End-to-End Data Protection of SMS Messages

Jo Mehmet Sollihagen Øztarman

Master of Science in Communication Technology
Submission date: June 2011
Supervisor: Stig Frode Mjølsnes, ITEM
Co-supervisor: Anton Stolbunov, ITEM

Norwegian University of Science and Technology
Department of Telematics

Project description

Modern 2G, 3G and future LTE mobile phone network standards do not provide end-to-end security. Both the visiting and home network operators have to be trusted for the generation and possession of cryptographic keys used for access control and wireless radio communication link security. Many security issues have been identified with 2G networks, like for example the broken A5/1 and A5/2 ciphers and the lack of mutual authentication in GSM.

The aim of this project is to design an enhanced Short Message Service (SMS) that provides end-to-end data protection using public key cryptography. The end-to-end security should include confidentiality, authenticity and replay detection by selecting or designing the best-suited public key cryptography scheme for the problem. The cryptographic signaling overhead required may be a challenge due to the very limited size of maximum 140 bytes per single message. SMS allows increasing the length by chaining up to 255 messages of 134 bytes each, but this will increase the communication cost proportionally. Another challenge may be the computational power available for the cryptographic functions. If time allows, essential parts of the proposed solution should be verified by implementation and testing. The implementation and testing will be done on the available Android platform.

Assignment given: 24.01-2011
Student: Jo Mehmet Sollihagen Øztarman
Supervisors: Anton Stolbunov and Stig Frode Mjølsnes
Professor: Stig Frode Mjølsnes

Abstract

Short Message Service (SMS) has become a very commonly used service. It does not only work as a substitute for voice telephony, but is also used for automated services. Some of these service are related to security issues like SMS banking, or one time passwords, even though SMS messages can be spoofed or eavesdropped.

We propose a design where we add security to SMS by making an easily configurable module that utilizes a fast cryptographic scheme called Elliptic Curve Signcryption. To prove our concept, we implement an SMS client for Android smart phones that utilizes our security module and serves end-to-end data protection of SMS messages with the same security level as *Top Secret* content.

Preface

This Master's Thesis is the result of a project completed within the field of Information Security in the 10th semester of the Master's Program in Communication Technology at The Norwegian University of Science and Technology, NTNU.

I would like to thank my supervisor, Anton Stolbunov very much for all valuable feedback and guidance in our meetings throughout the project and especially the very last meeting where he gave me critical questions and extra motivation for the project.

I would like to thank Professor Stig Frode Mjølsnes who gave me a very interesting and fun starting point defining the concept of the project.

Many thanks to all my fellow students who contributed proposals for functionality, volunteered in usability testing and listened passionately to me when I talked about my thesis during lunch breaks. In particular I am very grateful to Eirik Stien who gave me a crash course in programming Android apps.

Special thanks to my family who always supports me, and for their contribution to the English orthography in this report.

Abbreviations

AES	Advanced Encryption Standard
ADT	Android Development Tools
API	Application Programing Interface
app	application program
BTS	Base Transceiver Station
CBC	Cipher Block Chaining
DoS	Denial of Service
DSA	Digital Signature Algorithm
DSS	Digital Signature Standard
EC	Elliptic Curve
FIPS	Federal Information Processing Standards
GCD	Greatest Common Divisor
GSM	Global System for Mobile Communications
GUI	Graphical User Interface
IJCSNS	International Journal of Computer Science and Network Security
IV	Initialization Vector
JDK	Java Developer Kit
JRE	Java Runtime Environment
MIME	Multipurpose Internet Mail Extensions
MITM	Man-In-The-Middle

MS	Mobile Subscriber
MSISDN	Mobile Subscriber ISDN Number
MVC	Model-View-Controller
NSA	National Security Agency
OS	Operating System
PSTN	Public Switched Telephone Network
RSA	Rivest, Shamir and Adleman
SDK	System Developer Kit
SMS	Short Message Service
SMSC	Short Message Service Center
SHA	Secure Hash Algorithm
SHS	Secure Hash Standard
TE	Terminal Equipment
URI	Uniform Resource Identifier
UML	Unified Modeling Language
XML	Extensible Markup Language

Definitions

Authenticity

The assurance that the communicating entity is the one that it claims to be.

Confidentiality

Protection of data from unauthorized disclosure.

Fingerprint

A hash value that with very high probability identifies an amount of data.

Forward secrecy

Compromise of the current key should not compromise any future key.

Reply detection

Detection of the uniqueness of a message to ensure it has not been copied and replied.

Top Secret

The highest level of classification of material on a national level. Such material would cause ”exceptionally grave damage” to national security if made publicly available.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 History of SMS

SMS was defined as a part of the Global System for Mobile Communications (GSM) series of standards in 1985[rev85] as a means of sending text messages of up to 160 characters. SMS was commercially deployed by different mobile operators around 1993 as a service to inform the subscribers of new voice mail messages received. Support for sending messages person-to-person was introduced at a later stage, but the initial growth of SMS messaging was slow. In the decade of 2000-2010 the growth was enormous and today SMS messaging is commonly used for many purposes.

1.2 Usage of SMS

During the recent years, there have been few limitations on the services offered through SMS. People text each other instead of talking. Companies remind you of appointments, or inform you about the current status of package deliveries. Value added services are provided where you can buy music, games, metro tickets and even soda water from vending machines just by sending a code to a short code SMS number, which is much shorter than normal phone number.

Banks offer SMS Banking, where you may apply for loans, transfer money and receive your account balance. If we look at the governmental usage in Norway; in 2008 it became possible to confirm the tax return form by SMS. In Norway, when logging in to online public services, four levels of security are offered. For logging in with security level 3 of 4 you will receive a secret code by SMS message, which is defined as sufficient to complete a change of the address in the national address registry online[dif11].

There are systems facilitating control of one's home, e.g. control of the house alarm[fra11] or for example the heating, the possibilities are only limited by the number of existing electrical devices.

1.3 Security in SMS

By using some of the services we described in section 1.2, *Usage of SMS*, we can say that the public has confidence in the SMS services. Generally people do not consider it feasible to spoof nor eavesdrop SMS messages sent and received. This may be influenced by the attitude shown by big stakeholders, like banks and governmental institutions, since thse commonly deploy safety features through SMS. They communicate SMS to be secure by utilizing the technology in security contexts.

SMS messages only rely on the security provided by GSM, which only includes the encrypted radio link between the Terminal Equipment (TE) and the Base Transceiver Station (BTS). Once the content passes the BTS to the inner network, nothing is encrypted anymore. This means that anyone with access to the inner network may eavesdrop the content, and operators may also alter the content since messages are processed in a “store and forward” fashion.

The sender ID in SMS messages is an alphanumeric or numeric value that is set by the mobile operator using the Mobile Subscriber ISDN Number (MSISDN) that identifies a unique customer on a global basis. With SMS gateways like SMS Global[SMS11] connected to the Public Switched Telephone Network (PSTN), this value can be set manually and opens for the possibility to spoof the originator of an SMS message.

In chapter 2, *Motivation*, we will show examples on how these security flaws can be misused, by designing attacks that may potentially be realized by anyone.

1.4 Background

This Master Thesis has its background in the preliminary project *End-to-End Confidentiality and Integrity Protection of Short Message Service Messages over Cellular Wireless Networks*, a specialization project in Information Security in the 9th semester of the Master’s Programme in Communication Technology at The Norwegian University of Science and Technology, NTNU, written by Petter Andreas Strøm[Str10].

The reader should have basic knowledge in Elliptic Curve (EC) arithmetics and cryptography, this can be acquired in [Sta05, sections 10.3 - 10.4]

Object Oriented programming skills in Java will be necessary to understand the implementation found in the Appendix, but is not required to understand the main parts of the Thesis.

1.5 Limitations in this project

An important security problem of sending protected SMS messages is to have a proper key exchange. A solution to this problem has been analyzed and proposed in [Str10] and will for that reason not be analyzed any further, but will introduced if there is time for a complete implementation.

1.6 Methods

To achieve the anticipated outcome in this project a list of methods is set up as follows:

- Obtain domain knowledge
- Find the most feasible layer to protect SMS messaging
- Analyze how to apply the security services defined in the project description
- Learn how similar applications for smart phones work
- Design the system
- Learn the basic development environment for the selected platform by guides, tutorials and from colleagues
- Code and test iteratively
- Obtain feedback and perform revisions

1.7 Outline

In chapter 2, *Motivation*, we show examples of how you may become the victim of fraud by relying on the non existing security found in SMS. In chapter 3, *Analysis*, we will analyze our domain and security mechanisms to make choices that suit our problem in a best possible way. In chapter 4, *Design*, we will design a complete system to accommodate our problem by utilizing important results from the Analysis. In chapter 5, *Implementation*, we will realize a solution by programming. In chapter 6, *Results*, we will show what we have achieved by the process of analyzing, designing and implementing a solution for end-to-end protected SMS messaging. In chapter 7, *Discussion*, we will discuss some important choices made in our solution, point out some possible security flaws and compare our solution to other existing solutions. In chapter 8, *Conclusion*, a summary of what we have achieved in this Master Thesis is presented, as well as some ideas for how to bring our solution to a higher level. In the Appendices found at the end, there will be direct references to the implementation source code added in the zip-file attachment that can be found by searching for this Master Thesis at `http://daim.idi.ntnu.no`. The same code will also be open source at `http://github.com/jomehmet` and can be checked out as Eclipse projects by anyone.

Chapter 2

Motivation

This chapter intends to provide the motivational aspects for protecting SMS messages, by clarifying the lack of security in SMS through a number of practical examples of attacks.

2.1 SMS spoofing

The term SMS spoofing means to change the sender ID in an SMS message. This has been tested through an Australian SMS gateway called SMSGlobal[SMS11]. As pictured in figure 2.1 it is possible through their web interface to send SMS messages and choose any sender ID. This section shows the importance of authenticity in SMS messages.

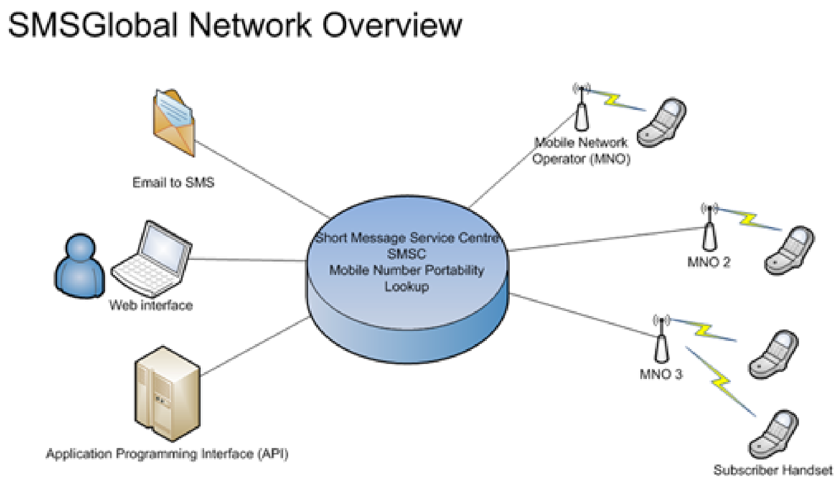

SMSGlobal Network Overview
Email to SMS
Web interface
Application Programming Interface (API)
Short Message Service Centre
SMSC
Mobile Number Portability
Lookup
Mobile Network
Operator (MNO)
MNO 2
MNO 3
Subscriber Handset

Figure 2.1: Overview of the SMSGlobal network[SMS11]

2.1.1 Example of SMS spoofing attack

We have constructed an example attack that shows how to lure necessary credentials from a bank customer and make a phone call to the bank using the customers identity. The Scandinavian bank Skandiabanken has been chosen as an example bank, but the attack could be modified to any other bank.

2.1.1.1 Signing up to the SMS gateway SMSGlobal

For signing up, SMSGlobal only requires name, mobile phone number and e-mail address and you may send 25 free SMS messages to test their platform. This means that people with criminal intentions may be totally anonymous, using an unregistered pre paid mobile subscription number.

2.1.1.2 Choosing a victim

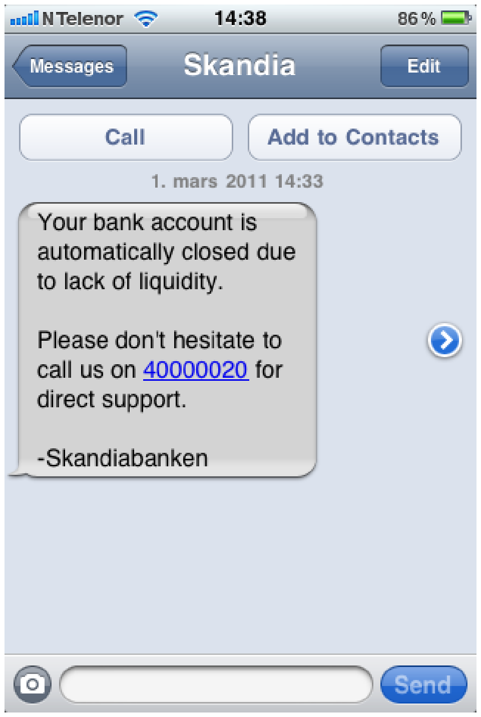
Before you can initiate an attack you need to know the phone number of your victim. Nowadays most banks have a connection to some kind of social media, for example a page on Facebook[fac11]. On these kinds of pages you get the list of people who are followers, giving their full name. Most likely these followers are customers of the bank as well. With a lookup on their social media profile their phone number may be public, or their phone number may be found in a public phone directory.

2.1.1.3 Send the victim a spoofed message

To send a Spoofed SMS message with SMSGlobal, their web interface is used as shown in figure 2.3. Their system allows the sender ID to be at most 11 alphanumeric characters, or 16 numeric characters. The spoofed sender ID ”Skandia” can be chosen as a shortening for the more than 11 letter word Skandiabanken. The message can be as shown in figure 2.2, which would most likely have a lot of customers call the given phone number. The chosen number is easy to remember and looks like a company number, but is a Norwegian cell phone number. Thus, the number is obtainable for a private individual.

