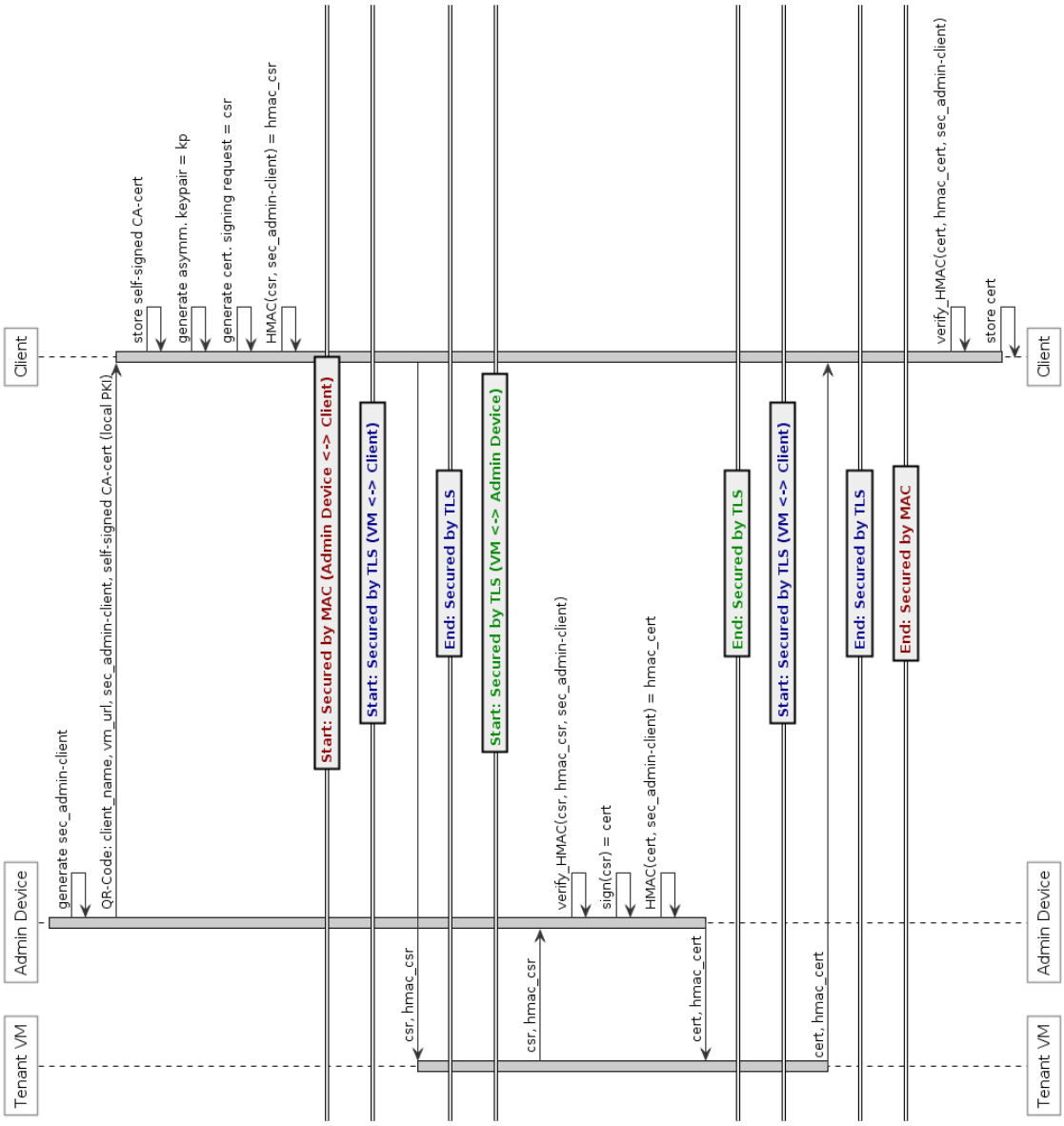


Figure 4.3: Pairing between a Client and a VM based on a PKI

4.2.2 VM ↔ VM

Participating entities are two VMs and the respective *Admin Devices*. The steps explained in the following are only necessary when we use a local PKI. The protocol is illustrated in figure 4.4.

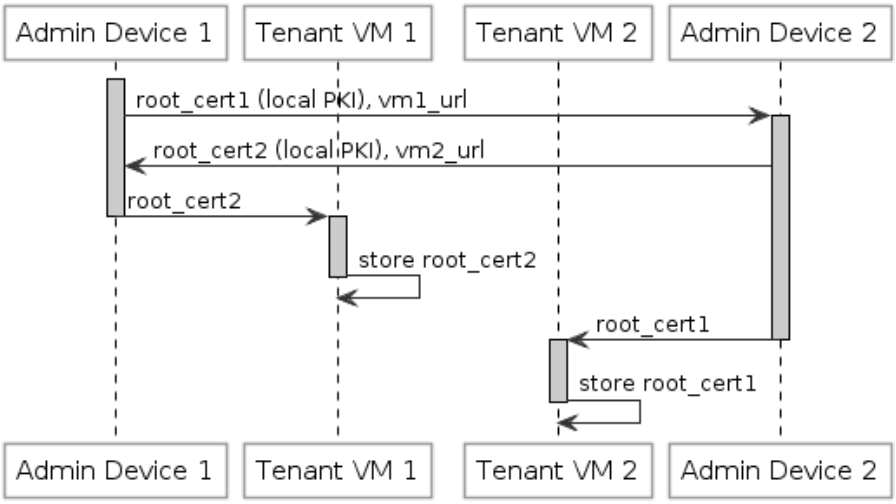


Figure 4.4: Pairing between two VMs based on a PKI

According to figure 4.4 the two *Admin Devices*, more precisely the two tenants exchange their respective root certificates. This step is necessary because the *Admin Devices* are the two entities for which a secure connection to the particular VM exists. The received root certificates are transferred to the VMs where they are stored. Apart from that, the URLs of the VMs are exchanged. This is required because the URLs contain the *domain names*, which are the unique identifiers.

