Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Secure P2P-Messenger

Distributed Systems Lab Project

Roni Häcki, Adrian-Philipp Leuenberger
haeckir@ethz.ch, leadrian@ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Philipp Brandes, Tobias Langner
Prof. Dr. Roger Wattenhofer

September 15, 2014

Contents

1 Abstract

In this Distributed Systems Lab Project we built on previous work on the P2P Messenger and introduced several new features. We tackled the problem of authenticity and developed an authentication server. The authentication server signs keys that are used for signing messages. Additionally we added the possibility to use several authentication servers and built a hierarchy. Further, we introduced encryption that has perfect forward secrecy. With this authenticity, integrity, and, confidentiality are guaranteed. We added key synchronization of different devices and the usage of the Google Cloud Messaging service to the P2P Messenger. Synchronisation is performed in order to let different devices of a user receive the same messages where as the usage of the Google Cloud Messaging service tries to reduce power consumption on mobile platforms.

2 Introduction

We all use instant messaging apps like WhatsApp or Skype to communicate with our friends. Unfortunately, most of these apps use a proprietary protocol, which, on one hand, makes them rather untrustworthy and, on the other hand, forces you to use one specific client. Open alternatives like XMPP (Jabber) are server-based and have a single point of failure. There is no existing solution that uses an open peer-to-peer based protocol.

Only recently people started adding encryption natively into their messengers, also due to the fact that the NSA is eavesdropping on all of us. Most apps still send messages in a way that is easily accessible for anybody with access to the connection. Not to mention that some messaging apps are well-known for having security issues on a regular basis. The few apps trying to cope with these issues often implement custommade cryptography solutions, which usually do not satisfy the high standards requested by cryptographers. On top of that, those solutions are mostly still server-based.

We have extended the already existing P2P Messenger presented in [1] and [2].

3 Related Work

There are several instant messenger that implement different feature set.

3.1 Jabber

Jabber [3] was the first instant messenger to use the open XMPP protocol [4]. Using XMPP, the Jabber instant messenger was able to exchange messages with other instant messenger that implemented the XMPP protocol. It is server-based and features encryption based on the Transport Layer Security (TLS) protocol.

3.2 Off-the-Record Messaging

One of the few open protocols that feature perfect forward secrecy is Off-the-Record Messaging [5] or short OTR. Perfect forward secrecy is guaranteed by exchanging new keys with every message. This key exchange is based on Diffie-Hellman and is combined with a hash function to generate symmetrical keys for AES [6].

3.3 WhatsApp

The most common instant messenger is currently WhatsApp [7]. WhatsApp uses XMPP to exchange messages. It supports a lot of features like exchanging photos, videos, audio and even the current location can be exchanged. One of the weak points of WhatsApp is its security. Currently it does not feature a sufficient end-to-end encryption and further, since Facebook acquired WhatsApp one can not be sure where our data ends up.

3.4 SafeSlinger

SafeSlinger [8] is a system to securely exchange authentication information between smartphones. Additionally it features a secure way to exchange messages. It is based on a centralized but untrusted cloud server. SafeSlinger features a protocol for the key exchange and verification of this information which is split up into three rounds. In a first round Keys are exchanged by Diffie-Hellman but also features a group Diffie-Hellman key agreement where not only two parties participate in the key agreement. The keys are hashed to a symmetric key using a SHA-3 hash function. The other two rounds remaining are for verification.

4 Architectural Overview

In this section, we present the overview of the architecture we have chosen. The main factors that influenced our choice of architecture are confidentiality, authenticity, and integrity.

4.1 Guaranteeing Authenticity and Integrity

An important goal of a secure instant messenger is to guarantee the authenticity of its users. This means, user Alice must be able to verify that the user that claims to be Bob is indeed Bob.

The common way to guarantee authenticity is to use digital signatures. Before Bob sends a message to Alice, Bob signs the message with his private key. Then, Alice can verify the signature using Bob's public key.

However, this *basic* approach is not completely safe. The problem is that a malicious entity *Eve* can still generate a key pair and simply send the public key of this pair to Alice, claiming that it is Bob's public key. Eve can therefore still impersonate Bob.

The solution to this problem is to certify the public keys using one or more *trusted* entities. We chose an approach very similar to the way SSL certificates are handled. Our approach looks as follows.

Once Bob has generated his private key pair, he connects to a certificate server (certificate authority), which then certifies Bob's key and sends back a signature of the key (and Bob's user name). Furthermore, the certificate server also sends Bob his own certificate. The certificates used are according to the X.509 [9] standard. The most important fields are the issuer, the subject the certificate belongs to and its validity. With the certificate, Alice can verify that Bob is truly Bob and Eve cannot impersonate Bob, as she does not have a valid certificate for her fake Bob-keys.

Now, another problem might arise if the certificate server itself is controlled by Eve. To prevent this, all certificate servers are ultimately certified by themselves, in a hierarchical way. To be more precise, a certificate server a user connects to is again certified by a higher tier certificate server which again is certified by another even higher tier certificate server. This goes on and on upwards in the hierarchy up to a root server (root CA) which *must* be assumed to be trusted and not to be compromised.

Using this hierarchical architecture, Alice *and* Bob himself can verify that Bob's certificate is indeed valid, assuming they trust the root CA. Furthermore, it simplifies adding new friends. If there were no certificate authorities, the only way Alice and Bob could make sure that they have the correct public keys to verifies each others signature is to exchange not only the usernames but also the very long public keys themselves. However, if we use CAs this is not necessary as the public keys can be exchanged and verified by the devices. Alice only has to tell Bob her username.

The architecture can be seen in Figure 1.

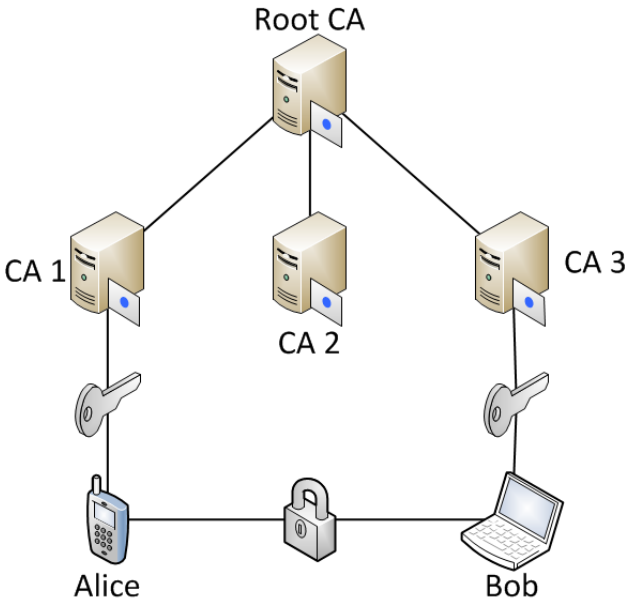


Figure 1: Hierarchical architecture for certificate authorities

4.2 Guaranteeing Confidentiality

Another important goal of a secure instant messenger is confidentiality.