2.1.1.4 Get the credentials from the victim

The attacker needs to have made a test call to the victim bank to know all the security questions. If the victim does not know the phone number of the bank by heart, the victim will probably call the given fake support number controlled by the attacker. When the victim calls, all the normal control questions of the bank should be asked and noted. Then the attacker may say that everything is all right and that the SMS message was sent in error.



1. mars 2011 14:33

Your bank account is automatically closed due to lack of liquidity.

Please don't hesitate to call us on 40000020 for direct support.

-Skandiabanken

Figure 2.2: Screenshot of spoofed message from SMSGlobal

2.1.1.5 Get authenticated using the credentials of the victim

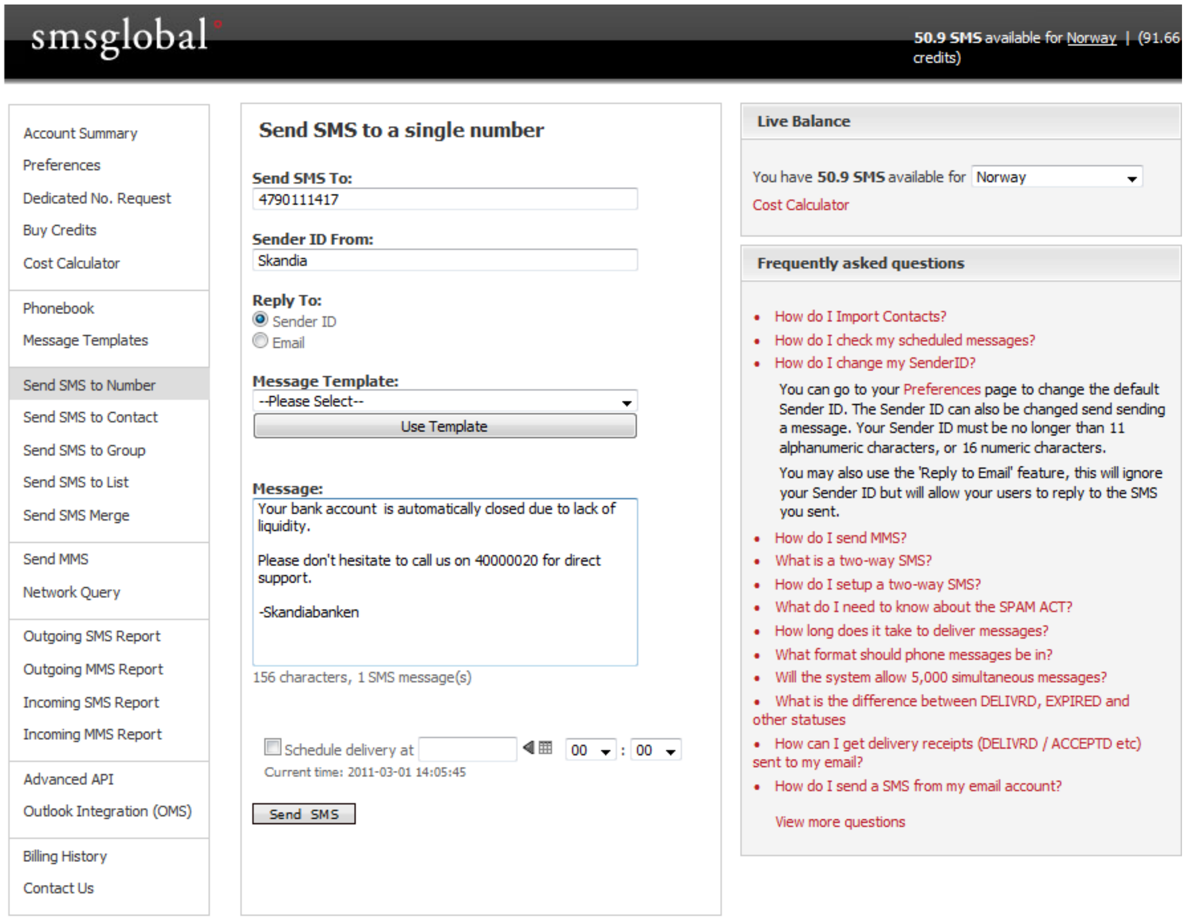
As the attacker now is in the possession of the customer's credentials, the attacker may now call the victim bank and be authenticated as the victim. As an authenticated customer the attacker may now demand all the services the bank offers by phone.

2.2 SMS eavesdropping

SMS is very often sent over GSM that lacks mutual authentication. The Mobile Subscriber (MS) is authenticated by the BTS, but the MS does not authenticate the BTS. This implies the possibility of a Man-In-The-Middle (MITM) attack where you have a fake BTS between the MS and the BTS. This type of attack has already been implemented and is sold as a portable box called IMSI-catcher NS-17-1[AS11] and makes it easy for anyone to eavesdrop the content of a GSM conversation, including SMS messages.

2.2.1 Steal one time password by eavesdropping SMS

Let us say an attacker is in possession of a customer's user name and password in an Internet bank that uses one time SMS passwords as extra security. The IMSI-catcher



Send SMS to a single number

Send SMS To: 4790111417

Sender ID From: Skandia

Reply To: [✓ Sender ID] Email

Message Template: --Please Select--

Message:
Your bank account is automatically closed due to lack of liquidity.

Please don't hesitate to call us on 40000020 for direct support.

-Skandiabanken

156 characters, 1 SMS message(s)

Schedule delivery at 00 : 00
Current time: 2011-03-01 14:05:45

You have 50.9 SMS available for Norway

- How do I Import Contacts?
- How do I check my scheduled messages?
- How do I change my SenderID?

  You can go to your Preferences page to change the default Sender ID. The Sender ID can also be changed send sending a message. Your Sender ID must be no longer than 11 alphanumeric characters, or 16 numeric characters.

  You may also use the 'Reply to Email' feature, this will ignore your Sender ID but will allow your users to reply to the SMS you sent.
- How do I send MMS?
- What is a two-way SMS?
- How do I setup a two-way SMS?
- What do I need to know about the SPAM ACT?
- How long does it take to deliver messages?
- What format should phone messages be in?
- Will the system allow 5,000 simultaneous messages?
- What is the difference between DELIVRD, EXPIRED and other statuses
- How can I get delivery receipts (DELIVRD / ACCEPTD etc) sent to my email?
- How do I send a SMS from my email account?

Figure 2.3: Test of sending spoof message with the web interface of SMSGlobal[SMS11]

NS-17-1 is portable and camouflaged as a business briefcase with capabilities to be remotely controlled over wifi. The attacker can place the IMSI-catcher in a location where the victim will be residing for a period of time. At a time when it is unlikely that the victim is paying any attention to his phone, the attacker can login and eavesdrop all one time passwords needed, e.g. to transfer money. The specifications of the IMSI-catcher do not say if it is possible to block SMS messages, but in theory the attacker could also block these one time password messages to hide the activity.

As we now have seen the potential economical consequences of having insecure SMS services that give false security, we will now go on to analyze what the best possible way to protect SMS messaging could be.

Chapter 3

Analysis

In this chapter we will analyze the relevant factors pertaining to the problem of securing SMS-messages and the reasons for cryptographic choices to serve authenticity, confidentiality and replay detection. In section 3.1, *Choice of Layer*, we discuss a justification for the choice of layer into which to implement our solution. In section 3.2, *Message size constraints*, we analyze limitations on the amount of data that can be transferred between the entities. In section 3.3, *Confidentiality*, we find an encryption scheme that prevents the messages from being eavesdropped without spending to much overhead. In section 3.4, *Authentication*, we look at common techniques for verifying message authenticity. In section 3.5, *Signcryption*, we look at a cryptographic new scheme that saves computational power and overhead, by adding authentication and confidentiality into one logical prime. In section 3.6, *Key size*, we justify the choice of key sizes to be used, to ensure sufficient security for any private or public institution that wants to use the system. In section 3.7, *Replay detection*, we analyze potential replay attacks that can be applied to secure SMS messages, and how we can detect them. In section 3.8, *Message encoding*, we look at an encoding technique called Base-64 and its successor Ascii85 that can help us to send byte messages longer than 133 bytes over SMS. At the end in section 3.9, *Summary*, we list the most important security parameters we have obtained by systematically analyzing the sub problems of securing an end-to-end SMS messaging system.

3.1 Choice of Layer

Our goal is to add security to an already globally established and standardized communication service. A good solution would be to change the standard in the lower layer of the protocol stack, which already has been proposed by Toorani et.al[TBS08]. This would imply that the end user would not notice the changes, nor need to make any action for it to work.

Unfortunately, changing specifications of such an established standard is a very long and tough process. This can be learned by looking at the deployment of IPv6, which is the successor of IPv4. IPv6 was specified for the first time in 1998 in RFC 2460[DH98]. A report from 2009, by AT&T Labs[KGP$^+$09], 11 years after RFC 2460 came out, concludes that although IPv6 is growing, there is still a huge lack of applications using it. This indicates how difficult it would be to change the standard defining SMS.

In this project we want to provide an end-to-end secured SMS implementation that can be used immediately between smart phones equipped with mobile subscription numbers. Independently from the operator, this is obtainable in the highest layer of the SMS protocol stack, the application layer. In the application layer, the only two parameters that can be changed are the recipient's phone number and the text message to be sent. Since the phone number is needed for the lower layer to work, we need to focus on the text to be sent.

3.2 Message size constraints

As we saw in section 3.1, *Choice of Layer*, the only parameters we can change for securing an SMS message in the application layer is the text to be sent. This value is limited to 140 characters if sent with UTF-8 encoding[Inc11e] or 160 characters with ASCII encoding[Cer69]. Since ASCII is a subset of UTF-8 it allows more characters in a message. The encoding of SMS messages is chosen dependently of what characters a message contains. If one or more non-ASCII characters are used, the message will be UTF-8 formatted. The standard also has support for concatenation of messages longer than 140 or 160 characters, thus every part of a message may contain up to 134 UTF-8 characters or 153 characters if it is ASCII encoded. The upper limit of concatenated messages is 255, and every part of a message will cost the same as a single message.

As we want to keep the extra cost as close to zero as possible, we will need to focus on keeping potential overhead as small as possible.

3.3 Confidentiality

Confidentiality is protection of data from unauthorized disclosure[Sta05]. To obtain this we need to encrypt, which means that the content will be transformed into a form that is not readily intelligible. In modern cryptography there are two main approaches for this; with symmetric or asymmetric cipher. The goal of this project is to use public key cryptography for end-to-end data protection. Before looking at public key cryptography we will look at some of the properties that may be provided by symmetric cipher.

3.3.1 Symmetric encryption

In symmetric cipher utilizes one shared key to do both the encryption and decryption as shown in figure 3.1. This means that the secret key has to be shared through a secure

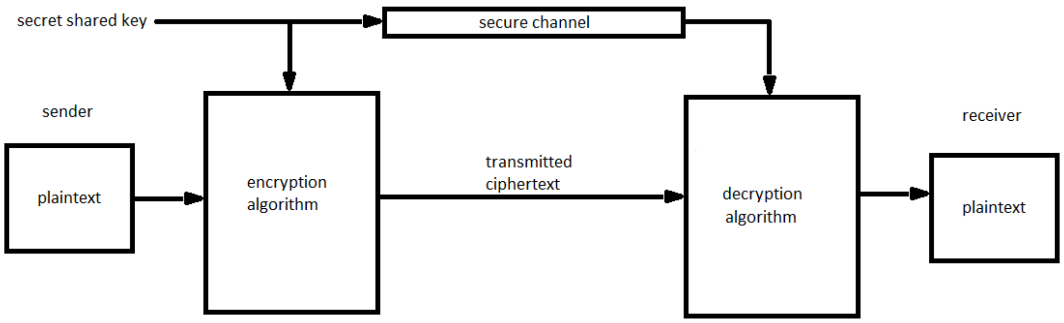


Figure 3.1: Basic model of symmetric encryption scheme

channel which in the case of communicating over GSM is not available. If an attacker deduces the key, all future and past messages are compromised.

Considering commercial solutions that are free of charge, the 128 bits symmetric block cipher Advanced Encryption Standard (AES) which is approved by the National Security Agency (NSA) for *Top Secret* information would be a good selection. This is because it has been proven to show good performance and also has no overhead other than a shared secret key of 128, 192 or 256 bits. If two individuals share a secret key, there is no waste of characters in the encrypted message if the blocks are filled up. Sending UTF-8 characters means 8 bits per character, thus every block of 128 bits may contain 16 characters. If for example 17 characters are sent, two AES blocks are needed and we get a waste of 15 characters.

In spite of the problems of key sharing, the positive properties of almost no overhead and fast computation in the symmetric block cipher are worthy of being kept in mind in the further analysis.

3.3.2 Public key encryption

Public key cryptography also known as asymmetric cipher is an encipherment and digital signature mechanism that provides confidentiality, data origin authentication and nonrepudiation.

Public key cryptography works better when no secure channel is available. Instead of having a shared secret key between the users, every user has a key-pair. The key-pair is based on a public key and a private key. As shown in figure 3.2, the private key can decrypt content encrypted with the corresponding public key. It is also possible to do it the other way around, so that the public key decrypts content encrypted with the private key, but that is commonly used for other services than confidentiality. Since the

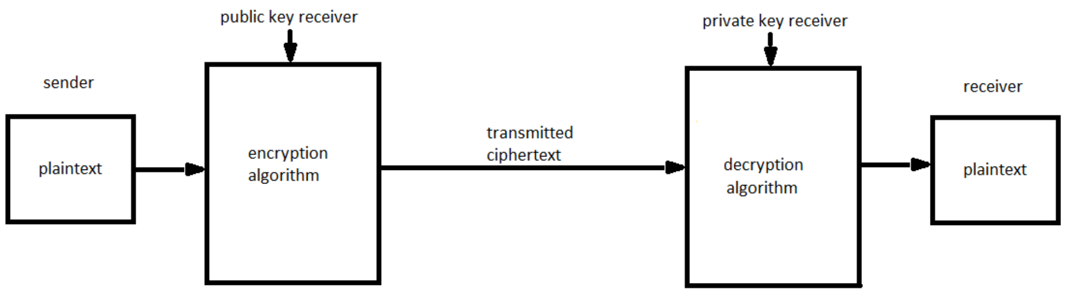


Figure 3.2: Basic model of public key encryption scheme

public key can be known to anyone, there is no need to have a secure channel for key exchange. The private key is only known by the key-pair owner. This means that if the private key is kept in a secure manner, the private key is vulnerable for being ”stolen” only based on the key size, the cipher strength and the choice of key.