The idea behind this is best explained in an example: We assume VM 1 wants to share data with VM 2 and therefore it connects to VM 2. As mentioned before, the connection should be TLS-protected. Therefore, VM 2 authenticates itself with its certificate signed by its root CA (*Admin Device* 2). Due to the fact that VM 1 knows the root certificate of *Admin Device* 2 it can verify the authentication and the connection can be established.

The exchange of the two root certificates between the two *Admin Devices* has to happen on an arbitrary side channel. The exchange itself does not have to be confidential because certificates do not consist of any sensitive data. However, the integrity must be verified to prevent *Man-in-the-Middle* attacks. This can be achieved by telephone, for instance. This also applies to the exchange of the two URLs of the VMs.

Exchanging certificates is not really comfortable for the tenants. However, in a real situation this is never needed because we assume that we use a global PKI if we basically decide to use the solution based on a PKI. In this case, it is only required to exchange

the URLs of the VMs, which can be done by telephone. We assume this is acceptable because if a tenant wants to share data with another one, it must also be reasonable for him to make contact with the other person. The example before would then look like this: VM 1 wants to connect to VM 2. Therefore, VM 2 authenticates itself with its certificate signed by the intermediate CA (*Admin Device* 2) which again has a certificate signed by a global verifiable CA. This way, the authentication can be verified directly and the connection can be established.

Chapter 5

Implementation

This chapter documents the implementation part of this thesis. The concrete protocol that has been implemented is the *MeasrDroid Pairing Protocol* between a client and a VM based on a PKI described in section 4.2.1. Due to the fact that no global PKI existed during the time of development, the solution is based on a local PKI. However, to improve this solution by the use of a global PKI only a few changes are necessary. The implementation required both work on the Android application and on the backend. The steps that are done by the *Admin Device* were not implemented because this is part of a different work.

5.1 MeasrDroid Core

The *MeasrDroid Core* is the Android application that has been extended. The work hereby was mainly the client part of the protocol. The programming language was exclusively Java. For testing purposes the *MeasrDroid Application* that is available on the *Google Play Store* was used [7]. The smartphone used for testing was a rooted *HTC Desire S* running a *Custom-ROM* based on *Android 4.0.4*.

5.1.1 QR-Code Scanner

A really important aspect of the protocol is the *QR-Code* because all necessary information is encoded there. This includes the root certificate, the shared secret between the *Admin Device* and the client, the URL of the VM and the client name. The information is encoded in URI style. This means that the *QR-Code* is only one String that contains all information concatenated with the & sign. An example would look like this: cert=abcd&secret=1234&url=measrdroid.de&name=client.

For the root certificate we used a size of 2048 bits. The certificate is encoded in the *PEM* format. Therefore, it has a size of 1402 characters. The shared secret is a symmetric key created with a KeyGenerator in Java. It is encoded using the Base64 encoding. Therefore, it has a size of 86 characters. The URL has a size of 26 characters and the client name has a size of 10 characters. Apart from that, the names of the variables must be encoded in order to be able to split the concatenated String in its components again. Therefore, 45 additional characters are necessary. In total, a size of 1569 characters must be encoded.

At the time of this writing a *QR-Code* can store a maximum size of 3Kb [13]. Hence, it is theoretically possible to encode all necessary information in one *QR-Code*. However, the test smartphone was not able to scan this amount of data. Maybe newer cameras are able to manage this. According to [14] it is not recommended to store more than 300 characters in one *QR-Code* if the *QR-Code* is meant to be read by a camera of a smartphone. Therefore, we decided to split the root certificate into three *QR-Codes* and used a fourth *QR-Code* for the encoding of the other information.

For the scanning of the *QR-Code* a library was used which is called *barcodescanner* [15]. The library is licensed under *Apache licence, Version 2* and can therefore be used for free. It is based on *ZXing* and *ZBar* [15]. The library provides an embedded solution for scanning a *QR-Code* without the need of installing any third party scanner applications. Positive is the simplicity of using it. The only thing that needs to be done is to implement the interface *ZXingScannerView.ResultHandler* and override the *handleResult(Result rawResult)* method. In the manifest of the application it must be allowed to use the camera. One negative aspect may be that development is still well underway. Hence, regular updates are quite presumable. Listing 5.1 shows an excerpt of the scanner class.

```
1  public class QRScanner extends Activity implements ZXingScannerView.
       ResultHandler {
2
3    [...]
4
5    @Override
6    public void handleResult(Result rawResult) {
7        splitRawResult(rawResult.toString());
8        finish();
9    }
10 }
```

Listing 5.1: Excerpt of QRScanner

The library allows some advanced settings, like automatic focus or flash. However, the standard settings are already suitable. When the camera is started and the *QR-Code*

is recognized, the method *handleResult(Result rawResult)* is called. In this context it is enough to only use the String representation of the *rawResult* because we know that the input is just an encoded String of the aforementioned information. The *splitRawResult(String rawString)* method is a helper method that splits the concatenated String in its components and stores them in the database. *MeasrDroid* uses the *SharedPreferences* for storing data. This way, stored data is accessible by all components of the application but not for other, installed applications.

Beside the *QRScanner* class, a *ScannerManager* class exists which manages all of the scanning. Therefore, it provides a method to actually initiate a scan and ways to check whether all needed information has already been scanned. Only if this is true the application can move on performing the pairing.

5.1.2 Certificate Tools

The *CertificateTools* class is a helper class that enables the creation of a key pair and a CSR. It is also possible to convert a CSR to the common *PEM* format. According to the coding conventions of tool classes, everything is defined statically here and it is not possible to create an object of this class.

The creation of the key pair is done by classes and methods from the *java.security* package. At first an instance of a *KeyPairGenerator* has to be created with the specified algorithm. The object then has to be initialized with the key size in order to generate the key pair. The algorithm and the key size are defined in the *PairingManager* class. We used *RSA* as the algorithm and a key size of 2048 bits which is strong enough at the time of this writing according to NIST [16].