We have implemented two approaches to achieve confidentiality. As a first approach, we chose to implement message encryption using asymmetric RSA encryption. For this approach, each user simply generates another key pair and some certificate server certifies the public key of the user. Just as for the digital signatures, when Alice receives a message from Bob she can verify that the data has been encrypted using Bob's private key.

However, using RSA does *not* provide Perfect Forward Secrecy – meaning, once Bob's private key has been leaked, all past messages sent to Bob *can* be decrypted.

To overcome this problem we have implemented an Off-the-Record protocol (to be used in the final messenger), which guarantees Perfect Forward Secrecy. The OTR protocol was presented in [5].

The diffence between OTR encryption and RSA encryption is that instead of using long-term asymmetric encryption keys, short-term symmetric Diffie-Hellman keys are generated and discarded again as soon after they have been used for encryption and decryption. With every message a new public key is added to the message. When Bob receives a message from Alice, it is encrypted with a key that was previously sent by Bob. Further, the received message contains a new public key from Alice from which Bob

can generate a new secret with which he can encrypt further messages sent to Alice.

4.3 Google Cloud Messaging

Additionally to the security features, we also implemented features which reduce power draw for mobile devices. Google has a service called *Google Cloud Messaging (GCM)* which is very handy for instant messenger applications. The GCM framework can reduce power consumption for the receiving side of an instant messenger by storing/queueing the message on GCM servers if the device is not online and deliver the message later by waking the Android application on the Android device. An implementation of GCM has two essential parts a client and a server. To leverage the features from GCM, we integrated it into the messenger. We implemented GCM in two ways, the official way how Google suggests it, and in a more peer-to-peer fashion without using a centralized server.

5 Implementation

5.1 Certificate Server

For our certificate servers, we naturally chose a multi-threaded architecture.

On the server, there is a main thread which is always listening for new incoming connections and there is a thread pool with a bound number of worker threads. Once a client connects, the main thread creates a new task to be eventually served by some worker from the thread pool. Then the main thread continues to listen for new incoming connections. Depending on the incoming message, different actions are performed.

We will now present the different possible message types and their matching action. All packets between clients and servers are JSON-formatted. The following types of messages can be sent by a client to a server (*Note that a client might also be another certificate server from a lower tier*):

- **Signature request:** A client (messenger user) might request a signature for his public keys. There are two different cases:

  1. The client has *never* requested a signature before.
  2. The client has *already* requested a signature before.

  In case 1, the server simply aggregates the clients public keys, user ID and a randomly generated salt (generated by the server) and signs this data. Then, the server sends back the signature, the salt and its own certificate. This way, the user Alice can verify that Bob's keys indeed belongs to him, as Bob's user ID and public keys have been signed together. Furthermore, the server stores Bob's user ID and keys into a MySQL-based certificate database. This database is very important for case 2.

  In case 2, the server cannot simply issue a new signature. The problem is that the malicious entity Eve might generate a new key pair, send a signature request and claim to be Bob, even though real Bob's real keys have already been certified. To prevent Eve from impersonating Bob, the server requires Bob to include a challenge. This challenge is a simple signature of Bob's name, the old and new keys. This signature must have been created using Bob's old private signature key and can be verified using Bob's old public signature key. The public key is stored in the previously mentioned certificate database, as the real Bob has already requested a signature before. If we constrain signature requests like this, we can prevent Eve from requesting a new signature in Bob's name, as long as Bob's private keys have not been leaked to Eve.

  The typical signature request packet can be seen in Listing 1. It includes the user ID `id`, the encryption key `enc-key`, the challenge `chal`,

the signature key `sig-key`, a request number `req-num` and a message type `type`.

Listing 1: Signature request packet

```
{
  "id":"Bob",
  "enc-key":"313233AF34D4298B..." ,
  "chal":"4465722050617073746520696
      e2064656e2057616c6465207363686569737374",
  "sig-key":"132DA35257234FFC..." ,
  "req-num":1,
  "type":"sig-req"
}
```

Upon receiving the server's response packet which includes the signature, the client can verify the signature using the certificate sent by the server. Once this signature has been verified, the client will also verify the certificate by requesting the certificate from the next higher-tier CA server until it contacts the root. The root certificate does not have to be verified as its public key is hard-coded.

- **Certificate request:** A client – which might be a messenger user or another certificate server – can request the certificate server's own certificate. In this case, the server simply fetches its certificate from a local key store and sends it back to the client.

  Certificate requests are required for a client in order to verify certificates up to the root.

  The typical certificate response packets can be seen in Listing 2. We omitted the request packet as it looks almost the same as the response packet. Both packet types include a request number `req-num` and a type field `type` and in addition, the response packet also includes the certificate `cert`, encoded in hexadecimal format.

Listing 2: Certificate response packet

```
{
  "req-num":2,
  "type":"cert-rsp",
  "cert":"BADF00D"
}
```

- **Bootstrap packet:** There are two cases when a client needs some sort of bootstrapping. The first case is when the client does not know any of the certificate servers and wants to contact a well known server, i.e., the root CA (root of the chain of trust). When the root is contacted using a Boostrap packet, it will respond with a packet containing a host name which is in the hierarchy of certificate servers. The next

time the client will no longer contact the root server (if there is any other), but it will contact the address returned from the server. This way the load can be balanced between the certificate servers. For a bootstrapping problem there are a lot of smart solutions that find a node to which a newly joining node may connect to. The easiest and frequently used solution is, as described before, contacting a well known node.