A security concern when it comes to publishing the public key is that an attacker may forge a fake public key to be owned by the victim. This threat can be met by authentication of the key by comparing with its key fingerprint. The comparing may be done in real life, by voice recognition over telephone or authentication from a trusted third party[Str10]. In our system we do not want to use a trusted third party, since some of the goal is to avoid the operator as a trusted third party.

Asymmetric cipher consumes more computational power and to achieve the same security level, a larger key is needed. In the most common public key scheme Rivest, Shamir and Adleman (RSA), the key size needed to obtain top secret classification is 7680 bits[NIS11]. These 960 octets would need at least 8 SMS messages just to share one public key.

During the last decade EC cryptography has been added to the Federal Information Processing Standards (FIPS). This scheme needs a much shorter key for achieving the same security level. While 7680 bits is the minimum key length for top secret classification in RSA, EC only needs a minimum key length of 384 bits[NSA10].

The computational cost and the key size is higher with asymmetric cipher than with symmetric cipher, even with EC. But since public key cryptography solves the key exchange problem of symmetric encryption, we need to use it.

3.3.3 Combination of symmetric- and public key encryption

A combined way to use symmetric and public key encryption is shown in figure 3.3, a hybrid solution. In such a hybrid solution, public key encryption is used to exchange

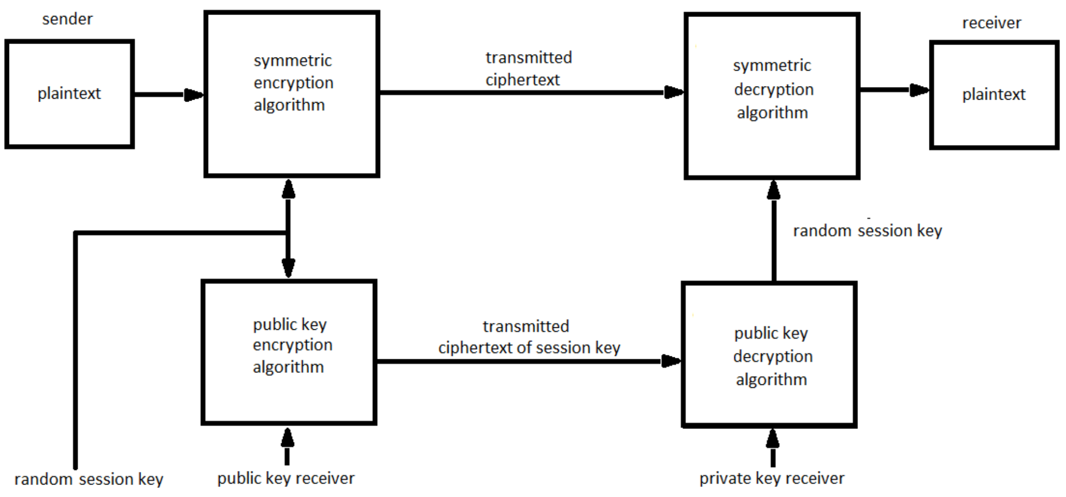

sender
plaintext
symmetric encryption algorithm
transmitted ciphertext
symmetric decryption algorithm
receiver
plaintext
random session key
public key encryption algorithm
transmitted ciphertext of session key
public key decryption algorithm
random session key
public key receiver
private key receiver

Figure 3.3: Basic model of hybrid symmetric- and public key encryption

a session key. The secret session key is then used for symmetric encryption to encrypt the payload. With this combination we get the property of key exchange by public key cryptography, and the low computational cost and overhead of symmetric encryption for the payload.

To sum up what we could use to achieve confidentiality, EC cryptography gives a scheme for key exchange with a minimum key length of 384 bits, and AES with a session key size of 256 bits would with a gracefully small overhead gives top secret classification with regard to [NSA10].

3.4 Authentication

Authentication is the assurance that the communicating entity is what it claims to be[Sta05, p.17].

In section 2.1.1, *SMS spoofing attack*, we saw an example of an SMS spoofing attack, in which the sender can be forged to be anyone and in section 3.3.2, *Public key encryption*, we saw that public key cryptography accommodates a digital signature mechanism.

In this section we will look at the possibilities for using public key cryptography for authenticating SMS messages. We used the public key of the receiver to encrypt the message for confidentiality. Since the key is public, the receiver does not know its origin.

To authenticate the origin, the sender has to use his private key and not the public key. We will look at two different configurations in which the private key is used to sign the message for authenticity.

3.4.1 Authentication by encryption

Before encrypting with the receiver's public key, we encrypt the message with the sender's private key. This way the authenticity lies in the fact that if the ciphertext is successfully decrypted with the sender's public key, the sender is confirmed to be the one claimed to be.

This configuration is not power efficient since it will consume twice the number of encryption courses needed as compared to the number needed for the confidentiality mechanism only. This model is shown in figure 3.4.

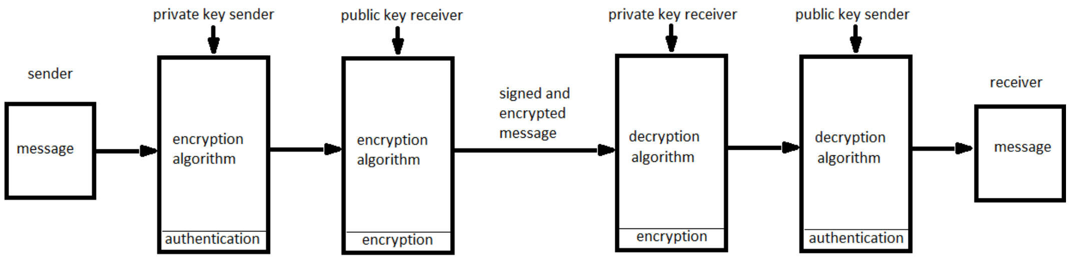


Figure 3.4: Model of authentication by encryption

3.4.2 Authentication by hashing

Another possibility would be to add a message digest as a hash value to the message for proving its authenticity. In figure 3.5 we show a scheme where the sender encrypts the hash value of the message with his private key. This value is called a message digest. This value is further concatenated with the message before it gets encrypted as in the hybrid scheme of section 3.3.3, *Combination of symmetric- and public key encryption*. On the receiver side, the hash value of the message can only be extracted from the message digest with the public key of the sender. If the hash value is compared to the same value as the hash of the decrypted message, the message is authenticated. This is the way defined in the Digital Signature Standard (DSS)[NIS09] together with a hash algorithm defined in the Secure Hash Standard (SHS)[NIS08]. The DSS further says:

> The security strength associated with the ECDSA digital signature process is no greater than the minimum of the security strength associated with the bit length of n and the security strength of the hash function that is employed. Both the security strength of the hash function used and the

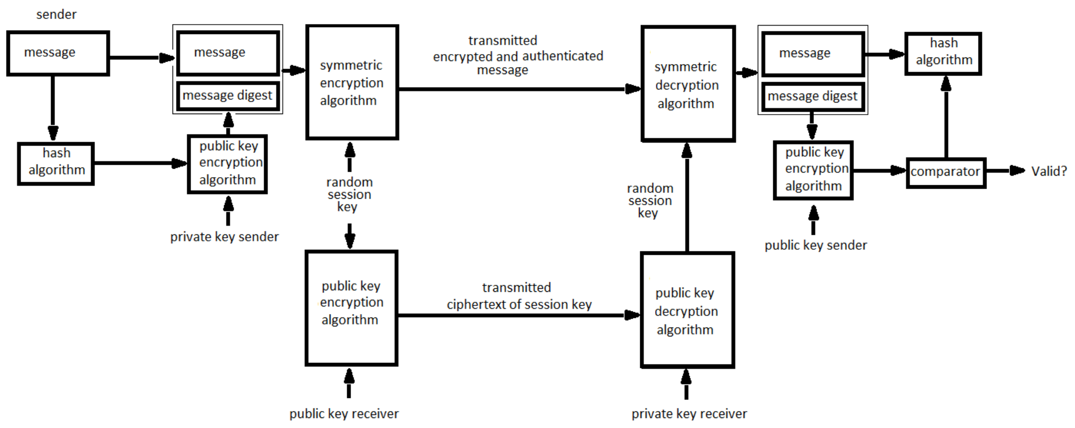

sender
message
message
message digest
hash algorithm
public key encryption algorithm
private key sender
symmetric encryption algorithm
random session key
public key encryption algorithm
public key receiver
transmitted encrypted and authenticated message
transmitted ciphertext of session key
symmetric decryption algorithm
random session key
public key decryption algorithm
private key receiver
message
message digest
hash algorithm
public key encryption algorithm
public key sender
comparator
Valid?

Figure 3.5: Model of authentication by adding a message digest to the hybrid model

> security strength associated with the bit length of n shall meet or exceed the security strength required for the digital signature process.

This means we have to choose the SHA-384 algorithm, since in Section 3.3.3, *Combination of symmetric- and public key encryption*, we considered EC with 384 bits key to be an appropriate public key encryption scheme to classify as top secret.

3.5 Signcryption

In figure 3.5 we saw a common cryptographic scheme of first signature then encryption. This scheme uses a large amount of computational power, since it consists of two cryptographic primitives.

In 1997, Yuliang Zheng for the first time presented a new approach named Signcryption [Zhe97]. The new way was to make the signature and encryption into one logical primitive. This approach decreases the computational power and the overhead compared to existing hybrid schemes. The most interesting aspect of Yuliang Zheng's article from 1998[ZI98] is that he shows an EC based signcryption scheme that saves 58% computational cost and 40% overhead compared to signature-then-encryption schemes based on EC. After this solution was developed, it has been proposed many similar schemes, each giving different security services.

3.5.1 An EC based signcryption scheme

In 2009 an EC signcryption scheme[ME09] was proposed in the International Journal of Computer Science and Network Security (IJCSNS) that adds forward secrecy, public verifiability and encrypted message authentication to the original scheme from Zheng. The scheme gives the following services:

- **Unforgeability**: It is computationally infeasible to forge a valid signcrypted text and claim that it is coming from Alice without having Alice's private key.
- **Non-repudiation**: If the sender Alice denies that she sent the signcrypted text, any third party can run a verification procedure to check that the message came from Alice.
- **Public verifiability**: Verification requires knowing only Alice's public key. All public keys are assumed to be available to all system users through a certification authority or by open publication. The receiver of the message does not need to engage in a zero-knowledge proof communication with a judge or to provide a proof.
- **Confidentiality**: Confidentiality is achieved by encryption. To decrypt the ciphertext, an adversary needs to have Bob's private key.
- **Forward secrecy**: An adversary that obtains Alice's private key will not be able to decrypt past messages.
- **Encrypted message authentication**: The proposed scheme enables a third party to check the authenticity of the signcrypted text without having to reveal the plaintext to the third party. It provides additional confidentiality in settling disputes by allowing any trusted/untrusted judge to verify messages without revealing the sent message to the judge.

Compared to the original EC signcryption by Zheng, this new scheme requires two additional point operations on the EC, which means more computational power is needed. This is justified especially by the property of forward secrecy, which means the compromise of the current key, used in one message, should not compromise any future key. Since mobile operators may store SMS messages for an unknown amount of time, and since mobile telephone equipment has no or poor content protection and may easily be lost on the street, thus forward secrecy is considered important. The overhead saved is shown to be 43%, which is 3% more than in Zheng's scheme. This is beneficial as we saw the cost of wasting data in section 3.2, *Message size constraints*.

3.6 Key size

We want to make a solution that is secure enough for everyone. By choosing the requirements defined by NIST for the level Top Secret[NIS09] we should use 384 bits as key size for the private and public key of EC signcryption, and a 256 bits random session key for the AES-256.

This choice of key size is considered to maintain safety up to many years after 2030 in six different recommendations from academic and private organizations listed in [Blu11].

3.7 Replay detection

It does not matter if a message is encrypted and signed with the strongest scheme available, if there is no detection of replay attacks. Detection is required for the following three of four replay attack examples that are listed in [Sta05] that concerns our domain:

- **Simple replay**: The opponent simply copies a message and replays it later.
- **Repetition that can be logged**: An opponent can reply a timestamped message within the valid time window.
- **Repetition that cannot be detected**: This situation could arise because the original message could have been suppressed an thus did not arrive at its destination; only the replay message arrives.

In our case we are sending messages in a "store and forward" fashion. That means sent messages are stored in the Short Message Service Center (SMSC) before being delivered to the recipient. If the recipient's phone is turned off, the message will be stored until the validity period of the message expires. This varies between a few days to several weeks in the SMSC configurations of the different operators.

The *Simple replay* and *Repetition that can be logged* can be detected by adding a time stamp in the signed messages and keep track of the latest time stamps from every sender as a counter. It is also possible to cache the hash of the old messages, since the random session key for confidentiality would make the hash of an encrypted message unique.

When it comes to *Repetition that cannot be detected*, the "store and forward" fashion of sending SMS messages makes it impossible to detect, since for example an operator may add a delay to a message if it wants to. The only thing the user could do, would be to evaluate the importance of the content when it comes to time. For example the message: "Come out now." with an one day old time stamp could easily get discarded. But if the time stamp is 10 minutes old, the recipient should contact the sender to get the message confirmed.