By contrast, creating a CSR is not possible in Java without external libraries. That is because a provider is needed that implements the functionalities [17]. We decided to use the *Bouncy Castle* library because it is probably the most known library related to this topic, written in Java [18]. Also, it is free to use and well documented. One negative aspect in the eyes of the author is that it is not possible to use all of the needed functionalities by just adding one *.jar* file. To be exact, two files have to be included: one for the aspects related to the provider and one for the main aspects of creating a CSR. Listing 5.2 shows the important parts of creating the CSR.

```
1 public static PKCS10CertificationRequest generateCSR(KeyPair pair, String
      subject, String algorithm) {
2   PKCS10CertificationRequestBuilder builder = new
        JcaPKCS10CertificationRequestBuilder(new X500Principal(subject),
        pair.getPublic());
3   ContentSigner signer = new JcaContentSignerBuilder(algorithm).build(
        pair.getPrivate());
4   return builder.build(signer);
5 }
```

Listing 5.2: Excerpt of the creation of a CSR

The method returns an object of the class *PKCS10CertificationRequest* which represents a CSR. The needed parameters are the key pair, the subject and the algorithm which are defined in the *PairingManager* class. The algorithm is used for signing. We used *SHA256* with *RSA* encryption. At first, a *PKCS10CertificationRequestBuilder* object is created specified with the subject and the public key. Secondly a *ContentSigner* object is created specified with the dedicated private key. The *build(ContentSigner signer)* method finally creates the CSR.

The common format of a CSR is the *PEM* format. To convert the *PKCS10CertificationRequest* to *PEM* format a second method has been implemented that uses a *JcaPEMWriter* object to obtain the desired goal.

5.1.3 HTTP Client

According to the protocol, sending the CSR and the MAC to the VM has to be TLS-protected. Therefore, a HTTPS client is needed. This is already implemented in *MeasrDroid*. However, since Android 5.1 (Lollipop, API 22) all classes from the *org.apache.http* package have been deprecated [19]. The existing *HttpClient* class is based on these classes. Therefore, a new HTTPS client has been implemented based on *HttpsURLConnection* from the *javax.net.ssl* package. A customized *SSLSocketFactory* is specified with a *TrustManager* that trusts the root certificate. The root certificate has to be stored in the *KeyStore* before.

Due to the fact that posting data to a VM triggers network traffic, the *HttpsPost* is implemented as an *AsyncTask* [20]. This is necessary in order to prevent the *UI-Thread* from getting blocked because of background tasks.

5.1.4 Pairing Manager

The *PairingManager* class is responsible for performing the pairing of the VM and the client. The class consists of a constructor and one method that does all the needed steps: *public boolean doPairing(Context ctx).* This is the highest possible encapsulation and allows an easy usage. The only thing that needs to be done is to create an object of the class and call the *doPairing()* method. To be exact, this is never done directly but with a service (see 5.1.5). The method returns *true* if the pairing was successful, otherwise *false*. Everything else is kept private. For correct work a few input variables are needed. These are the application context, the client name, the URL of the VM, the shared secret and the root certificate. Apart from the context, all of this information is stored in the database. The *PairingManager* class also defines a few static variables at the very start which are needed for all subroutines. This concludes the algorithm for the MAC, the algorithm for the key pair and the key size. Also, the algorithm for the CSR is defined here and the subject itself where only the client name is added dynamically. These variables can be changed in order to satisfy the own needs.

The steps that are done in the *doPairing()* method are as follows:

- **Store root certificate**: This step is necessary for the TLS-protected connection between the client and the VM. That is because the VM authenticates itself with a certificate signed by the root CA. Therefore, the root certificate needs to be stored in the *KeyStore* so it can be trusted by a *TrustManager*. The location of the *KeyStore* is a directory on the filesystem. A *KeyStoreManager* already exists in *MeasrDroid* which is used for this task.

- **Generate CSR**: This is done using the aforementioned *CertificateTools* helper class. The CSR is converted to *PEM* format immediately.

- **Calculate MAC**: Before the MAC can be calculated the secret has to be created from the encoded String that is already stored in the database. This is done using classes and methods from the *javax.crypto* package. Also, an algorithm for the MAC has to be specified. We used the HMAC procedure based on *SHA-1*.

- **Send CSR and MAC**: This is done using the aforementioned *HttpClient* helper class. CSR and MAC are stored in a *HashMap* which is converted to post parameters before it is sent.

- **Receive certificate**: When the integrity of the sent CSR is verified by the *Admin Device* the certificate is created, signed and stored on the VM. From there it can be pulled by the client together with the MAC. Afterwards the certificate is stored in the *KeyStore* if the calculated MAC matches the received one.

5.1.5 Pairing Service

The overall *Pairing Protocol* can take quite a long time, depending on how fast the *Admin Device* signs the CSR and stores the certificate on the VM. Therefore, we decided to use an *Android Service*. [21] This has the advantage that the user does not have to wait the whole time for the application to be finished. The user can always exit the application and come back later on. Once started the service will continue its work until it is finished even if the application is closed. Secondly, a service provides an easy way to perform tasks in the background. This is needed in order to prevent the *UI-Thread* from getting blocked.

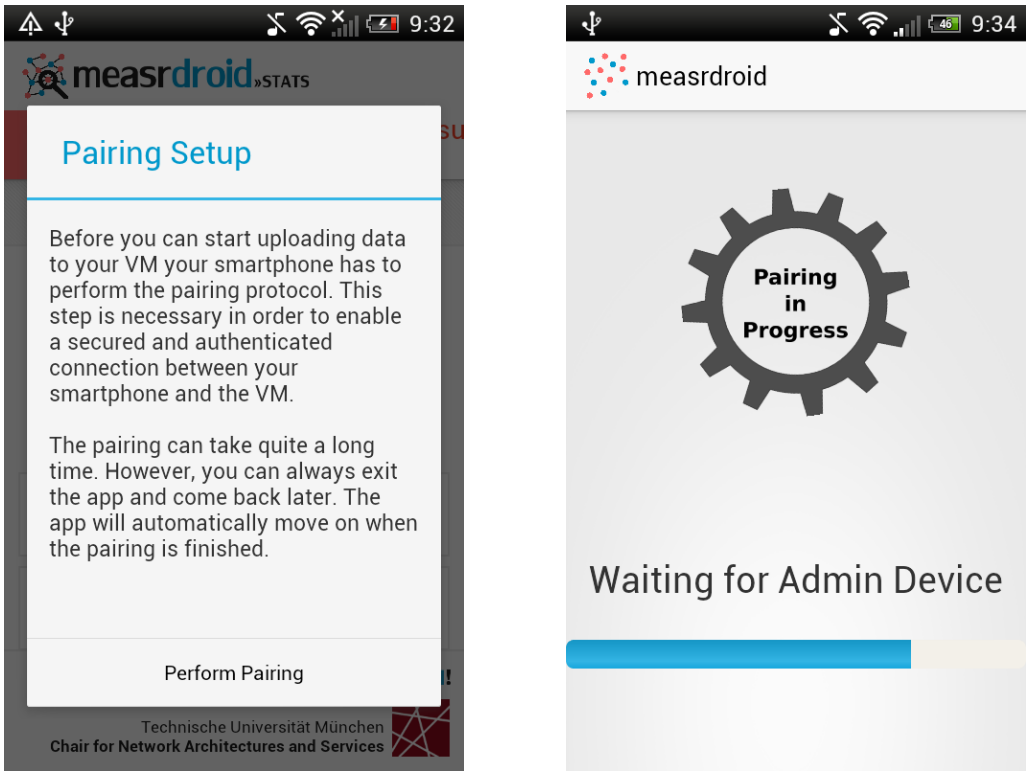
The *PairingService* class simply creates an object of the *PairingManager* class and calls the *doPairing()* method. If something went wrong, it waits for one minute and tries again until the pairing was successful. The service is managed by an already existing *ServiceManager* class.