In the same way, by contacting the root certificate server, another certificate server can join the hierarchy of the certificate servers. At the moment (out of simplicity) the root certificate server knows the whole hierarchy and can directly return the address of the server which will be the next element up to the root. Another way to implement this, would be to let the server itself go from the root certificate sever down the hierarchy and find its place. This could avoid the problem of one node knowing the whole structure.

In Listing 3 a bootstrap packet can be seen. As other packets, it contains the two fields `req-num` and `type`. The field `host` is either empty (for clients) or contains the address of the server (for other certificate servers). The field `subtype` indicates if this packet is sent by a messenger application or by another server. The response uses the same packet type and just changes the `host` field.

Listing 3: Bootstrap packet

```
{
  "req-num":2,
  "type":"boot",
  "host":"pc-10129.ethz.ch",
  "subtype":"0"
}
```

- **Certificate creation packet:** After certificate server knows where it belongs in the hierarchy, it must request a certificate from a higher authority in the chain of trust. This can be achieved by sending a Certificate creation request packet. The request contains the subject (host address) for which the certificate should be issued, and the public key of a public/private key pair. This is enough information for the server higher in the hierarchy to issue a certificate for the subject specified in the request.

  After the server received the issued certificate, a client requesting a signature for his public key can verify the signature by following the chain of trust up to the root CA by requesting their certificates using certificate requests.

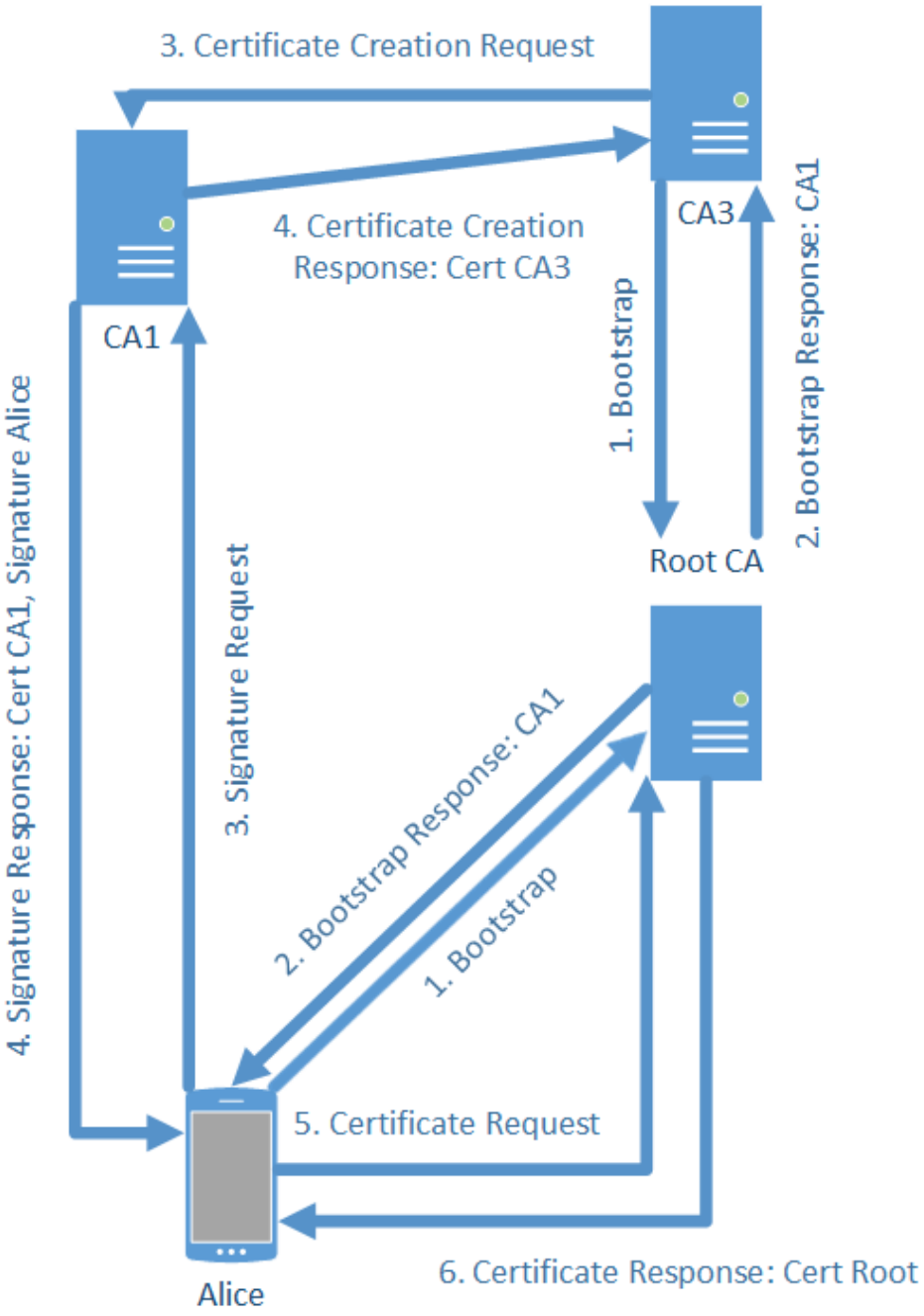


Figure 2: Example of Alice first bootstrapping then requesting signature and verifying it. Additionally, CA3 joins the hierarchy.

5.2 Message Encryption

In this section, we present the mechanism behind the message encryption. Before clients can send to each other they need to establish a chat session. In the following, chat sessions are referred to as streams. The mechanisms for messages encryption include the handshakes between the clients when streams are established as well as the encryption and the storing of the keys and certificates. As already mentioned, we have implemented Perfect Forward Secrecy. To be more precise, we used Diffie-Hellman key agreement where the DH keys are frequently dropped and new ones are created.

5.2.1 Diffie-Hellman Basics

The Diffie-Hellman key agreement protocol is secure based on the assumption that *prime factorization* is a hard problem. The goal is that two users Alice and Bob are able to agree on a secret shared key. This key agreement is performed over an insecure medium where a malicious entity Eve might eavesdrop.

We now briefly present how Diffie-Hellman key agreement works. Initially, both Alice and Bob and likely even Eve know about the public generator g and the modulus m. To perform the key agreement Alice and Bob choose both generate a secret/private key a and b respectively. Neither of these keys is ever sent to a communication partner. Next Alice and Bob compute their public keys $A = g^a \mod m$ and $B = g^b \mod m$ respectively and exchange these keys. Now Alice can compute

$$B^a \mod m = (g^b \mod m)^a \mod m = g^{ba} \mod m$$

while Bob computes

$$A^b \mod m = (g^a \mod m)^b \mod m = g^{ab} \mod m$$

.