The overhead needed for adding a time stamp would be 4 bytes, using Unix timestamp. The caching of old message hashes would with SHA-256 be 32 bytes, which is not much since smart phones usually have gigabytes of internal memory. These cached uniqueness values could then be deleted as the SMSC max validity period of a message expires. If a time stamp is older than this validity period, this would indicate a replay attack or a malfunctioning SMSC.

3.8 Message encoding

In our system the encoding and signature of messages will be in terms of bytes. SMS has support for data messages of 133 bytes, but the messages can not be concatenated. Since it is only possible to have concatenated text messages but not concatenated data messages we need a scheme to convert bytes into characters to obtain support for long messages.

From the old standard of e-mailing it follows the restriction that only letters readable by humans can be sent, which at that time was ASCII characters. To send attachments or parts of the e-mail message that contain characters beyond the ASCII alphabet, RFC for Multipurpose Internet Mail Extensions (MIME)[BF92] was made. In RFC an encoding scheme called BASE-64 was defined where 64 ASCII characters were used to represent a bit combination. The encoding has an overhead of approximately 4/3 which means we have to use 4 characters per 3 bytes we want to send. BASE-64 also has a less known successor called Ascii85, mainly used in Adobe's Postscript. In this encoding scheme the overhead is only 5/4 and would work as a better solution in our system because of the large overhead expenses we saw in section 3.2, *Message size constraints*. Another benefit of using only ASCII letter is 153 characters per part of a message compared to 134 characters with UTF-8 encoded characters.

3.9 Summary

By analyzing the different parts for the development process of Secure SMS we have found a list of parameters on which to base the solution.

- **Choice of layer**: The solution should be implemented on the applications layer for fast deployment.
- **Message size constraint**: There are high costs for sending extra data and this should therefore be avoided.
- **Confidentiality**: Messages should be encrypted with AES-256 and a random session key exchanged by the EC-signcryption scheme defined in [ME09]
- **Authentication**: Messages should be signed by the EC-signcryption scheme defined in [ME09]
- **Key size**: The EC-signcryption should have a key pair of 384 bits and a random symmetric session key of 256 bits, since this implies Top Secret level as discussed in [NSA10], and thereby the solution can be used by any user, private or public.

- **Replay detection**: The signed part of the message should contain a time stamp. The SHA-256 hash of old messages should be cached locally until the max SMSC storage time runs out, for checking the uniqueness of messages.

With this list of security parameters, we will in chapter 4, *Design*, plan how to realize a real life solution that serves end-to-end data protection of SMS messages.

Chapter 4

Design

This chapter covers the preliminary work of designing a total solution that ensures confidentiality, authenticity and replay detection of SMS messages. In section 4.1, *Outer structure*, we start by presenting an overview of the whole system. In section 4.2, *Use Case Diagrams*, we sketch the scenarios as seen from a user´s perspective. In section 4.3, *Inner structure*, we define precisely the cryptographic scheme that will be used and how it should be put together on generic basis in terms of Unified Modeling Language (UML). In section 4.4, *Mockup*, we sketch the Graphical User Interface (GUI) screen by screen and conduct a brief usability test with volunteers.

4.1 Outer structure

In figure 4.1 we see an overall diagram of the system. Given that Alice and Bob has exchanged public keys, the only difference seen when sending an ordinary SMS message would be for Bob that the header of the message shows up in green. This is to show that the signature of Alice is valid and having the time stamp showing the system time of when the signcryption was actually done. The implementation will be as modular as possible, so it easily may be adapted to different systems. This is shown in the figure by having a signcrypt and unsigncrypt box between the SMS client and the send and receive SMS message service of the phone.

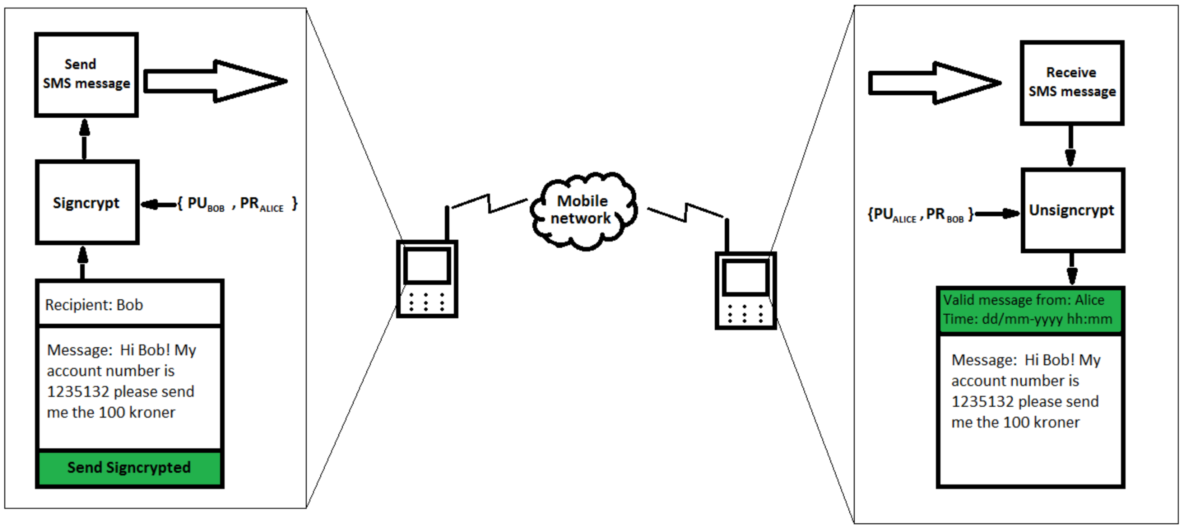

Send
SMS message
Signcrypt
{ PU_BOB , PR_ALICE }
Recipient: Bob
Message: Hi Bob! My account number is 1235132 please send me the 100 kroner
Send Signcrypted
Mobile network
Receive
SMS message
{PU_ALICE , PR_BOB }
Unsigncrypt
Valid message from: Alice
Time: dd/mm-yyyy hh:mm
Message: Hi Bob! My account number is 1235132 please send me the 100 kroner

Figure 4.1: The outer structure of the system.

4.2 Use Case Diagrams

In this section we have Use Case Diagrams to show what activities a user will need to handle. Listing all the user actions needed makes it easier to design the graphical user interface since clear constraints are given for what we need to design.

4.2.1 Install the app

Figure 4.2 shows the actions needed to get started with our app, we will call it SigncryptedSMS. The user has to be signed up for the phone specific app market, most likely Android Market. The app can then be found by searching for the app name SigncryptedSMS. When the app has been found the user needs to accept the access details

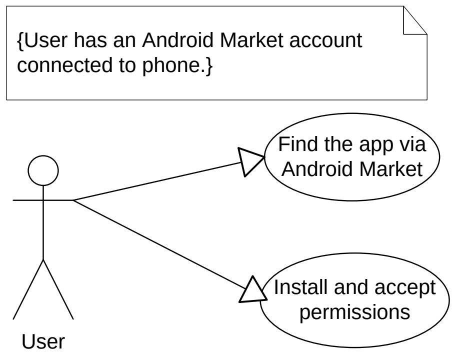

{User has an Android Market account connected to phone.}
Find the app via Android Market
Install and accept permissions
User

Figure 4.2: Use Case Digram for installation of the app.

that will be required to use the app, e.g. Send SMS message, Read SMS message etc.

The permissions required will be covered in more detail in section 5, *Implementation*. Once accepted the app will be installed and ready to use.

4.2.2 Generate key pair

The use case diagram in figure 4.3 shows the actions for generating a key pair. At first time use, the user needs to generate a key pair consisting of a public- and a private key. It is important to store the private key in a manner that will prevent anyone other

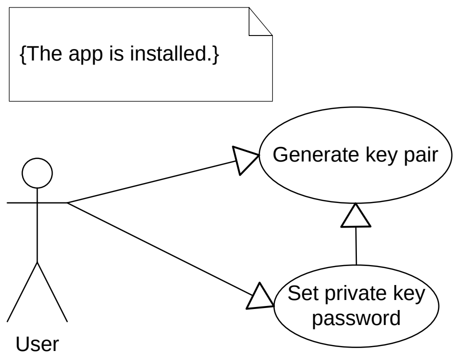


Figure 4.3: Use Case Digram for generating a key pair

than the owner from knowing it. This will be done by requesting the user to type a password for encryption before storing of the key.

4.2.3 Share Public key

Figure 4.4 shows that a user can send the public key by e-mail or SMS message to any recipient. It comes implicitly that the user needs to type an e-mail address or a mobile number.

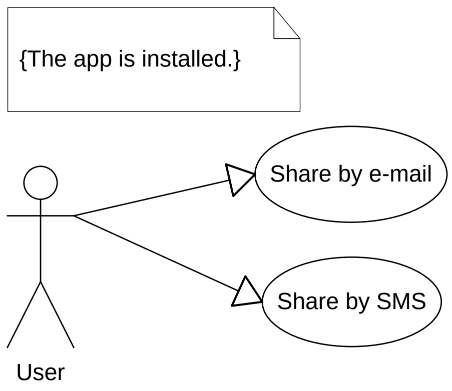


Figure 4.4: Use Case Digram for sharing public key with another user

4.2.4 Receive public key

Figure 4.5 shows that a user can insert a public key manually by typing or copying a key or the user may automatically insert a key by a public key SMS message. The user

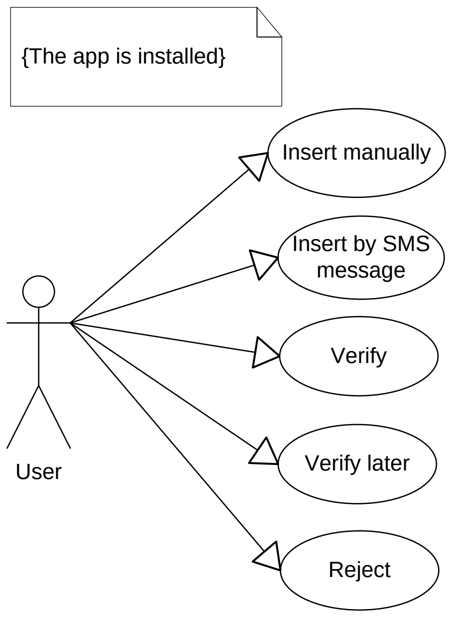


Figure 4.5: Use Case Digram for receiving a public key

can then choose to verify the key as shown in section 4.2.5, *Verify public key*, or verify it later. The user can also immediately reject the public key.

4.2.5 Verify public key

After a public key has been received the user has to verify it before using it as shown in figure 4.6. The sender and the receiver looks up the fingerprint for the public key and compare the fingerprint letters by dictating over phone or in real life. If the fingerprints are shown to be identical, the receiver of the public key can mark it as verified. If the fingerprints show dissimilarities, the public key should be rejected by the receiver and the whole process of sending it should be repeated by the sender.

4.2.6 Send message

If a user has received a public key and verified it, the process of sending a signcrypted SMS message is very easy and totally similar to sending a normal SMS message. Figure 4.7 shows that the sender only needs to choose the recipient, write the message and hit the *send* button. The entire process of securing the message is done in the background, and the user does not need to pay any more attention to the process.

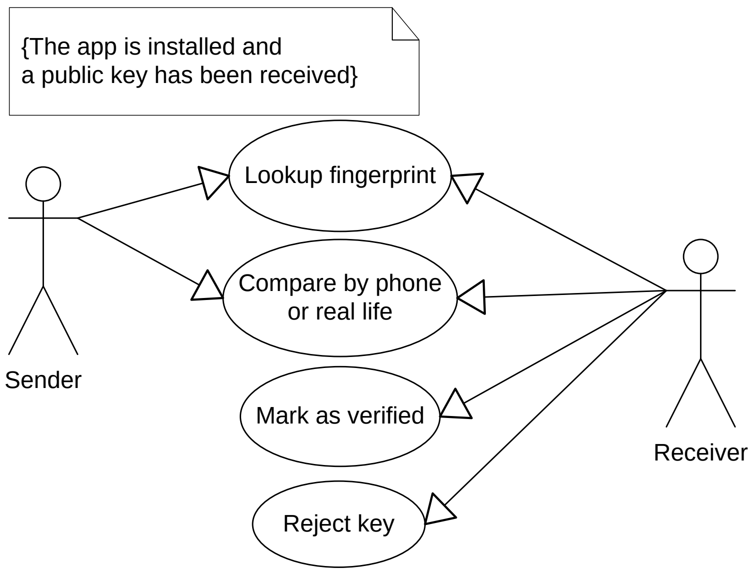

{The app is installed and
a public key has been received}
Lookup fingerprint
Compare by phone or real life
Mark as verified
Reject key
Sender
Receiver

Figure 4.6: Use Case Digram for verifying a received public key

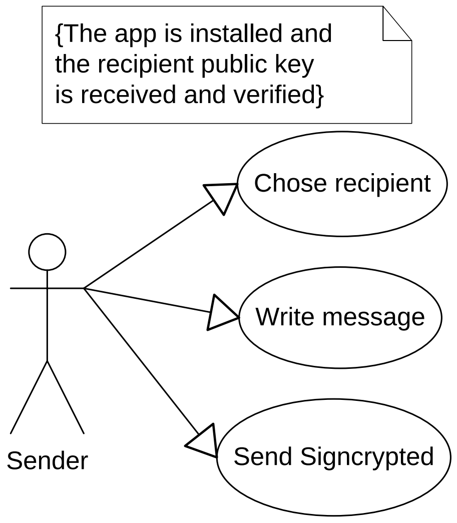


Figure 4.7: Use Case Digram for sending a signcrypted message

4.2.7 Receive message

When a signcrypted message has been received, as shown in figure 4.8, it needs to be opened with the app SigncryptedSMS. If opened with the normal SMS client the signcrypted message will only appear as an encoded message and the user will most probably understand that the SigncryptedSMS app has to be used. If the app runs in the background, the user gets a notification that the message is signcrypted and can

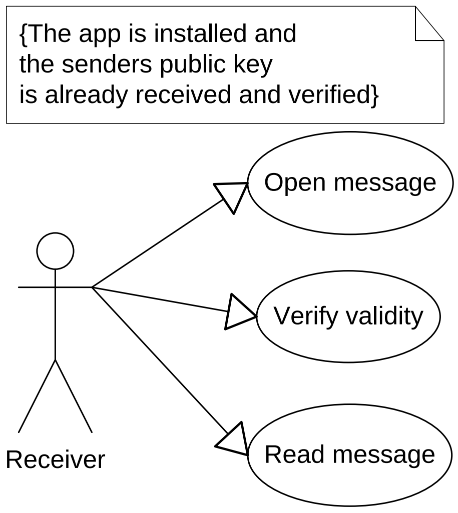


Figure 4.8: Use Case Digram for receiving a signcrypted message

open it directly by the notification icon. The user should pay attention to the validity marking of the message before giving the message trust.