5.1.6 Graphical User Interface

The implemented protocol does not really need a *Graphical User Interface* apart from the scanning of the *QR-Code*. That is because all of the main steps are done in the background using the aforementioned service. However, in two situations it appears to be useful to have a graphical feedback. Firstly, to inform the user of what will be done during the setup. This is achieved by a simple *Wizard Page*. Secondly, to have some kind of a *placeholder* which prohibits the user from doing something during the time when the service is running. Therefore, an activity has been introduced. A simple gear animation shows that the service is currently running and a progress bar together with some text informs the user about which step is currently done by the service. The activity can not be left, apart from exiting the whole application. When the pairing is finished it automatically destroys itself and moves on. Figure 5.1 shows two screenshots of the two mentioned situations.

5.2 MeasrDroid Backend

In this context the *MeasrDroid Backend* is the VM where CSR and MAC are pushed to and the certificate is pulled from. This thesis concentrates only on part of the web development. Therefore, a framework was used which is called *Flask* [22]. It is written in *Python* and is based on *Werkzeug* and *Jinja*. It is licensed under *BSD* licence and can therefore be used for free [22]. The advantages of using a framework like this are easy usage and the fact that not everything needs to be implemented by one's self. This includes *Cookie management* or defence against *Cross-Site Scripting*, for instance [22].



(a) Pairing Wizard (b) Pairing Activity

Figure 5.1: Graphical User Interface

Also, it is quite easy to implement a routing system based on URLs. Only a special decorator needs to be added. Listing 5.3 shows an example of this.

```
1 @app.route("/")
2 def index():
3     return "MeasrDroid is awesome"
```

Listing 5.3: Example of URL routing in Flask

The *route(path)* decorator tells Python to call a specified method when a certain path is requested. Requesting the index page automatically calls the *index()* method in this example. It is also possible to add more settings to the *route* decorator, like the allowed methods (GET, POST).

Flask comes with a simple built-in web server. This is good enough for development and testing purposes. However, in a real situation it may be better to use a more advanced server, like *Apache*.

5.2.1 TLS Protection

Due to the fact that the protocol foresees a TLS-protected connection between the client and the VM, the server must be able to handle *HTTPS*. This is also quite easy to achieve in *Flask*. Only the *ssl_context* variable must be specified in the *run()* method. This is in fact just a tuple of a valid certificate and the dedicated private key. The certificate is signed by the root CA.

5.2.2 Pairing and User Feedback

The only thing that needs to be implemented in this context is to receive the CSR and MAC and store them in a directory from where the *Admin Device* can pull them later on. *Flask* provides a *request* object which enables to read from a POST variable. The content of the *csr* variable is written to a file and stored in the *Uploads* directory. The exact same thing is done with the *mac* variable.

For user feedback the server sends the *HTTP* status code back to the client. This can either be 200 if the upload was successful, or 400 otherwise. Also, a short message is provided which describes what value is missing in the case of an error.

Chapter 6

Evaluation

In this chapter both the theoretical and practical part of this thesis are evaluated under certain aspects. This chapter also compares the two possible solutions of the *MeasrDroid Pairing Protocol* described in chapter 4.

6.1 Usability

Easy usage is an implicit requirement of every application. If an application is too complicated, it is quite unrealistic that the application ever becomes successful.

The parts that have been added to the Android application are directly inserted into the start wizard. Therefore, the same colour scheme is used and the operation is not more complicated than it was in the already existing application. In fact, only two user interactions are necessary: Firstly, accepting the information given in a wizard page by clicking on a button. Secondly, the scanning of the *QR-Code*. All of the rest is done automatically in the background which is quite comfortable for the user.

The scanning of the *QR-Code* itself is not really convenient in the current solution because the user has to scan four times. However, this is only because we use a local PKI at the moment as explained in section 5. In a real situation the local PKI will be replaced by a global PKI and in this case there is no need for scanning a whole certificate. Then only one *QR-Code* has to be scanned which is reasonable.

6.2 Performance

The overall performance of the *MeasrDroid Pairing Protocol* in the form it is implemented, depends an a few factors. A major factor is the power and equipment of the smartphone. If the smartphone is only equipped with a camera with low resolution, it may take

longer to recognize and scan the *QR-Codes*. The smartphone used for testing (HTC Desire S) was introduced in 2011. By comparison to newer smartphones the general power and equipment is much worse. However, also in this case it did not take longer than a few seconds to scan the *QR-Codes*. Also, the creation of the key pair, the CSR and the calculation of the MAC took only a few seconds. Due to the fact that we used a service which runs in the background, this depends furthermore on how many other tasks are currently running on the smartphone.

If we assume that the VM is online all the time, the upload of CSR and MAC is also really fast. In our test scenario the sent package had a size of only 1049 bytes. Hence, the upload does not take very long even with a low internet connection.