Of course, the keys that Alice and Bob compute are identical. However, Eve cannot compute $g^{ab} \mod m$ even if she knows g, m, g^a and g^b. To be able to compute $g^{ab} \mod m$ Eve would have to know about a or b. Assuming that *prime factorization* is hard, Eve cannot derive a from g^a or b from g^b.

Still, there is a problem regarding the key agreement – Eve can perform a man-in-the-middle attack and pretend to be Alice while talking to Bob and pretend to be Bob while talking to Alice. Eve can generate a secret key e and would perform a key agreement with Alice and Bob where the key that Alice uses is $g^{ea} \mod m$ and the key that Bob uses is $g^{eb} \mod m$. Eve would decrypt messages received by Alice using the key $g^{ea} \mod m$ and re-encrypt it again using $g^{eb} \mod m$. Neither Alice nor Bob would be aware that their messages have been tampered. To overcome this problem, the

messages have to be signed by Alice and Bob which guarantees authenticity and integrity as modifications would be detected.

5.2.2 Handshaking

In our architecture, we focused on encrypting messages sent in active chat sessions (streams) such that eavesdropping by malicious entities is not possible.

Assuming that Alice and Bob are friends, the first step user Alice does to start a chat with user Bob is to send a stream request. Upon receiving a stream request, Bob automatically sends a stream response back to Alice. We used this functionality of the already existing P2P Messenger and extended it such that it also serves as handshaking for key agreements.

Both message types, stream request and stream response, include the same data that is required to set up a secure stream:

- A signature for all the keys S
- A signature key K_S
- A certificate C to verify that S is indeed Alice's or Bob's signature key
- A first Diffie-Hellman key $g^a \mod m$ or $g^b \mod m$.

Neither stream request nor stream response are encrypted as it is not required. Alice's signature key is public and as already mentioned in Section 5.2.1, malicious entities cannot factorize $g^a \mod m$ or $g^b \mod m$. Furthermore, the signature key K_S can be used to verify the signature S and therefore verify the proposed keys. The signature key itself can be verified using the certificate C which in turn can be verified as described in Section 5.1.

5.2.3 Message Encryption

Message encryption should guarantee Perfect Forward Secrecy. To achieve this in our implementation, user Alice generates a new private Diffie-Hellman key every time she sends a new message to Bob. To encrypt the i-th message, Alice generates key a_i and encrypts the message using the secret key $g^{a_{i-1}b_j} \mod m$ where b_j is the key of j-th message that Alice has received from Bob. Note that Alice has used the key a_{i-1} for the encryption. Key a_i is included in the message to be sent but not encrypted, as it is not really necessary, assuming that the private part x of a public key $g^x \mod m$ cannot easily be reconstructed. Furthermore, encrypting the new keys would lead to chain of encryption. Decrypting a message m_i would require to have decrypted message m_{i-1}, which in turn would also require message m_{i-2} to

be decrypted at the receiver. If even *only* one message is lost, none of the following messages could be decrypted. This would break the whole chat session. However, if only the texts are encrypted, but not the keys, losing a message would not be catastrophic. If for example message m_{i-1} is lost, the ciphertext in message m_i cannot be decrypted but the key can still be extracted and therefore the following message m_{i+1} can be decrypted again.

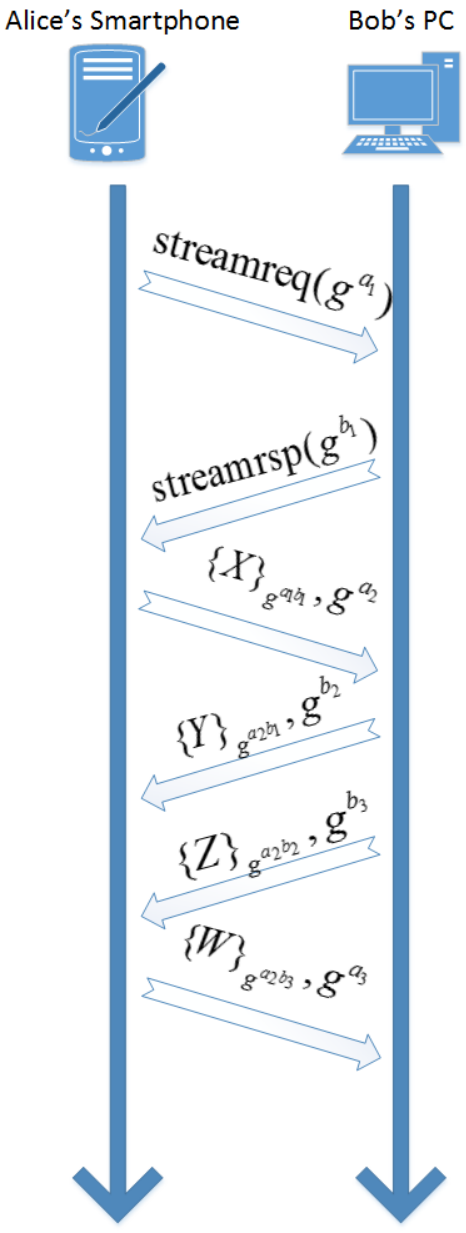
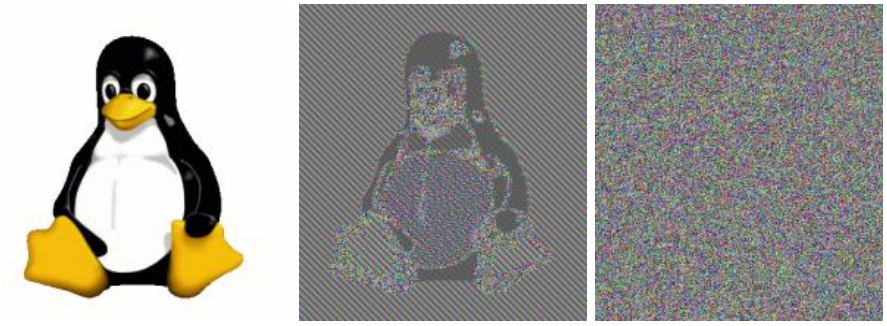


Figure 3: Chat example using the implemented encryption scheme

Furthermore, we also guarantee authenticity and integrity by signing all encrypted messages using the private counterparts of the public keys that are sent in the initial stream requests and responses. To verify authenticity and integrity of an encrypted message sent by Bob, Alice verifies the signature using the public key she obtained and verified when the stream was set up.