4.3 Inner structure

In this section we will define the signcryption scheme mathematically. Then we will design a UML packet that shows what happens in the boxes `signcrypt` and `unsigncrypt` in figure 4.1.

4.3.1 Definition of the signcryption scheme

Definition of the EC signcryption scheme based on [ME09]:

Public parameters
C: an elliptic curve over $GF(p^h)$ with $p \geq 2^{150}$ and g$h = 1$.
q: a large prime whose size is approximately $|p^h|$.
G: a point with order q, chosen randomly from the points on C.
$hash$: a one-way hash function.
$hash_k$: a keyed one-way hash function.
E, D: the encryption and decryption algorithms of a private key cipher.

Alice's keys
v_a: Alice's private key, chosen uniformly at random from $[1, ...q-1]$.
P_a: Alice's public key, $P_a = v_a G$ a point on C.

Bob's keys
v_b: Bob's private key, chosen uniformly at random from $[1, ...q-1]$.
P_b: Bob's public key, $P_b = v_bG$ a point on C.

Signcryption of a message by Alice the sender
$v\varepsilon_R[1, ...q-1]$ a random nounce.
$k_1 = hash(vG)$
$k_2 = hash(vP_b)$
$c = E_{k_2}(m)$
$r = hash(c, k_1)$
$s = v/(r + v_a) mod\ q$
$R = rG$
Send c, R, s to Bob

Unsigncryption of c,R,s by Bob the recipient
$k_1 = hash(s(R + P_a))$
$r = hash_{k_1}(c)$
$k_2 = hash(v_bs(R + P_a))$
$m = D_{k_2}(c)$
c has a valid signature only if $rG = R$

Verification of c,R,s by a third party without disclosure of m
$k_1 = hash(s(R + P_a))$
$r = hash_{k_1}(c)$
c has a valid signature only if $rG = R$

4.3.2 Signcryption packet

A good way to make a new system is modularity. This makes it convenient to reuse blocks of code in later projects as well as to port code to different computer languages. The diagrams will be presented in block diagram level with marked direction of input and output data of the class interfaces. The internal functions will be covered in chapter 5, Implementation.

4.3.2.1 Signcrypt message

In figure 4.9 the process of signcrypting a message is shown. To signcrypt a message, the input to the signcryption packet is the recipient´s public key, the sender´s private key and the message. The keys are first used in the signcrypt class to generate the symmetric encryption key K_2. The `AES` class encrypts the message with K_2 and sends it back to the `Signcrypt` class to generate the EC point R and the integer s. As data you send in most cases are handled as individual bytes, the class `SignCryptPacket`, wraps c, R, s into a byte packet. The details of `SignCryptPacket` will be shown in

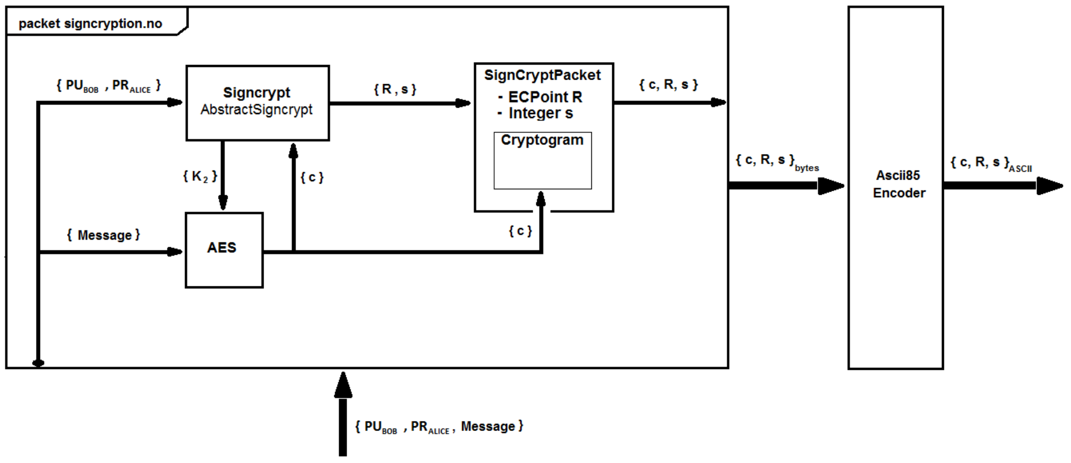


Figure 4.9: The structure of the signcryption box for signing and encrypting a message.

section 4.3.2.3. The data packet then gets encoded into ASCII letters with an Ascii85 encoder. We now have an encrypted and signed SMS message that can be sent as usual through the send SMS message service of the smart phone.

4.3.2.2 Unsigncrypt message

After we have now shown the process of signcrypting, we will in this section, in figure 4.10, show the process of unsigncrypting a message. First the ASCII encoded packet

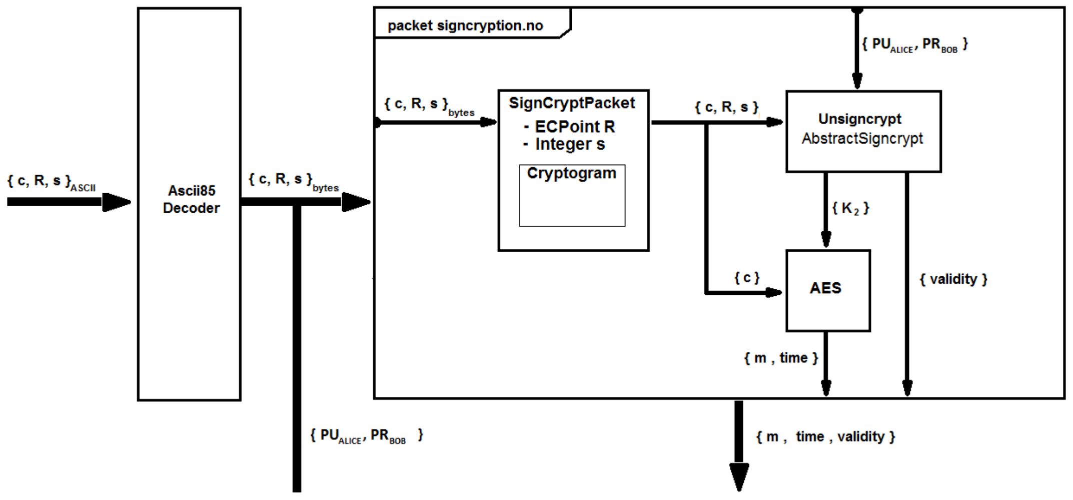


Figure 4.10: The structure of the unsigncryption box for validating and decrypting a message.

of c, R, s gets decoded back to bytes utilizing the Ascii85 decoder. Then the `SignCryptPacket` class wraps out the components *c, R and s.* With the public key of the sender, the private key of the recipient, the AES ciphertext c, the EC point R and the integer s, the `Unsigncrypt` class can calculate the AES symmetric key K_2 and whether the message has a valid signature. Notice that the validity can be verified without decrypting the ciphertext c. The `AES` class decrypts the ciphertext c with the session key K_2, and we get the two components, the cleartext message m and the *time stamp* from the sender. This tuple of $m, time, validity$ can be sent to the SMS client as we saw in figure 4.1, showing the overall diagram.

4.3.2.3 SignCryptPacket

In the figures 4.9 and 4.10 we saw the class `SignCryptPacket`. This class wraps the data we need to send as shown in figure 4.11. The one byte field *preamble* indicates

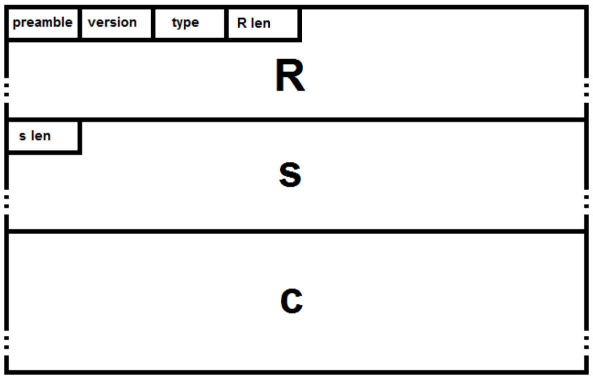


Figure 4.11: Layout of signcrypted message packet made by the `SignCryptPacket` class

whether to process an arbitrary packet as a signcryption packet. The one byte field *version* gives the possibility to change the design of the signcryption protocol. This includes all the public parameters and the packet layout. All the layout of the packet can be changed with the version number, except the preamble and the version fields that are fixed and set as two bytes for every possible new version. The one byte field *type* tells what kind of packets that are sent. The two version types that will be used in our version will be signcrypted message as in figure 4.11 or a public key as pictured in figure 4.12. This would require comparison of the key fingerprint as a key could be forged by an attacker. How to apply a fingerprint will be covered in section 4.3.2.5, *Fingerprint*. In the signcrypted message packet we need to add the two values EC point R and integer s. These values will have a variable size dependent of the random nonce v and what EC we choose, but not larger than q. The point R would have a bit size ≈ 2*(len s)* since R consists of the point elements x and y that each have the bit size $\approx$*(len s)*, but since $\pm y = x^3 + Ax + B$ we only need to send the x and a sign bit to know y. This is called point compression in EC cryptography and will make *(len R)*$\approx$*(len s)* and save a lot of overhead. The rest of the signcrypted message packet will be the symmetric ciphertext c. The size of c will be determined by the size of the whole package minus the length of all data used until this point, thus no need for a *len c* field. The public key packet are straight forward. Since a EC signcryption public key is a

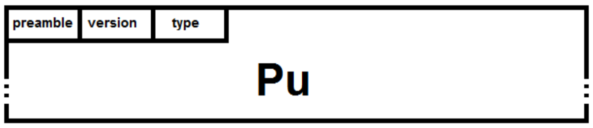


Figure 4.12: Layout of a public key packet made by the `SignCryptPacket` class

point on the EC, compression will be used and the size of it will be determined by the packet size minus the size of the fields used.

4.3.2.4 Symmetric ciphertext

The symmetric ciphertext is mainly used for confidentiality of the SMS message, but since we also want to obtain detection of replay attacks we add a *timestamp* field as shown in figure 4.13. Since AES block cipher rounds contain 16 bytes, a one byte

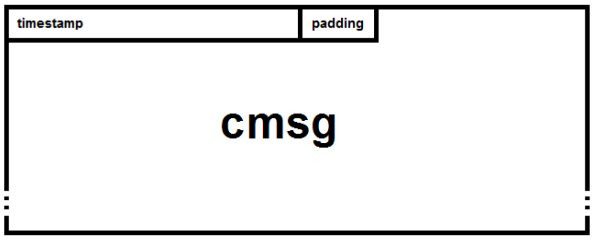


Figure 4.13: The content of the symmetric ciphertext c

padding field is needed for the recipient to know how many bytes that are padded. As an example, if a message of 17 bytes is sent, the rest of the second block, 15 bytes, will be padded with zeros. Figure 4.14 shows that when decrypting back to cleartext, we need to know the length of padded bytes since there is no need to print zeros.

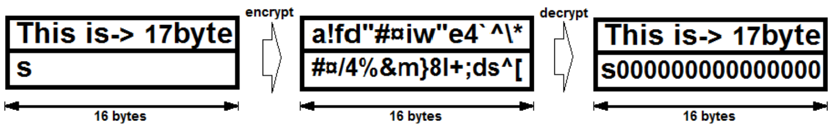


Figure 4.14: Two 16 bytes blocks of an example cmsg

4.3.2.5 Fingerprint

To make a fingerprint of a public key, we use SHA-384 as reasoned in section 3.4.2, *Authentication by hashing*. We let the phone number be the key, and the byte representation of the public key be the input. We return the fingerprint as an integer since that easily can be read and dictated.

4.4 Mockup

There is no point having a system with a lot of fancy functionality unless the users know of its existence and understand how to use it. The problem with designing a system not made specifically for an enterprise is the lack of possibilities to train the users. For this reason a high degree of usability is required. The user has to understand the functionality intuitively. Unfortunately it has been shown in a usability evaluation of PGP 5.0 that it is very difficult for a novice computer user, unknown to public key cryptography, to understand how to sign and encrypt given 90 minutes to experiment with the system[WT99]. Bearing this in mind, we will sketch all possible user scenarios that comes up in the GUI of the app and conduct a usability test with voluntary users to see if the participants can explain how every screen works.

In this project, a graphics-editing program will be used for sketching, since such a program provides a very convenient way to move the position of the objects and to change the colors. In the following subsection, we will see the most important drafts made before the implementation process.

4.4.1 Main view

Figure 4.15 is a sketch of the first screen a user will see when starting the app. First run of the app is an exception to this, since the app will generate a key pair and redirect to the screen for encrypting the private key. In the *main view*, the user will have five options; *New message, choose between conversations, share public key, manage public keys, change password* and *more options*. In the following sections we will see the screens for all of these options except *more options*. *More options* will be choices like *configuration*, *generate new key pair*, *about* and other rarely used options.