The further performance depends on how fast the *Admin Device* becomes active and signs the CSR. Due to the fact that the *Admin Device* is not online all the time and is also not in our area of competence, we can not make any statement about the amount of time in this case. However, in a real situation we assume that the tenant who wants to register a client immediately performs the necessary steps.

When the certificate is stored on the VM, it can take a maximum time of one minute until the smartphone tries to pull again. Then the certificate is downloaded, the integrity is verified and it is stored in the *KeyStore*. This takes only a few seconds again.

In our test scenario the whole pairing took only approximately 3 minutes. An important aspect is also the fact that the protocol needs to be performed only one time. Afterwards uploading data to the VM is possible in real-time (see 6.5). Hence, one can say that the overall performance of *MeasrDroid* is much better than before because the most frequently step (uploading data to VM) is much more efficient now.

6.3 Privacy Preservation

As even being part of the title, privacy preservation is a really important requirement of this thesis. According to the definition in section 2.1 privacy preservation are methods to obtain privacy.

In this context privacy means that collected data - owned by a tenant - is only accessible by the particular tenant at any time. This is mainly reached by encryption. Data is always encrypted on the client device before it is uploaded to the VM. Furthermore, data is only stored in encrypted format. This way, it can be ruled out that any unauthorized person is able to gain insights into the tenant's private data even if the person manages to get access to the VM. The private key for decryption is stored on the *Admin Device* which is only accessible by the tenant. Hence, the tenant is able to inspect or evaluate collected data on the *Admin Device*.

Secondly, privacy means that the decision on sharing private data with others can only be made by the tenant itself. A tenant can use the *Admin Device* in order to select parts of the collected data that should be shared with others. Also, the tenant can explicitly decide to whom selected data should be accessible.

To achieve these two aspects it is required to have a secure, integrity-checked and authenticated connection in every data transmission from one entity to another one. This thesis concentrates only on the situations when data is sent from a client to a VM (upload) and when data is sent from one VM to another one (sharing). By default there is no possibility to either verify the integrity of a message or to authenticate the sender/receiver of a message. According to the attacker model presented in section 3.4 this is susceptible to several attacks. Data can be manipulated during the transmission and it can not be verified that data is sent to a valid communication partner but not to an attacker.

Therefore, it is necessary to provide a mechanism that enables the establishment of such an integrity-checked and authenticated connection, which is basically a TLS-protected connection. This is achieved by the *MeasrDroid Pairing Protocol*. After performing the protocol both communication partner are able to authenticate each other and establish the secure connection for exchanging data. Hence, it can be ruled out that either the sender of a message is malicious or the receiver. Furthermore, no *Man-In-The-Middle* attacks are possible.

Hence, the security of the overall system lies in the *MeasrDroid Pairing Protocol* itself. If the protocol is secure, one can say that no attacks are possible to gain insights into private data during the transmission. This is proven in section 6.4 for both possible solutions presented in chapter 4. Due to the fact that data is always sent and stored in encrypted format, one can also say that privacy is obtained in the whole system.

6.4 Evaluation of the Two Solutions

The two solutions presented in chapter 4 are both suitable. Hence, it is worthy to compare the two solutions to each other. Apart from that, the security models are investigated in this section in order to prove that both solutions are secure.

It is useful to provide two possible solutions to achieve the same goals. In general, one can not say that either solution one is better or solution two. It depends on the situation.

The most common use case is probably that a tenant rents a VM which is managed by the service provider. Therefore, the whole IT infrastructure is provided by the service provider. This situation basically allows both solutions. Therefore, one can say that solution two based on a PKI is more beneficial here because it has more advantages according to section 6.4.3. However, if a tenant is not willing to pay any money, which

is the most decisive factor in most cases, solution one is also a good opportunity because self-signed certificates can be created for free.

In general, it is not required that the VM is managed by the service provider. It is also conceivable that a tenant sets up the whole IT infrastructure at his home. In this case it may be better to use solution one based on self-signed certificates. If the whole infrastructure is not connected to the Internet, it is not even possible to use a global PKI in this situation.

6.4.1 Investigation of the Security Model (Self-Signed)

The initial connection between the VM and the client or two VMs can not be secured. The security of this protocol lies only in the verification of the integrity of the sent messages and authentication of the sender. This way it can be proven that the messages were not manipulated during the transmission and were not sent from an attacker. This is achieved by a *Message Authentication Code*. Hence, the security lies in the shared secret between the VM and the client or between the two VMs.

The secret is sent to the VM securely because we assume that the VM is already assigned to the tenant. Hence, a secured connection between the VM and the *Admin Device* already exists. The *QR-Code* is a secure side channel per definition because we assume that the tenant only provides this information to the particular client he wants to register. This also applies to a different side channel like USB flash drive, for instance. With these two assumptions in mind we can assume that the shared secret is only known to the VM and the client and the setup for using a *Message Authentication Code* is therefore secure.

In the other situation we assume that the agreement on the shared secret has happened on a secure side channel, like telephone, for instance. Hence, we can assume that the shared secret is only known to the two VMs and again the setup for using a *Message Authentication Code* is secure.

Using a non-confidential channel for sending self-signed certificates is acceptable because certificates do not consist of sensitive data. However, the integrity must be verifiable because of *Man-in-the-Middle* attacks. Assuming that the HMAC procedure is secure, the whole protocol is therefore secure.

6.4.2 Investigation of the Security Model (PKI)

The security of this concept lies mainly in TLS. It is used to secure both the connection between the client and the VM and the connection between the VM and the *Admin Device*. We assume that the secure connection between the VM and the *Admin Device* already exists after the point when the VM is assigned to the tenant. Using TLS for the

connection between the client and the VM is possible, too because we assume that the VM possesses a valid certificate which can be verified by the client.

Under the assumption that TLS is a secure protocol, it can be ruled out that any transmitted message is manipulated during the transmission. However, authentication is only possible for the client at the initial connection but not for the VM. By implication, this means that anybody can generate a CSR and send it to the VM. Apart from that, the message can still be manipulated when it is stored temporary on the VM. Therefore, it is necessary to verify the integrity of transmitted messages and also authenticate the sender. This is achieved by the *Message Authentication Code* based on a shared secret between the *Admin Device* and the client. Hence, the further security lies in the shared secret.