It is important to know what kind of encryption we use. Since Diffie-Hellman key agreement produces symmetrical keys, we had to choose a symmetrical encryption algorithm. We choose the *Advanced Encryption Standard (AES)* [6] algorithm as it was known to be fast and save even if the keys have only a length of 128 bits. To be safe we chose to use 256 bit long keys. The keys are generated by hashing the symmetric Diffie-Hellman keys into 256-bit keys using SHA-256. At some point we had to downgrade the hashing to the less secure MD5 as SHA-256 did not seem to be supported by our JVM on Windows machines. However, we assume this downgrade only to be temporary.

AES is a block-cipher-based algorithm meaning that the data is partitioned into blocks and the encrypted block-wise. This block-wise encryption can be performed using several schemes. The most simple scheme is to encrypt each block independently using *Electronic Code Book* mode (ECB) [10]. However, this mode is not very secure. Duplicate blocks will always be encrypted the same way. Such a behaviour might be fatal in certain use cases. A relevant example is the transfer of image files. Any modern instant message should be able to support file sharing and even though this is not yet supported in the P2P Messenger, we already provide functionality to support secure file transfer. One can see the results of block cipher encryption using ECB mode in Figure 4. The reader can clearly see that Tux is still visible in Figure 4b even though the image has been encrypted.

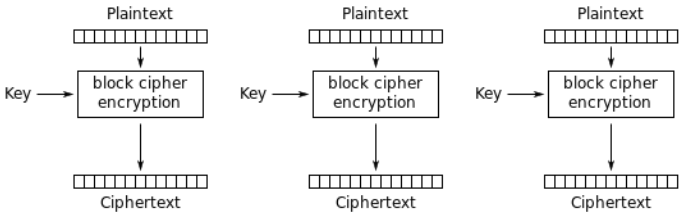


(a) Tux (Original) (b) Tux (ECB) (c) Tux (Secure, e.g., CBC)

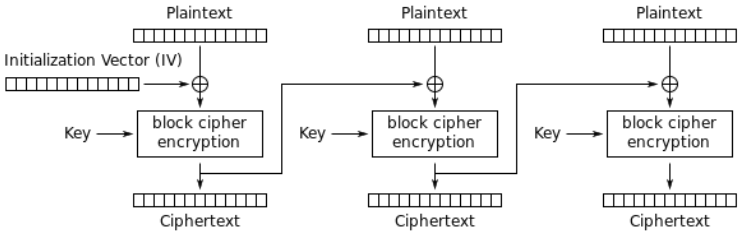
Figure 4: Tux image unencrypted, encrypted with ECB and encrypted with a secure block cipher mode (e.g. CBC). [10]

To prevent this kind of information leakage, we have decided to use

AES in combination with a *secure* block cipher mode, namely Code Block Chaining (CBC) [10]. Sketches of EBC and CBC can be found in Figures 5a and 5b.



(a) Electronic Code Book



(b) Code Block Chaining

Figure 5: EBC and CBC mechanism sketches.[10]

The idea behind CBC is to encrypt the data such that the encryption/decryption of the data block always depends on the previously encrypted datablock. We can represent this idea with the following formula [10]:

$$C_i = E_K(P_i \oplus C_{i-1}), C_0 = IV$$

To encrypt the i-th datablock P_i it is first XORed with the previous ciphertext C_{i-1} and than encrypted with the encryption key K. Note that IV is an initialization vector on which both parties agreed upon and which must be changed with every encryption.

5.2.4 Discarding Temporary DH-keys

To achieve Perfect Forward Secrecy it is important to discard old or unused secret Diffie-Hellman keys. Otherwise an unauthorized entity might see all the keys if she/he gains access to some user's device(s). The challenge of discarding keys is to discard keys as soon as possible. This section describes how the Diffie-Hellman keys are deleted.

First of all, keys created by a user's own devices cannot be deleted the same way as keys generated by his peers. Deleting keys of a peer is straight forward. When Alice receives the i-th message from Bob, she can directly

delete the key that she has received with the previous message, the $(i-1)$-th message. Let us assume Bob's i-th key is the key Alice received in Bob's i-th message. The keys of Alice and Bob are denoted by a_i and b_i, respectively. Alice can safely delete b_{i-1} after she has received Bob's i-th message because she knows that Bob will never use this key again as he will generate a new Diffie-Hellman key with every new message he sends. Bob on the other hand cannot directly delete key b_{i-1} after generating key b_i and sending his i-th message. This is because he cannot tell whether and when Alice will use key b_{i-1}. Even if Bob receives a message from Alice where she has used key b_{i-1}, he cannot delete it. Assuming that Bob has received Alice's message after he sent his i-th message, it is possible that Alice has not yet received the i-th message and will use key b_{i-1} again for one, two or many more messages until she receives Bob's i-th message.

The approach to safely delete locally generated Diffie-Hellman keys is simple – when a user deletes a peer key upon receiving a new message, he stores the key's ID in a list containing *old* or *deleted* keys. When this user sends his next message, he simply attaches this list of key IDs to the message. His peer can then safely delete all the keys in the list since he knows that they will never be used again.

5.2.5 Key Caching and Certificate Storage

In our implementation there is a lot of information that needs to be stored, foremost the keys and certificates. After first approaches to store this data using text files, we had to change to another approach since the more features we implemented, the more information we needed to keep track of. In the current implementation the data is stored in a persistent SQLite Database. There are four tables containing different information:

- The public part of Diffie-Hellman keys which were received
- The private part of the Diffie-Hellman keys which the client has sent
- The signature keys of other clients
- The certificate, signature and salt of the key exchanges.

The keys are stored as plain text using a hex encoding. We are aware that it would be more save to store some of these keys encrypted, but on the other hand the sensitive keys are thrown away after some time anyway. All of these keys are stored with the user name the key is used with. Additionally, to keep track of which Diffie-Hellman keys can be deleted and which are still in use, we added an ID to the keys. The certificates are stored in the same way which means mostly text based. The certificates that are used for signing (Authentication server) are stored using the *Java Key Store (JKS)*. JKS has some features like passphrases to provide a certain level of security.