4.4.2 Generate key pair and Encrypt private key

As mentioned in section 4.4.1, *Main view*, the user will at least once need to generate a key pair and secure its private key. Since the key pair must be generated anyway, this happens in the background the first time the app is run and the user only needs to take the action of securing its private key with a password. Figure 4.16 shows that the user first needs to type the password in the input box named *Type password* and then confirm it in the input box *Confirm password*. The *Encrypt Private Key* button starts the encryption process and the user is redirected back to the *main* view with a notification of the action result status. The *cancel* button would in the first run not be

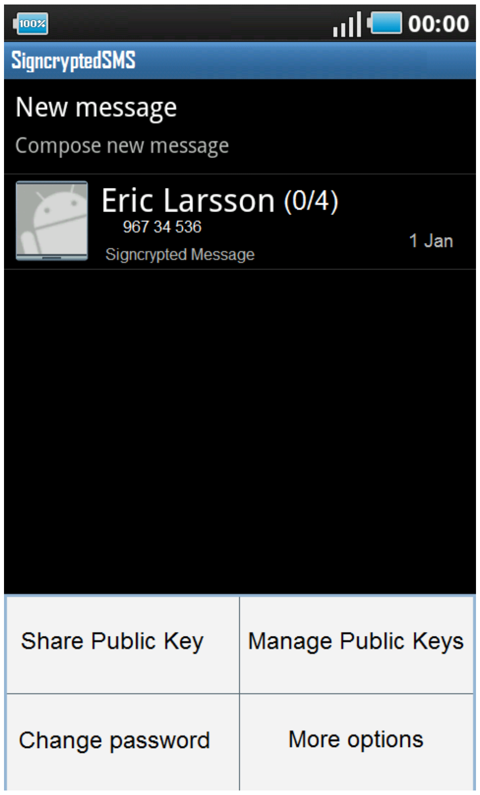


Figure 4.15: Design of the *main* view, with the options expanded

a available and in the process of changing the password it will take the user back to the *main* view with no action completed.

4.4.3 Share public key

In the *share public key* design, shown in figure 4.17, in one step the user can pick one or more users with whom to share the public key, and in the second step hit the *share with selected* button to send the public key(s). In the case where a contact has more than one contact entry of the type phone number or e-mail, the user will have to select one of them. The *cancel* button will take the user back to the *main* view with no action completed.

4.4.4 Manage public keys

Figure 4.18 shows the *manage public keys* view. In this view the user can see all public keys stored in the app. If a public key entry has an orange pending symbol, the key needs to be verified. The *start verifying* button will open a dialog that displays the public key fingerprint and encourage the user to compare the fingerprint with that of

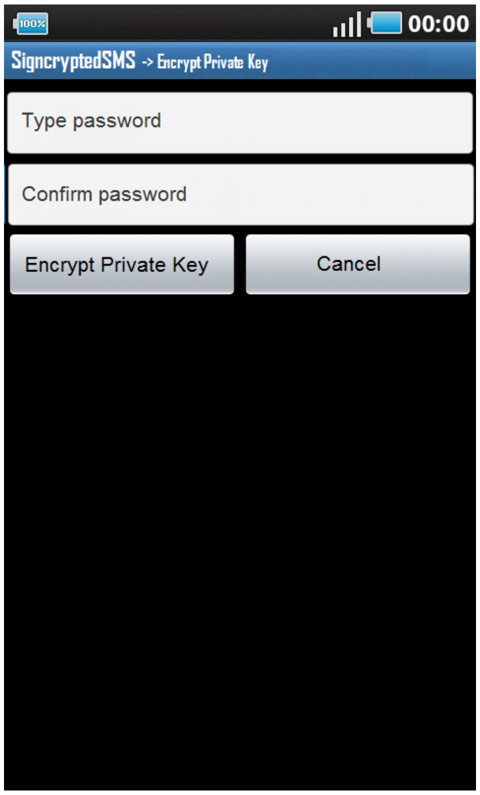


Figure 4.16: Design of the *encrypt private key* view

the owner, live, over a phone call or in any other authenticated way. The user can then choose to accept or reject the public key, or just to cancel the process. If a public key entry has a green *verified* icon, it means the public key has been verified and signed with the users private key at some point. The *more info* button in the same row will show when the verification was signed and the respective public key fingerprint. A public key entry can be deleted with the standard Android *"long hold item"* option. If a public key has been received by mail or in any other text based manner, it can be imported via the clipboard with the *import from text* option. If the format is not recognized, the user will be notified. With the *import from QR code* option, the systems default QR code reader will be used to return a public key.

4.4.5 Conversation

If the user selects one of the conversations listed in the *main* view as we saw in section 4.15, *Main view*, a *conversation* view like the one in figure 4.19 will be shown. This view is very similar to the default SMS client in Android. The talking bubbles from the left are the messages received and the ones from the right are the ones sent from the user. In the example conversation with Eric Larsson, the fist bubble is a normal incoming SMS

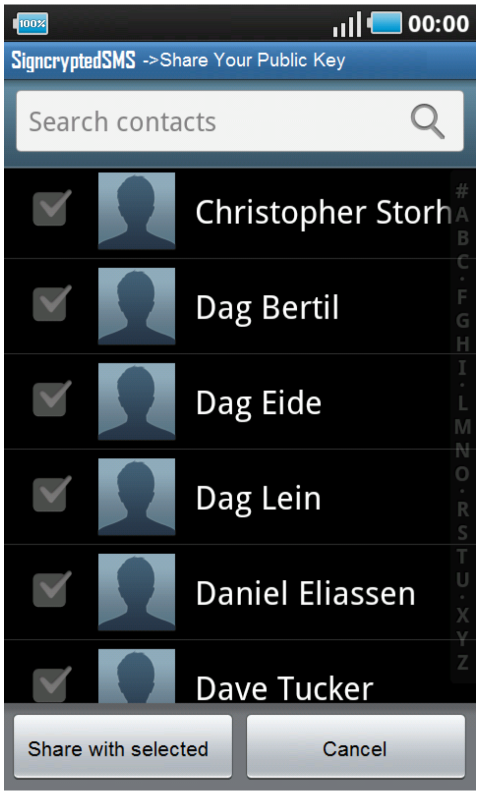


Figure 4.17: Design of the *Share public key* view

message with no signcryption. Since the message is not signcrypted, the upper area of the message notifies in red that the message is not secured, and encourages the user to exchange keys for a secured conversation. The next message in the example is sent out from the user as a normal SMS message. When the user does not have any public keys from the recipient, the *send signcrypted* button will be marked *send unsecured*. The outgoing message will be marked in red as not signcrypted. In the third bubble the sender and the recipient has exchanged public keys, and the upper part is marked as signcrypted by the one sent it and at what time it was signcrypted. The time marked in the lower corner is the time the message was received. The last message sent out in the conversation is in the same way marked in green as signcrypted with the time stamp of the signcryption. As the user now has the public key of the recipient, the only thing needed for sending a secured SMS message is to write in the input box *type to compose* and hit the *send signcrypted* button. There are almost no difference from the default SMS client except for the upper red or green info bar in the messages, so the system should be intuitive for the novice Android user. Another difference is the *more info* button that gives the same key status as given in the key management menu for a public key entry, and makes it more convenient to check the public key status.

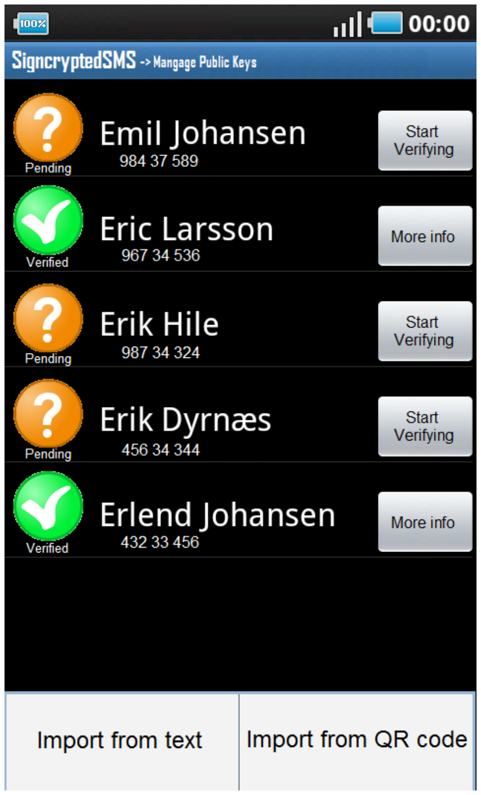


Figure 4.18: Design of the *manage public keys* view, with options expanded

4.4.6 Usability testing

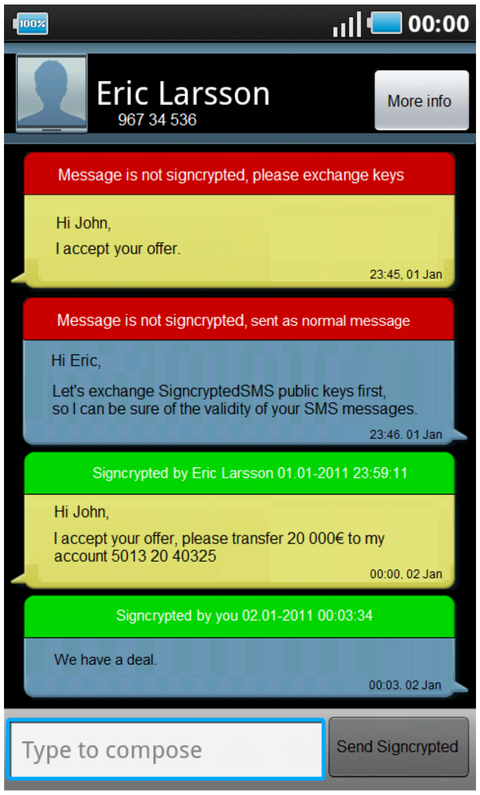
As we now have the drafts made in section 4.4.1 - 4.4.5, we want to test the usability on volunteers.

4.4.6.1 Test set up

1. Use volunteers without knowledge of the system.
2. Show every draft in the same order as they were presented in section 4.4.1 - 4.4.5.
3. Ask the volunteers to explain what they expect as possible to do in every view.
4. At the end of the test, explain the parts of the app they did not understand.
5. Ask for proposals for improvement.

4.4.6.2 The volunteers

The testing is based on the input from seven participants; Six students of Communication technology in the fifth year, of whom four specialize in Tele Economics and two



Message is not signcrypted, please exchange keys

Hi John,
I accept your offer.
23:45, 01 Jan

Message is not signcrypted, sent as normal message

Hi Eric,
Let's exchange SigncryptedSMS public keys first,
so I can be sure of the validity of your SMS messages.
23:46, 01 Jan

Signcrypted by Eric Larsson 01.01-2011 23:59:11

Hi John,
I accept your offer, please transfer 20 000€ to my account 5013 20 40325
00:00, 02 Jan

Signcrypted by you 02.01-2011 00:03:34

We have a deal.
00:03, 02 Jan

Figure 4.19: Design of an example conversation with an recipient

students of Information Security. The last participant is a student of Electronics in the third year.

4.4.6.3 Test results

Only the most interesting results will be presented and taken into account for implementation.

- The two participants specializing in information security understood everything with no doubts.
- The third participant studying Tele Economics had hard time understanding any of the options considering keys. A proposal from the participant was to have an *information* button to explain the concept of public key cryptography, since everything was understandable after a brief explanation of the concept.

Based on the experience from the discussions with the third participant, the one not knowing the concept of public key cryptography, the next participants were first asked if they knew the concept of public key cryptography. If not, they were briefly explained this concept, especially the importance of keeping the private key secret.

- Four of the participants thought the *Change password* option in the *main* view was to change a system password, but understood it was for encrypting the private key when the associated view was shown.
- Everyone understood the coloring red and green, and if a message was signcrypted or not.
- Three of the participants confused incoming with outgoing messages in the *conversation* view.
- Five of the participants thought the Signcrypted time stamp indicated the time when the message was received.
- Two participants discovered a design error, that the old password was not requested when changing the private key encryption password.
- All the participants understood how to share a public key and how to import one.
- In the *Manage Public Keys* view, the participants not studying Information security could not understand what the *Start Verifying* button would do, but they would have hit it if they had the real app.
- In the *conversation* view, the participant studying Electronics, proposed a *Send normal SMS* button.

4.4.6.4 Design to change

Based on the results shown in section 4.4.6.3, *Test results*, the following changes in the final implementation are proposed:

In the *main* view, the *Change password* option should be named *Change private key password.* A *help* option should be added to give a brief explanation of the public key cryptography concept with some explanatory figures.

In the *Encrypt Private Key* view, an old password input box should be added on the top.

In the *conversation* view, each message should be marked with the sender´s name. An additional button for sending unsigncrypted messages should be added.

We are now ready for the chapter 5, *Implementation*, where we can use the guidelines and the feedback from the usability test in this chapter, to implement a final system, that works in real life.

Chapter 5

Implementation

In this chapter we will go through the most important details of the implementation of end-to-end data protection of SMS messages. Section 5.1, *Equipment*, shows the equipment utilized for coding and executing the implemented programs. Section 5.2, *Development environment*, shows the installation needed for development. The actual implementation will be done in three steps. As the first step in section 5.3, *Signcryption packet*, we will make a Java package that handles the cryptographic scheme. Then in section 5.4, *Android app*, we will first implement a small Android app that tests the capability of smart phones, with their limited resources, to run signcryption. As the final step we will, based on the planning and experience from the previous work, implement and realize a secure SMS client that utilizes the signcryption packet we prepared in the first step.

5.1 Equipment

For coding, compiling and emulating;

- Dell Latitude E4310 laptop with 2.4GHz Intel Core M520 CPU, Mobile Intel QS57 Express Chipset, 4GB RAM, Windows 7, 64-bit Enterprise OS installed.

For run and test in real environment;

- Samsung Galaxy S smart phone, with 1GHz ARM Cortex-A8 CPU, Hummingbird chipset, Android OS, v2.2(Froyo) installed.

- Samsung Galaxy 5 smart phone, with MSM7227-1 600MHz CPU, Android OS, v2.1 update 1(Eclair) installed.

Both smart phones are equipped with Norwegian mobile subscription numbers with access to send and receive SMS messages.