The secret is encoded in a *QR-Code*. The *QR-Code* is a secure side channel per definition because we assume that the tenant only provides this information to the particular client he wants to register. This also applies to a different side channel, like USB flash drive, for instance. Hence, we can assume that the shared secret is only known to the *Admin Device* and the client and the setup for using a *Message Authentication Code* is therefore secure.

Assuming that the HMAC procedure is secure, the whole protocol for the pairing of a client and a VM is therefore secure.

For the pairing of two VMs only non-sensitive data has to be exchanged. These are certificates (only local PKI) and URLs (local and global PKI). This is also no security leak if we assume that the integrity of the sent messages are verified by the respective tenants. The exchange itself can happen on an arbitrary side channel.

6.4.3 General Comparison

This chapter compares the two provided solutions in general and independently from the situation.

Advantages of solution one based on self-signed certificates:

- **Direct connection**: The main conversation is either between a client and a VM or between two VMs. In both situations there is a bidirectional, direct connection, whereas solution two has to use the VM for redirecting transmitted messages.
- **Smaller QR-Code**: There is no need to encode a whole certificate in a *QR-Code*. Hence, this solution can get along with just one *QR-Code* which is convenient for the client. However, this is only an advantage over solution two if we assume that solution two is based on a local PKI which is probably never the case in a real situation.

- **Costs no money**: Self-signed certificates can be created with free tools. Hence, this solution does not cost any money, whereas solution two can be very expensive if every *Admin Device* needs a certificate signed by a globally verifiable CA.

Advantages of solution two based on a Public Key Infrastructure:

- **Based on TLS**: TLS is currently the international industry standard for secure communication. TLS is used to secure all transmitted messages during the performance of the protocol. As a consequence, it is also easier for foreigners to understand the security model of this solution.
- **No need for storing client certificates**: The VM does not have to store any client/VM certificates which saves memory. Also it is more performant to only authenticate a client/VM by verifying the root certificate, instead of looking through all stored client/VM certificates.
- **Scalability**: Based on the previous aspect authentication stays the same in performance even with more than 100 valid clients/VMs because only the root certificate must be verified. In solution one this requires to look through more than 100 certificates.
- **Easy extensibility**: If money is a decisive factor, this solution can simply be based on a local PKI which costs no money. However, improving this solution by the use of a global PKI is possible without many changes and should always be preferred because of other advantages.
- **Easy protocol**: If we assume that we use a global PKI, the pairing between two VMs is very easy to achieve. The only step that is required is the exchange of the URLs which is done directly by the tenants. Therefore, no further implementation is necessary, too.

Both solutions have their advantages and can be used in a real situation. However, advantages of solution two are considered to be more strong. This is mainly because of easy extensibility and scalability. If the protocol is only needed in order to register one client, the performance of the system after performing the protocol is nearly the same in both solutions. However, using solution two, the system is also performant in the case of more than 100 registered clients. Therefore, solution two was taken for the implementation described in chapter 5.

6.5 Real-Time Data Collection

As mentioned in section 1.3 real-time data collection is one of the desired goals of this thesis, more precisely of the whole new concept of *MeasrDroid*.

This was not possible in the old version of *MeasrDroid* due to security reasons. That is because a client was not able to directly push data to the database where it is stored. Instead, it was only possible to push data to the *Upload.droid* from where it is pulled after a certain time by the *C3PO*. The reason for this is that the *C3PO* does not need to be accessible on the internet which is probably the highest possible level of security. This way, most attacks against an IT system can be warded off. However, as explained in section 2.2.3 the time steps between two pulls can not be made arbitrary small which makes real-time data collection impossible.

In the new version of *MeasrDroid* real-time data collection is possible because every client can directly push data to the VM where it is stored. This means that uploading data from a client to the location where it is stored is no longer done by polling, but is event-based. This is the fastest possible solution because the only delay that may appear is due to network traffic.

However, allowing clients to directly push data to the VM is only possible because the VM is accessible on the Internet. This looses up the former security concept because in general, this enables two kind of attacks:

1. **Distortion of Data**: If every client is allowed to directly send data to the VM, an attacker could also distort collected data by uploading senseless data to the VM.
2. **General Attacks against the VM**: All conceivable attacks against a server, which is accessible on the Internet, can be performed against the VM, as well. This could be attempts to cripple the whole VM or to get access to it. If an attacker manages to get access to the VM, he has also access to all stored data.

Hence, a new protection mechanism is required in order to ward off these attacks. This is mostly achieved by the *MeasrDroid Pairing Protocol*. The protocol enables the establishment of a TLS-protected connection with mutual authentication. Therefore, the VM can be configured to never accept a connection with a client for uploading data if the client is not able to authenticate himself. If the client has performed the protocol successfully, he is able to authenticate himself and establish a connection. This means that attack 1 can be completely warded off only because of the *MeasrDroid Pairing Protocol*.

For attack 2 the argumentation is nearly the same. The only problem is that the VM can not be configured to never accept a connection in general. This is due to the fact that it must be possible for a client to either send his self-signed certificate or the CSR to the VM. However, it is not realistic that many attempts to perform the pairing appear in a short time. Therefore, the VM can be configured to only accept a few connections per day, for instance or use an IP blacklisting/whitelisting in order to prevent Dos attacks. Apart from that, common security aspects like *Intrusion Detection Systems* and *Firewalls* are in place to generally protect the VM against attackers. However, this is part of a different work and is therefore not investigated in this thesis.

Under the assumption that the mentioned security aspects are applied correctly, the further security lies only in the protocol itself. Due to the fact that the protocol is secure, as proven in section 6.4 one can say that the overall system is not more insecure than it was before and therefore real-time data collection is now possible without undermining the common security concept.

Chapter 7

Related Work

This chapter gives an introduction to other work that is related to this thesis.

There are many bachelor -and master theses related to the original *MeasrDroid* project. A selection can be found in [6]. Of course, these theses are also related to this work. However, this is not investigated in this chapter.