Since we implemented the encryption protocol for both a PC client and an Android app, we had to define a common interface for accessing the SQLite databases which both these platforms had to implement.

5.3 Key Synchronization

A user may use multiple devices, for example a phone and a tablet or a phone and a PC. If the user is online with multiple of his devices at the same time, all devices should be able to receive *and* decrypt the same messages. Since we are using an architecture that implements Perfect Forward Secrecy using Diffie-Hellman key agreements the private Diffie-Hellman keys have to be synchronized among the different devices of the same user.

But first, we present a short example. Assume that Alice uses the devices X and Y while Bob uses the device Z and both users have the same generator g and modulo m. If Bob sends a message to Alice, he will use his own private key b and one of Alice's public keys witch is either g^{a_X} or g^{a_Y}, depending on which device has the last message. In case the last message was sent by device X, Bob will compute the secret key $g^{a_X b}$ to encrypt his message. Upon receiving Bob's message device X will have no problems decrypting the message, but device Y will not be able to decrypt it directly. In order to decrypt the message, device X must tell device Y about the key a_X, otherwise, device Y cannot compute the secret key $g^{a_X b}$.

There are several possible approaches to implement synchronization. A first approach would be to use a centralized entity. This could be one of the user's device, preferably a stationary device such as a computer. The centralized entity would coordinate the whole synchronization, including the creation of keys with *unique* key IDs but also the distribution of the keys. However, this approach would result in a heavy load on the coordinator which is certainly not desirable if the user is using multiple mobile devices and no stationary one. The other problem is that the coordinators might disconnect for some reason and leader election would have to be performed leading to non-negligible delays.

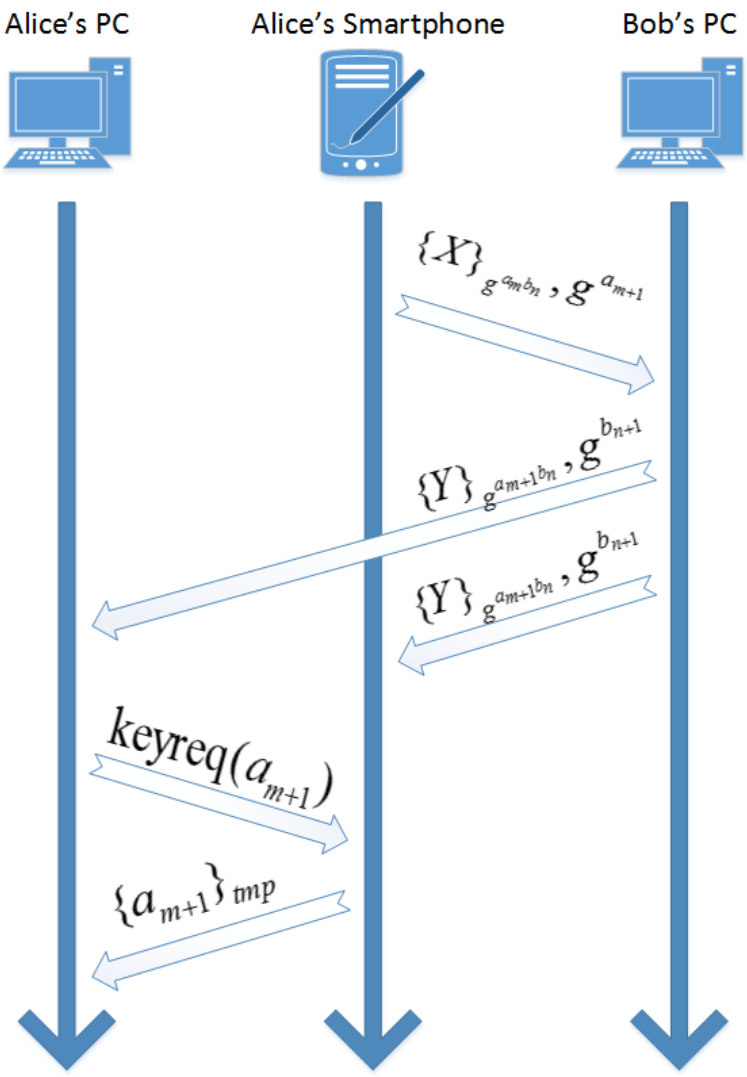


Figure 6: Example of key synchronization

A better approach is to perform synchronization in a truly distributed manner, which is exactly the approach we have chosen. Instead of having a centralized entity, each device computes its own set of keys where the set of key IDs of any two devices are *disjoint*. In our protocol keys are identified by their IDs. This minimizes protocol overhead as key IDs use fewer bytes than keys (e.g., one of Alice's keys might be used multiple times if Bob sends multiple messages to Alice without receiving a message from Alice). Keys generated by different devices of the same user *must* have disjoint key IDs or the messages cannot be decrypted by all devices. If Alice's devices would generate keys with the same IDs, Bob would tell Alice's devices that he used the key with ID k to encrypt the message but for every one of Alice's

devices, the keys with ID i would be different and therefore, only one of the devices would be able to decrypt the message.

To achieve disjointness each device computes its key IDs as follows:

1. Compute unique *offset* D where $0 \leq D < \text{MAX\_CLIENT\_DEVICES}$.
2. Keep a counter i that increases by 1 everytime a new key is generated locally.
3. Key ID k of the i-th key is $k = i \cdot \text{MAX\_CLIENT\_DEVICES} + D$.

Using this scheme we can assure that no two devices compute the same key IDs, assuming each device has a unique offset. In our implementation the offsets are computed by taking the hash code of the `clientId` modulo `MAX_CLIENT_DEVICES`. Note that `clientId` denotes the string that stores the user ID of the user. Since we can assume that a user has no more than 4 or 5 devices (e.g. a phone, a tablet, a notebook and maybe a desktop), we can assume that collisions among the offsets are highly unlikely if `MAX_CLIENT_DEVICES` is set to a high number such as 65536. The computation from the probability of a collision follows from the birthday problem.

We have used the following formula to compute the probability for a collision between n devices given that the maximum number of distinct offsets (`MAX_CLIENT_DEVICES`) is set to 65536.

$$P[\text{Collision}|n] = 1 - P[\text{No collision}|n] = 1 - \frac{65536!}{65536^n(65536-n)!}$$

Assuming that there are at most $n = 5$ devices for any user (which seems reasonable), the probability that two or more devices have the same offset D is less than 0.00015.