5.2 Development environment

Four initial steps are required before starting the development of the Android application:

1. Install Eclipse Helios build:20100917-0705[ecl11] and the latest Java Developer Kit (JDK)
2. Install the Android System Developer Kit (SDK) for Windows[Inc11a]
3. Install the custom Eclipse plugin called Android Development Tools (ADT) offered by Android[Inc11b]
4. Use the Android SDK and AVD Manager to Install SDK Platform Android 2.1 update 1, Application Programing Interface (API) 7.

Since third party apps for Android are Java based and run on top of Java core libraries, the development environment is very similar to the environment for developing Java applications. The ADT plugin mentioned in step three generates a new Android project, building the base structure needed.

5.3 Signcryption packet

In section 4.1, *Outer structure*, we plan the development in a modular way, to easily reuse the code in other projects or port the code into other programming languages. Figure 4.1 showed a module that handles the securing of SMS and which has been further designed in UML, in section 4.3.2.1 and 4.3.2.2, *Signcrypt* and *Unsigncrypt message*. Since this is the essence of the project, we will develop this as an ordinary Java packet and then make it into an importable library that can be utilized in Android projects.

The Java code written for the signcryption packet is added in Appendix A and named `no.altconsult.signcryption` according to the unique domain name convention used in Java packages. The domain *altconsult.no* is owned by the writer. This provides high probability for uniqueness and avoids packet naming conflicts with other packet providers. The reader is encouraged to look at the code for all details of the implementation, but is not required to do so for further reading.

Figure 5.1 shows the associations of the packet classes. The following sections will briefly explain each of the classes in the diagram.

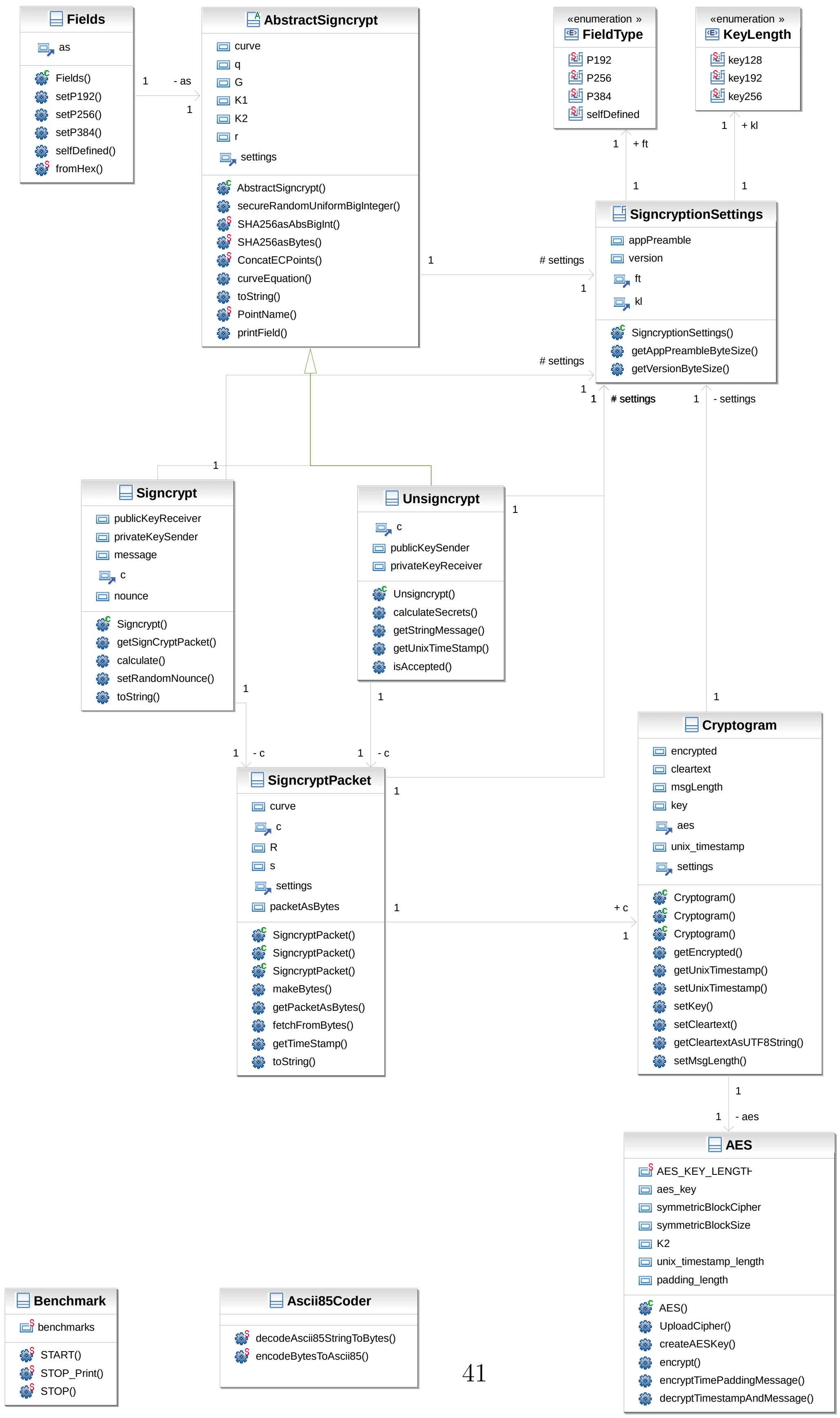


Figure 5.1: Association diagram for the signcryption packet

5.3.1 SigncryptionSettings class

Before explaining any of the other classes we will point out the class with the most associations to the other classes. The `SigncryptionSettings` class is a class that will give configurability to the package. The idea is to instantiate a class with configurations, and pass it to all the other classes. The configurations to set are: the *app preamble* identifying the app we are using, the *version* of the app, what *field type* is used and the *key length* of the AES encryption.

The code for this section is listed in Appendix A.4, *SigncryptionSettings.java*.

5.3.2 Fields class and FieldType enum

Defining efficient and working EC fields requires computation and testing. Since NIST already has defined and recommended curves for different security levels in [NIS09, page 88-89], we have added three curves called P192, P256 and P384. The P stands for prime number and the number behind, tells the bit size of the prime. By instantiating `Fields` with an `FieldType` enum, the global cryptographic parameters will be set. We decided previously in section 3.6, *Key size*, that the key size should be 384 bits in order to achieve the security level *Top Secret*. This implies that the curve P384 will be used since the key is used modulo to a 384 bits prime. We also add the EC fields P192 and P256 to make it possible to compare the efficiency between the three curves.

The code for this section is listed in Appendix A.7, *Fields.java* and Appendix A.6, *FieldType.java*.

5.3.3 AbstractSigncrypt class

The signcryption scheme is divided into two sub schemes, signcrypt and unsigncrypt. They share common properties like the global cryptographic variables, settings and some functions. For this reason we choose to have an `AbstractSigncrypt` class with common properties that can be inherited.

The Java main library accommodates Elliptic Curves, but these lack arithmetic functionality. In our case we need EC point summation and scalar multiplication. For this reason we import a third party open source cryptographic Java library called `org.bouncycastle`[bou11] with open licensing.

The code for this section is listed in Appendix A.1, *AbstractSigncrypt.java*.

5.3.4 Signcrypt class

The purpose of the `Signcrypt` class is to use the recipient's public key, the sender's private key and the message to calculate the parameters *c, R and s* and give access to functionality that wraps it into bytes. The calculations are done using the `calculate` method. The random uniform nonce called v in the signcryption scheme is provided by the `java.security. SecureRandom` class that uses sources of real entropy from the local machine provided by the OS. This complies with FIPS 140-2[NIS01] and is regarded as secure. The hash function is of the type SHA-256 served by the `java.security.MessageDigest` class and gives us the 256 bits keys *K1* and *K2*. It also works as a keyed hash function to generate r with the AES-256 cryptogram and *K1* as input. The calculation of

$$s = v/(r + v_a) mod\ q$$

defined in section 4.3.1, *Definition of the signcryption scheme*, was a bit hard to solve by programming, where the first attempt was to divide $v/(r + v_a)$ for which the computer could not provide a correct solution. Rewriting the expression mathematically

$$s = v((r + v_a)^{-1} mod\ q)\ mod\ q$$

and to Java code with `java.BigInteger`

```
s = v.multiply(r.add(v_a).modInverse(q)).mod(q);
```

gave a fast and correct solution of the problem. At the end of the calculation process, the R is re-instantiated with point compression served by `org.bouncycastle` that saves half the overhead as shown mathematically in section 4.3.2.3, *SignCryptPacket*. When all these calculations are done we give access to a byte packet with a function called getSignCryptPacket explained in section 5.3.5, *SigncryptPacket class*.

The code for this section is listed in Appendix A.2, *Signcrypt.java*.

5.3.5 SigncryptPacket class

The universal way to transmit through a communication channel is in terms of bits and bytes. For this reason we have a class called `SigncryptPacket` that handles this. The class can be instantiated with the parameters *c, R, s*, or with bytes. The byte representation is ordered exactly as we designed in figure 4.11. The stacking of the bytes is mostly done with `System.arraycopy` and a position counter that keeps track of where in the copy process we are. Almost the same procedure is done to reverse the process for fetching the bytes back into the original field values. The byte translation of the different values are mostly served by the Java types except the byte representation of a EC point, that is served by the `org.bouncycastle` library with:

```
        //Encode EC point into bytes
        RBytes = R.getEncoded();
        //Decode EC point back from bytes
        R = curve.decodePoint(RBytes);
```

The only methods that are made public are `getPacketAsBytes` and `getTimeStamp`. All the processing is activated in private methods, dependent of whether the instantiation is in terms of bytes or the parameters *c, R and s.*

When it comes to the the symmetric cryptogram c more wrappings are done, and section 5.3.6, *Cryptogram class*, explains this.

The code for this section is listed in Appendix A.9, *SigncryptPacket.java.*

5.3.6 Cryptogram class

With a UTF-8 formatted cleartext string or a ciphertext of bytes and a symmetric key, this class outputs the encryption, decryption and the timestamp for when it was done. The class primarily structures the necessary fields, the calculations are done by the class called `AES` and is explained in section 5.3.7, *AES class.*

The code for this section is listed in Appendix A.10, *Cryptogram.java.*

5.3.7 AES class

The `Signcrypt` and `Unsigncrypt` classes calculates a shared session key called *K2*. *K2* ends up in the `AES` class as a byte array. Depending on the key length set in `SigncryptionSettings`, the first 128, 192 or 256 bits are used of *K2*. *K2* will always be 256 bits, since it comes from an SHA-256 function.

Before encryption, Unix timestamp and the padding size is calculated and converted to bytes, then added to the leading edge of the cleartext as designed in section 4.3.2.4, *Symmetric ciphertext.* This is delivered through the public method `encryptTimePaddingMessage`.

The reverse process, decrypting, is done in reverse order and a pointer to the Cryptogram is used to set the values; time stamp, message length and the cleartext message.

The symmetric cipher is configured to Cipher Block Chaining (CBC) Mode. This requires the last cleartext block to be padded. For the first encryption an Initialization Vector (IV) of one block size is needed, which in AES is 16 bytes. To make an IV, we use the 16 first bytes of the shared session secret key *K1*.

The code for this section is listed in Appendix A.11, *AES.java.*

5.3.8 Ascii85Coder class

As all the variables *c, R and s* are calculated and wrapped into a byte packet, they are now ready to be sent. To give the choice of sending the byte packet in terms of ASCII letters instead, an `Ascii85Coder` class has been made. The class is optional to use and contains two static methods. One method has byte array as input and ASCII formatted string as output, and the other method has ASCII formatted string as input and bytes as output.

These methods are based on `input` and `output stream` classes of Ascii85 from a third party library called `com.idataconnect.lib`. The library is open source, and has been modified in this project to return *-1000* instead of *-1* as last byte read notification. This solves a problem since AES by chance could allocate the value *-1* to some of the bytes and that would make the algorithm stop. Since a byte can not be larger than 255, the last byte read notification was changed to a value outside the possible range of byte values.

The Ascii85 encoding has an overhead of 25% as mentioned in section 3.8, *Message encoding*, and becomes useful for sending long SMS messages.

The code for this section is listed in Appendix A.12, *Ascii85Coder.java.*

5.3.9 Unsigncrypt class

This class is used by the recipient with `SigncryptPacket` as input, holding *c, R and s* to calculate the session keys *K1* and *K2* and verify the signature. Then with the help of the classes `Cryptogram` and `AES`, the cleartext message, the signcryption timestamp and the validity are given as output.

The calculation is done straight forward with the same functions as explained in section 5.3.4, *Signcrypt class*, thus by EC point summation, EC point scalar multiplications and hash function.

The code for this section is listed in Appendix A.3, *Unsigncrypt.java.*

5.3.10 Benchmark class

For testing the run time for different configurations and hardware a `Benchmark` class has been established. This class consists of a start and stop pair where every pair is given a String identifier. The idea is to put pairs around different operations, to see if there is code of configurations that needs to be tweaked. The implementation is done by adding the current system time to the String identifier as a key with the `START` method. When the `STOP` method with the same key is run, we use the stop time minus the start time as the run time. The following listing shows an example of how to benchmark:

```
Benchmark.START("Test method")
testMethod();
String result = Benchmark.STOP("Test method")
```

The code for this section is listed in Appendix A.13, *Benchmark.java*.

5.3.11 Export as library

We have covered the process of signcryption using Java code. Now we need to make a library that easily can be imported into other projects. In Eclipse we go to: *Export* -▷ *JAR file*, then the concerned packets `no.altconsult.signcryption` and `com.idataco-nnect.lib.ascii85codec` are selected. We mark the options boxes; *Export generated class files and resources* and *Compress the contents of the JAR file*. Hitting the **finish** button makes a file called *no.altconsult.signcryption.jar* which is an external library.

This is a library that can be imported into any project running on Java Runtime Environment (JRE) 1.6 or later. In the next section, section 5.4, *Android app*, the signcryption packet is imported into an Android project and will utilize the cryptographic features we have prepared so far.