A paper that introduces a related topic is called *Multi-tenant Databases for Software as a Service: Schema-Mapping Techniques*, written by A. Kemper et al. and was published 2008 [23]. The authors describe a new schema-mapping technique which is multi-tenant capable. This technique is called *Chunk Folding*. The idea is that "logical tables are vertically partitioned into chunks that are folded together into different physical multi-tenant tables and joined as needed" [23]. Transferred to our context, this means that no isolated virtual machines are necessary in order to achieve multi-tenancy. Instead, there is only one server where all collected data is pushed to and multi-tenancy is achieved by the special schema-mapping technique. This is an interesting approach. However, it is not really suitable for our requirements, especially not for *privacy-preservation*. This is because the service provider learns all data collected by all of the clients and tenants, respectively. Instead, we require that each particular tenant is the only party with access to the collected data.

A second paper that is related to this work is called *CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization*, written by F. Zhang et al. and was published 2011 [24]. The situation in this paper is nearly the same as in our context: virtual machines are used in order to achieve multi-tenancy. The authors present the statement that "software stacks in typical multi-tenant clouds are non-trivially large and complex, and thus are prone to compromise or abuse from adversaries including the cloud operators" [24]. In simpler words, this means that virtual machines can be attacked and exploited. The authors developed a system for security monitoring which is based on nested virtualization. The prototype implementation is called *CloudVisor* and is mainly "responsible for protecting privacy and integrity

of resources owned by VMs" [24]. As often mentioned in this thesis, security plays a major role in order to obtain privacy which is the central requirement of this project. Therefore, the paper has also importance for our project but in fact, it is not required in our situation. The main idea of *CloudVisor* is to filter and monitor every access to the VM. If anybody - even the service provider - tries to get access to the data owned by the VM, he "can only see the encrypted version of that VM's data" [24]. This means that the best security guarantees (apart from integrity-checks) of *CloudVisor* are to never make data available to anybody in plain format. However, in our solution data is never stored in plain format. Hence, a monitoring tool like *CloudVisor* is not required.

Related to our project are also all kind of platforms where a user can upload personal data and evaluate it later on. An example for this is *Runtastic*. *Runtastic* is an austrian start-up founded in 2009 which was sold to *Adidas* in 2015 [25]. The idea of *Runtastic* is that every user can measure several aspects related to sports training (pulse, calories, kilometres, etc.) with a smartphone and upload it to the *Runtastic* platform where it is stored [26]. The goals are mainly to motivate users to do more sports. Therefore, uploaded data can be evaluated or shared with others in order to enable a comparison [26]. This is exactly the same which can be done in our project. However, there is one big difference: Due to privacy preservation in our project it is guaranteed that even the service provider has no access to uploaded data. This is not the case in a commercial platform like *Runtastic*. According to the privacy explanation the provider is explicitly allowed to collect personal data of its users and use the data for purposes of the *Runtastic* application [27]. However, the data is not transmitted to any third-parties [27].

Chapter 8

Conclusion and Outlook

This thesis is part of a bigger project. The entire project is to introduce multi-tenancy in *MeasrDroid* in a privacy-preserving way. Therefore, the overall architecture and structure of *MeasrDroid* has changed. Virtual machines are introduced in order to provide individual data sinks for each tenant. A tenant can have an arbitrary number of clients as data sources. Every client is able to upload collected data to the VM where it is stored confidentially. Furthermore, new concepts are introduced like sharing mechanisms between two tenants.

Privacy is the central requirement of the whole project. To obtain privacy data is never sent or stored in plain format but always encrypted. As a consequence it is not possible for unauthorized people to ever gain insights into some tenant's private data. Furthermore, privacy preservation requires the presence of an authenticated and integrity-checked connection for every data transmission. The *MeasrDroid Pairing Protocol* has been developed to enable the establishment of such a connection. The protocol needs to be performed before data is sent from one entity to another one and can therefore be seen as the initial pairing. As a side effect, it enables real-time data collection which has not been possible in the existing *MeasrDroid* project.

This thesis provides two different solutions for the *MeasrDroid Pairing Protocol*. The first solution is based on self-signed certificates, whereas the second solution is based on a PKI. Both solutions are suitable and secure which is proven in the thesis. Apart from providing the theory, the *MeasrDroid Pairing Protocol* has also been implemented in the form based on a PKI.

Currently there are two further students involved into the project: The first one is mainly responsible for the management of the VMs. This includes the VMLCS as it is called in figure 3.2 in section 3.2. The second one is mainly responsible for the web development of the VMs and the *Admin Device*. This includes user management or the steps in the protocol that are done by the *Admin Device*. Next, we also have to think about ways how to encrypt sensor data on the client device in order to efficiently enable

partial sharing with other tenants. This could be achieved by the use of *Attribute Based Encryption*.

All in all the project is very interesting and first steps are already done. However, a lot more things need to be done in order to have a multi-tenant *MeasrDroid* that is ready for the market.

Appendices

Appendix A

Abbreviations

- **CA**: Certificate Authority
- **CRL**: Certificate Revocation List
- **CSR**: Certificate Signing Request
- **HMAC**: Keyed-Hashed Message Authentication Code
- **HTTP**: Hypertext Transfer Protocol
- **HTTPS**: Hypertext Transfer Protocol Secure
- **MAC**: Message Authentication Code
- **MD5**: Message Digest 5
- **NIST**: National Institute of Standards and Technology
- **PKI**: Public Key Infrastructure
- **SHA**: Secure Hash Algorithm
- **SSL**: Secure Sockets Layer
- **TLS**: Transport Layer Security
- **UI**: User Interface
- **URI**: Uniform Resource Identifier
- **URL**: Uniform Resource Locator
- **VM**: Virtual Machine
- **VMLCS**: Virtual Machine Lifecycle Service

Appendix B

Glossary

- **Admin Device**: The *Admin Device* is the tenant's device with direct access to the particular VM. It is meant to be the control center of *MeasrDroid* for the respective tenant. Collected data can also be inspected and evaluated here.
- **Android**: Android is an operating system for smartphones. Android is based on Linux and is open source. Applications for Android are usually written in Java.
- **CA**: A *Certificate Authority* is an organisation that signs digital certificates.
- **Client**: A client is basically a device that uses some service of an IT-system. In this context, client always means smartphone.
- **CSR**: A CSR is kind of a precursor of a digital certificate. It contains all information related to the person for which the certificate is issued. This especially concludes the public key.
- **Digital Certificate**: A digital certificate is a signed piece of data that maps the identity of somebody to the public key. A certificate can either be self-signed or signed by a *Certificate Authority*.
- **MAC**: A MAC is a calculated value for a given message which enables both, the verification of the integrity of the message and the authentication of the sender.
- **MeasrDroid Backend**: The *MeasrDroid Backend* is the counterpart to the *MeasrDroid Core*. This concludes basically everything that is part in the architecture of *MeasrDroid* apart from the client.
- **MeasrDroid Core**: The *MeasrDroid Core* is the Android application that has to be installed on the device in order to measure sensor data.
- **PKI**: A PKI contains all parties that are necessary in order to create, distribute and verify digital certificates.