Using this scheme a device can determine whether a key ID that has been used to encrypt a message is a key of its own set of keys or whether another device must have generated the key. In case another device has generated the key, a *signed key request message* (including a temporary Diffie-Hellman public key) is broadcasted to all other devices of the same user. If a device receives such a broadcast message, it determines whether it has generated the requested key and if so, it answers the key request or else simply drops the request. Therefore, only the creator of the key will answer (if it has not disconnected).

To answer the key request, the responsible device fetches the key from its key cache (if it has not been deleted to do clean-ups) *and* also pre-fetches a few more keys it is going to use next. Afterwards it generates its own temporary Diffie-Hellman key and combines it with the public DH key that was included in the request to generate a secret key. This key is then used to encrypt the private keys the other device has requested. Obviously this

is necessary since we cannot transfer private DH keys in plaintext. Finally, both temporary keys are discarded on the device that responded to the request and the response is sent back.

After receiving the response Alice's device can then decrypt the private keys, then decrypt Bob's message and cache the keys for further use. As already mentioned a device always receives more than one key. This optimization has been chosen to minimize the number of key requests necessary. After requesting key k, a device will also know about the keys $k + \text{MAX\_CLIENT\_DEVICES}$, $k + 2 * \text{MAX\_CLIENT\_DEVICES}$, $k + 3 * \text{MAX\_CLIENT\_DEVICES}$ and so on.

Another optimization is to cache the responsible device for a certain offset. Initially a device does not know which one of the other devices is responsible for offset o. However, after the first request for a certain offset o, the device knows the responsible device. This knowledge allows us to avoid unnecessary broadcasts by being able to send a request directly to its responsible device.

5.4 Google Cloud Messaging for Android

In the following sections, we present the implementation of the client and the server that use the GCM framework. We also present an implementation that does not use a server.

5.4.1 Client

The client side i.e. the Android application first needs the Google Play Services SDK installed [11]. Additionally, the manifest of the application needs to be adapted. GCM requires the following permissions listed in Listing 4.

Listing 4: GCM permissions

```
1 com.google.android.c2dm.permission.RECEIVE
2 android.permission.INTERNET
3 android.permission.GET_ACCOUNTS
4 android.permission.WAKE_LOCK
5 "applicationPackage"+.permission.C2D_MESSAGE"
6 com.google.android.c2dm.intent.RECEIVE
```

Most of these permissions are defining that the device can receive from GCM, but also which application can receive these messages. This restriction is implemented by GCM by a field called restricted package name. The packets sent with this fields set to a value can only be received by receiver services which have defined the same value as the packets sent.
Additionally to these permissions, a developer has to register his project with Google using the *Google Developer Console* [12]. When a project is registered, it gets a Project Number that will be later used as a Sender ID.

Further for some sort of authentication, an API Key must be generated. As a last step in the developer console, the GCM Service itself must be enabled. The first thing an application has to do if it wants to use GCM is to check if the device is compatible with Google Play Services. After this quick check the device can register itself to the GCM servers using the sender ID. The Google servers will respond with a registration ID. This registration ID specifies for GCM to which device a message is sent to. The GCM registration ID is then stored in the shared preferences and needs to be kept secret. The only other application that needs to know the registration ID, is the second part of the implementation, the server. The server stores the registration IDs and associates them with some other sort of identification like the username.
The last part that needs to be implemented on the client side, is the receiving part itself. Googles Framework offers a class called *WakefulBroadcastReceiver* which can be extended by just implementing a method called `onReceive()`. This method gets an Intent as a parameter which contains in the extras the message sent.

5.4.2 Server

The server has two main tasks: store the GCM registration IDs and associate them with a username/account name and communicate to the Google servers.
The first task -storing the registration IDs- is necessary because a messenger client does not know the registration ID of the device it sends to. This means in order to let two device communicate, the server needs the registration ID of both device. To send a message the device will send a message in a format the server understands, to the server. This message will contain some sort of username/account name which the server then can translate into a registration ID.
A server can communicate in two ways with the Google server: HTTP or CCS (XMPP). CCS has the benefits over HTTP of bidirectional asynchronous communication channels which also allow the device itself to communicate with the Google servers for upstream messages to the server. One of the problems with using CCS is the requirement to unlock that CCS can actually be used. This can take up to 3 months and thus it was clear, that for the beginning we used HTTP. An example of an HTTP request to the Google servers is shown in Listing 5.

Listing 5: GCM request

```
Content-Type: application/json
Authorization: key=AIzaSyB-1uEai2WiUapxCs2Q0GZYzPu7...
{ "collapse_key": "score_update",
  "time_to_live": 360000,
  "delay_while_idle": true,
  "data": {
    "message": "hello",
  },
  "registration_ids":["APA91bHun4MxP5egoKMwt2KZFBaFUH-1RYqx..."]
  "restricted_package_name": "ch.ethz.disco.p2pmessenger.Client"
}
```

The first few lines are the header that contains the authorization key which was previously obtained from the Google Developer Console (API key) and the type of the content, in the shown case JSON. The parameters of the JSON object are:

- `collapse_key`: An arbitrary string that is used to collapse messages to avoid sending too many messages in the case when a device was offline and then comes online. With this key a group of messages is delivered together and not on their own if several messages were queued for a device.
- `time_to_live`: Specifies how long the Google servers should store a message if it can not be delivered.
- `delay_while_idle`: Specifies if the Google servers should wait until the device is active and not idle.
- `data`: The actual data to be sent. This is a JSON object itself and can have a size of up to 4kb.
- `registration_ids`: These are the registration IDs to which the message should be sent to. Can contain multiple addresses.
- `restricted_package_name`: This is a string containing the package name of the application. This prevents that messages are sent to registration IDs that are not registered with this package name.

In this format the server communicates with the Google servers. If there is an error with this format, there is a lot of information in the response to the request helping the developer debug. Additionally to the format, Google suggests that some kind of back-off mechanism is implemented when sending a request fails.