- **Privacy**: Privacy is the area in which a person can decide to whom, when and why information is accessible.
- **QR-Code**: A *QR-Code* is a two-dimensional code which can be used to store text. A *QR-Code* can contain a maximum of 3Kb. [13] In order to read a *QR-Code* specific scanner applications are necessary.
- **Smartphone**: A smartphone is a mobile phone with more possibilities than a conventional mobile phone. This concludes e.g. navigation, email, etc. [3]
- **TLS**: TLS is a protocol for secure communication. It contains authentication of the communication partner, confidential end-to-end data transmission and verification of the integrity of sent data.
- **VM**: A VM is a computer system that is not installed on a real hardware but runs in an isolated virtual environment of a real computer system. It can either have complete, partial or no access to the hardware. [28]

Bibliography

[1] "Statista - Prognose zur Anzahl der Smartphone-Nutzer weltweit von 2012 bis 2019," http://de.statista.com/statistik/daten/studie/309656/umfrage/prognose-zur-anzahl-der-smartphone-nutzer-weltweit/, Accessed: 2015-12-31.

[2] "Statista - Weltbevölkerung von 1950 bis 2015," http://de.statista.com/statistik/daten/studie/1716/umfrage/entwicklung-der-weltbevoelkerung/, Accessed: 2015-12-31.

[3] "Smartphone," Website, http://wirtschaftslexikon.gabler.de/Definition/smartphone.html, Accessed: 2016-01-07.

[4] P. D. L. Prechelt, "Vorlesung Anwendungssysteme: Privatsphäre," http://www.inf.fu-berlin.de/inst/ag-se/teaching/V-AWS-2011/31_Privatsphaere.pdf, Accessed: 2016-01-05.

[5] "MeasrDroid," Website, http://www.droid.net.in.tum.de/, Accessed: 2015-12-08.

[6] M. Faath, "Analysis of content delivery networks with an android-based measurement framework," Master's Thesis, Technische Universität München, Munich, Germany, 2013, advised by Dipl.-Inf. Johann Schlamp.

[7] "Measrdroid on the Google Play Store," Website, https://play.google.com/store/apps/details?id=de.tum.in.net.measrdroid.gui.stats, Accessed: 2015-12-07.

[8] "Public-Key Cryptography Standards (PKCS) 1: RSA Cryptography Specifications Version 2.1," Website, https://tools.ietf.org/html/rfc3447, Accessed: 2015-12-08.

[9] C. Eckert, *IT-Sicherheit*, 8th ed. Oldenburg Verlag, Jan. 2013.

[10] "Secure Hash Standards," http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf, Accessed: 2016-01-06.

[11] "MD5," Website, https://www.ietf.org/rfc/rfc1321.txt, Accessed: 2016-01-06.

[12] K. Schmeh, *Kryptografie*, 5th ed. dpunkt.verlag, Feb. 2013.

[13] "QR-Code," Website, http://qrcode.meetheed.com/question7.php, Accessed: 2015-12-08.

[14] "QR Code Generator," Website, http://goqr.me/de/, Accessed: 2016-01-12.

[15] "Barcode Scanner Library," Website, https://github.com/dm77/barcodescanner, Accessed: 2015-12-07.

[16] "Cryptographic Key Length Recommendation," Website, http://www.keylength.com/en/4/, Accessed: 2015-12-08.

[17] P. Lipp, J. Farmer, D. Bratko, W. Platzer, and A. Sterbenz, *Sicherheit und Kryptographie in Java*, 1st ed. Addison-Wesley, Jul. 2000.

[18] "The Legion of the Bouncy Castle," Website, https://www.bouncycastle.org/, Accessed: 2015-12-08.

[19] "Android 5.1 APIs," Website, http://developer.android.com/about/versions/android-5.1.html, Accessed: 2015-12-08.

[20] "AsyncTask," Website, http://developer.android.com/reference/android/os/AsyncTask.html, Accessed: 2015-12-08.

[21] "Services," Website, http://developer.android.com/guide/components/services.html, Accessed: 2015-12-08.

[22] "Flask," Website, http://flask.pocoo.org/, Accessed: 2015-12-08.

[23] S. Aulbach, T. Grust, D. Jacobs, A. Kemper, and J. Rittinger, "Multi-Tenant Databases for Software as a Service: Schema-Mapping Techniques," *SIGMOD '08 Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008, http://wwwkemper.informatik.tu-muenchen.de/research/publications/conferences/sigmod2008-mtd.pdf, Accessed: 2016-01-08.

[24] F. Zhang, J. Chen, H. Chen, and B. Zang, "CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization," *SOSP '11 Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011, http://www.sigops.org/sosp/sosp11/current/2011-Cascais/printable/15-zhang.pdf, Accessed: 2016-01-08.

[25] "Adidas kauft Runtastic," Website, http://wirtschaftsblatt.at/home/boerse/europa/4794623/Adidas-kauft-Runtastic_Ein-Deal-mit-perfektem-Timing, Accessed: 2016-01-13.

[26] "Runtastic," Website, https://www.runtastic.com/de/ueber, Accessed: 2016-01-13.

[27] "Runtastic Datenschutz," Website, https://www.runtastic.com/de/datenschutz, Accessed: 2016-01-13.

[28] "Virtuelle Maschine," Website, http://www.itwissen.info/definition/lexikon/Virtuelle-Maschine-VM-virtual-machine.html, Accessed: 2016-01-07.