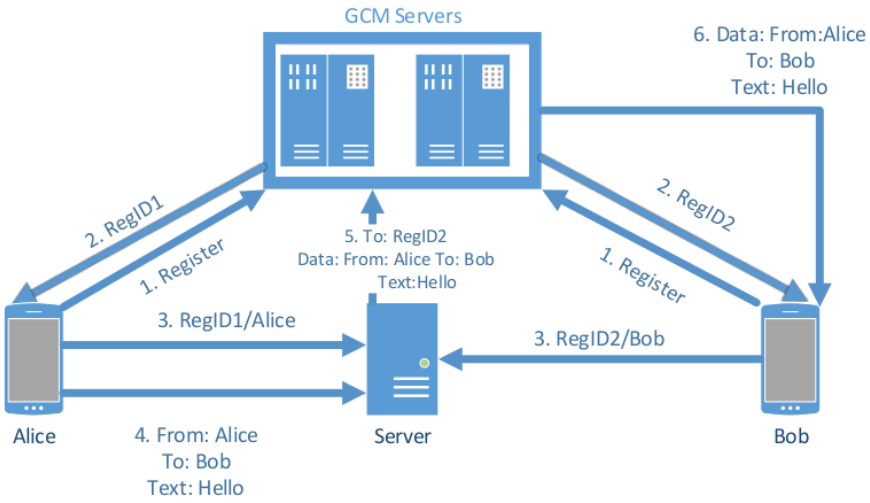


Figure 7: Sending a message using GCM with standard implementation

5.4.3 Implementation

Our first approach to the implementation was as described in the previous subsections. The problem with implementing GCM in this way is, that it introduces a single point of failure with the server. Our approach to solve this problem was, to implement most of it in the peer-to-peer network. At first the messenger client registers itself with the Google servers and gets a registration ID. Instead of sending this ID to our server, we send the ID to the client's parents in the network. Every client has a certain amount of parents in the network. The parents are the gateway for clients to communicate over the overlay network. When the parents receive an ID, they store this ID and the username and resource it is associated to. When a client wants to send a message, the messages are first sent to the parent of the receiver. Instead of sending this message to the device, the peer will check if it has a registration ID store for the receiver, and send it to the GCM servers which will deliver the message later. The receiving service extends the class *WakefulBroadcastReceiver*. When this receiver gets a message from the Google servers, it hands the intent to a `IntentService` which processes the received message.

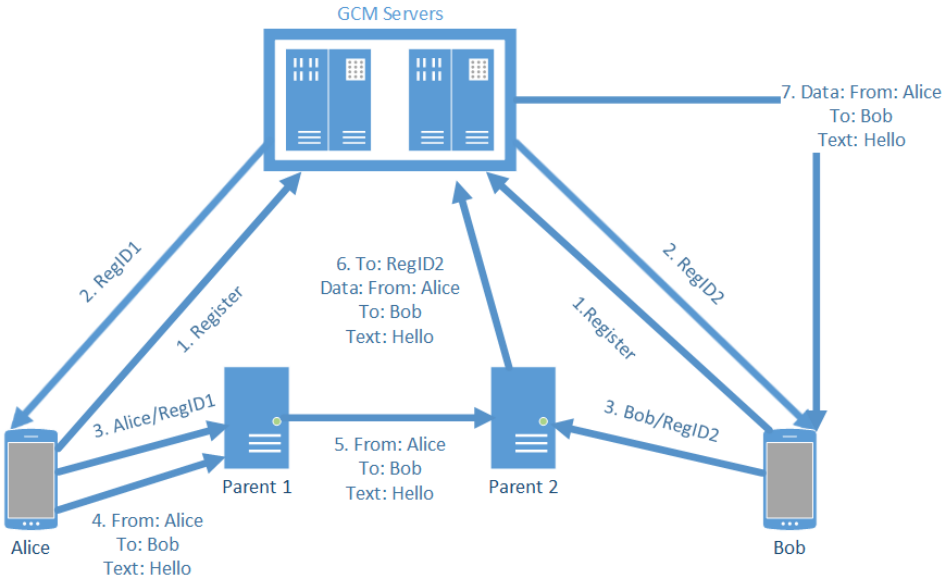


Figure 8: Sending a message using GCM without centralized server

6 Summary

We extended an already existing Peer-to-Peer instant messenger to support encryption and therefore authenticity, integrity and confidentiality. Furthermore, we implemented encryption in such a way that Perfect Forward Secrecy can be guaranteed even if a user uses multiple devices at once. We let ourselves inspire by SSL when creating a hierarchical chain of trust to issue certificates to users of the messengers. We also provided building bricks for future implementations of secure file transfer.

However, there is yet much to be done concerning the P2P messenger until it can finally be released into the Google Play Store and such. We are looking forward to see further progress of the P2P messenger.

7 Acknowledgements

Finally, we would like to thank Tobias Langner and Philipp Brandes for the (usually) weekly meetings and their constant support.

A References

[1] Olafsdottir, H.: Peer-to-peer based instant messenger. (2014)

[2] Bourchas, T.: Distributed hashtables for p2p-messenger. (2014)

[3] Jabber: Jabber instant-messenger-client http://www.jabber.org/.

[4] (IETF), I.E.T.F.: Xmpp (rfc 6120) http://tools.ietf.org/html/rfc6120.

[5] Borisov, N., Goldberg, I., Brewer, E.: Off-the-record communication, or, why not to use pgp. In: WPES '04 Proceedings of the 2004 ACM workshop on Privacy in the electronic society, New York, NY, USA. (2004)

[6] Wikipedia: Advanced encryption standard http://de.wikipedia.org/wiki/Advanced_Encryption_Standard.

[7] WhatsApp: Whatsapp http://www.whatsapp.com.

[8] Farb, M., Lin, Y.H., Kim, T.H.J., McCune, J., Perrig, A.: Safeslinger: Easy-to-use and secure public-key exchange. In: Proceedings of the 19th Annual International Conference on Mobile Computing & Networking. MobiCom '13, New York, NY, USA, ACM (2013) 417–428

[9] (IETF), I.E.T.F.: Internet x.509 public key infrastructure certificate and certificate revocation list (crl) profile http://tools.ietf.org/html/rfc5280.

[10] Wikipedia: Block cipher mode of operation http://en.wikipedia.org/wiki/Block_cipher_mode_of_operation.

[11] Google: Google play services https://developer.android.com/google/play-services/index.html.

[12] Google: Google developer console https://developer.android.com/distribute/googleplay/developer-console.html.