5.4 Android app

Using the previously installed ADT in Eclipse, we generate an app structure by selecting *new android project*. The common development pattern in Android is Model-View-Controller (MVC) shown in figure 5.2, and the application is organized in accordance with this. The controllers are defined in classes called `Activities`. The views are defined in Extensible Markup Language (XML) formatted files, called `layouts`. The models are defined as clean data classes with getters and setters connected to data via content providers.

5.4.1 BenchmarkSigncryption app

In this section we implement an app that gives the time spent on running signcryption on Android phones. The project imports the `no.altconsult.signcryption` packet implemented in section 5.3, *Signcryption packet*, and its dependency `org.bouncycastle`.

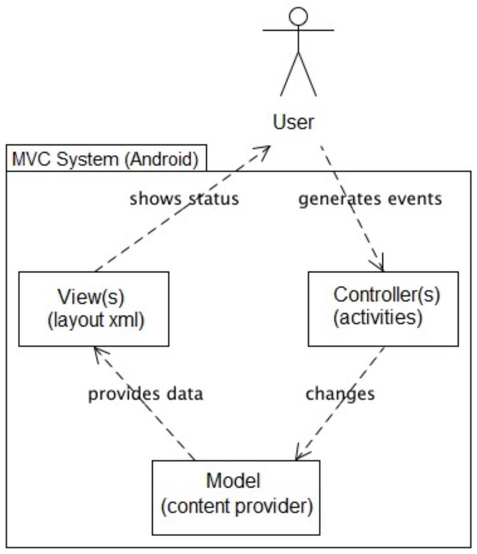


Figure 5.2: Structure of the MVC pattern with the Android terms covered in parenthesis

This app only needs one `Activity` to show the results. The whole signcryption scheme is added in the method `test` in the same order as shown in the following list.

1. Make a `SigncryptionSettings` object, where an elliptic curve type of `P192`, `P256` or `P384` is selected, and a symmetric key size of `128`, `192` and `256` bits.
2. Generate a random key pair to the sender and the recipient.
3. Instantiate a `Signcrypt` object with a message and the parameters already set.
4. Encode the byte packet to Ascii85.
5. Decode the Ascii85 message back to bytes.
6. Use the same `settings` object as in the first step and instantiate a `Unsigncrypt` object.
7. Write out the received message and timestamp to check if these correspond to the signcrypted cleartext message and timestamp.

As explained in section 5.3.10, *Benchmark class*, we surround the code we want to benchmark with identified start and stop pairs. The results will be printed to screen and will be shown in section 6.1.2, *Benchmark Signcryption*.

5.4.2 SigncryptedSMS app

This part of the assignment is the realization of everything planned and designed so far. We implement the GUI designed in section 4.4, *Mockup*, import the cryptographic functionality implemented in section 5.3, *Signcryption packet*, and add the services of sending and receiving SMS messages. We thus provide Android users with end-to-end protected SMS messaging with only small changes to their common messaging environment.

The first thought of implementing the signcryption packet, was to find an open source SMS client for Android and modify it to fit the planned design. With lack of such projects the client had to be programmed from scratch. Before explaining the most essential parts of the app implementation we will give a very brief explanation of `Activities`, `Intents` and `ContentProvider` in Android.

5.4.2.1 Short about Activities, Intents and ContentProvider

As mentioned, a common way to organize an Android app is by using one `Activity` class for every view shown to the user. Every `Activity` has its own `layout` file that specifies the appearance. This would mean that we need to make an `Activity` for the figures 4.15 - 4.19. Every `Activity` will be explained in the following sections.

The communication between the activities is done with messages called `Intents`. Android uses this to communicate with other services and apps running on the OS, and in many situations simplify the programming, because we can utilize already implemented functionality. This will be apparent in the explanation of the activities. In figure 5.3, we show how the activities in `SigncryptedSMS` are connected together. The `Activity` outside the `SigncryptedSMS` package is to show how we can get service from other apps integrated.

Phone specific data such as for example contacts and SMS messages are available in Android through a service called ContentProvider. This serves all data in a SQL manner, and makes data access convenient. What data set to access is done by specifying an Uniform Resource Identifier (URI), like for example to access the SMS inbox:

```
uri = content://sms/inbox
```

5.4.2.2 SigncryptionManager and Signcryption class

To simplify the connection between the *no.altconsult.signcryption* packet and the android project. We have to add a class to the `signcryption` packet that extends the `AbstractSigncryption` class to get access to the EC field and some of the functions needed from the packet. To connect the app, we make a class in the Android project called `Signcryption` that extends `SigncryptionManager` and adds the public methods needed to serve signcryption and unsigncryption. This means that we for example make a public method called `signcryptMessage` that takes the message, the private key of the sender and the public key of the recipient, and then returns an Ascii85 encoded

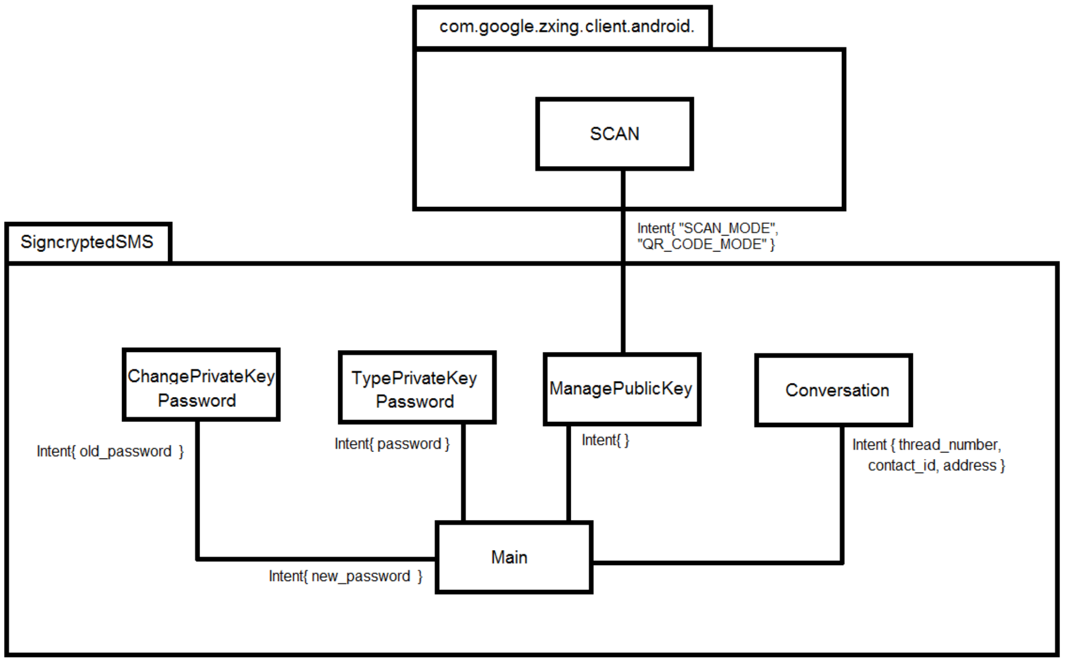


Figure 5.3: Structure of the Activities in SigncryptedSMS app

string with *c, R and s.* This way we organize the Android specific aspects, concerned with the signcryption scheme, into their own class in the `SigncryptedSMS app`.

The code for this section is listed in Appendix A.5, *SigncryptionManager.java* and Appendix C.7, *Signcryption.java*

5.4.2.3 MainActivity

The `MainActivity` is the first class started when the app starts. All other activities are called from this class by `Intents`. The implementation of the class is listed in Appendix C.1, *MainActivity.java*.

In creation of this `Activity` we initialize a static field called `signcryption` that is an instance of the `Signcryption` class. With the signcryption field, in the first launch of the app, a private key is generated and stored on the private folder of the app.

All SMS conversations on the phone are requested through the content manager with the URI:

```
uri = content://sms/conversations/
```

The result is added into a model object that holds the data fields defined in the class `ModelSMSThreadRow` listed in Appendix C.8, *ModelSMSThreadRow.java*. Every conversation row is then added to an `Arraylist` and presented by an `ArrayAdapter`. The `ArrayAdapter` defines the connection between the model and the view defined in the layout file *row_message_thread.xml* listed in Appendix C.13.

If the user clicks on a conversation row or a new message, the `ConversationListActivity` is called by `Intent` with the selected conversation ID. If a new message is selected the thread identifier is set to zero.

The options menu item:*share public key* activates an `Intent` that sends the user to the pick contact service, served by Android. The service returns an URI that gives direct access to contact info through the content provider. This is a nice example of how Android provides extra functionality with a few lines of code from other services.

The options menu item: *Manage Pu Keys*, sends the user to the `ManagePublicKeysList` activity.

The options menu item: *Pk password*, sends the user to the `ChangePrivateKeyPassword` activity.

5.4.2.4 TypePrivateKeyPasswordActivity

The `TypePrivateKeyPassword` activity takes the private key encryption password as input from an `EditText` input box and evaluates it. When the correct password is entered the `Activity` sends the user back to the `MainActivity`.

The code for this activity is listed in Appendix C.2, *TypePrivateKeyPasswordActivity.java.*

5.4.2.5 ChangePrivateKeyPasswordActivity

The `ChangePrivateKeyPassword` activity takes the old password, new password and confirm password from `EditText` boxes, and evaluates the inputs with regard to whether it is blank, the old password is incorrect or whether the new password and old password differ. If the input testing is evaluated as positive the user gets redirected with the new encryption password to the `MainActivity`, which also handles the request.

The code for this activity is listed in Appendix C.3, *ChangePrivateKeyPasswordActivity.java.*

5.4.2.6 ManagePublicKeysList

The `ManagePublicKeysList` activity, has the purpose to list all imported/ received public keys and organize their validity.

The public keys are stored in the note field of Android contacts. This implies that for saving a public key the contact has to be listed. When a key is received the first time, it gets tagged as *<pending>*.

To list all the public keys from the contacts a query for all contacts with a note pattern like *'<verified>,<pending><PuKey>%'* is made. The data is added to `ModelPublicKeyRow`, listed in Appendix C.10 and presented through an `ArrayAdapter` configured to use the layout defined in *row_public_key.xml*, listed in Appendix C.14

The **verify** button invokes a dialog that presents the key fingerprint and the choices verify and cancel. If the user verifies, the key gets tagged <verified> and updated in the current contacts note.

There is no security added to the tagging and the tag can be changed manually in the contacts manager. If a key is tagged as verified, the button will appear with the label **more info** and open the same dialog, but present the key as verified and show the key fingerprint.

The option button **Import from text** has no functionality implemented.

The option button **Import from QR** triggers an `Intent` to a package called `com.google.zxing.client.android` that serves a QR code reader through the camera on the smart phone. The package is only available if the user already has installed the app called Barcode Scanner from Android Market. If the scanner is available, a scanned public key results in a String with the format:

```
String qr = "<{phonenumber}><~@~><PuKey><~{Ascii85 encoded ↩
    → public key}~>";
```

This string will be evaluated and separated into a phone number and a public key that gives us the possibility to add the public key to the contact having this particular phone number.

Another way to obtain a new public key is if someone sends the key by the share public key option described in section 5.4.2.3, *MainActivity*. The SMS message containing the public key will be in the form:

```
String PuKey = "<PuKey><~{Ascii85 encoded public key}~>";
```

The phonenumber will be attached to message header so we do not need to divide the String this time. The public key gets added to notes in the same way as with QR code.

The code for this `Activity` is listed in Appendix C.4, *ManagePublicKeysList.java*.

5.4.2.7 ConversationListActivity

An SMS conversation selected in the `MainActivity` is listed by the *ConversationListActivity* activity. This `Activity` shows all messages sent from and received by a phone number. The `Activity` also serves the possibility to write and send a new message to a contact. If a verified public key for the contact is available, the message is sent signcrypted.

Messages from a specific conversation is accessed with the content provider with the URI:

```
uri = content://sms/conversations/{id}
```

where the id is obtained from the `MainActivity` as a message from the invoking `Intent`. The information for every message is added into a ModelSMSmessage object defined in *ModelSMSmessage.java*, and listed in Appendix C.9. Every message content is parsed with the methods `isSigncryptedSMSMessage` and `isPublicKeySMSMessage` defined in the `Signcryption` class. With the signcryption state set in each model object, the `ArrayAdapter` can attach every message to a row item with the formatting defined in its layout file.

Messages are sent via an Android class called `SMSManager`. With this class we first use a method called `divideMessage`, establishing an array of parts of messages, then we use the same array as input to a method called `sendMultipartTextMessage`. This way we have support for long SMS messages. The messages sent have to be added manually via the content provider with the URI:

```
uri = content://sms/sent/
```

Since signcrypted messages only can be read by the recipient, we choose to store the messages as cleartext in the sent folder. To show the last sent message, we have to render the array adapter through a method that queries the conversation again and gets the last state.

The code for this `Activity` is listed in Appendix C.5, *ConversationListActivity.java*.

5.4.2.8 SmsReceiver class

We mentioned that in Android, messages can be sent between processes as `Intents`. `Intents` can also be broadcast to all processes that have been set to listen for an `Intent` type. This is what the SMS service on the phone does every time a new SMS message arrives.

We have made a class that extends `BroadcastReceiver`. This class is configured in the app manifest to filter out all `Intents` of the type `android.provider.Telephony.SMS_R-ECEIVED` ie. incoming SMS messages. When a new message is received, the content is checked if it is a public key. If so, the message is sent further to the `ManagePublicKeysList` where the user may choose to verify the new public key received.

The code for the SmsReceiver is listed in Appendix C.6, *SmsReceiver.java*.

5.4.2.9 App Manifest

Android runs a very strict policy regarding the rights of a third party app. Every app has an XML formatted Manifest file, which will request the appropriate rights from the OS. In this app we have to send a request for:

- Send, Read and Receive SMS messages
- Read and Write in Contacts

In the Manifest file we also define the name and the logo of the app, all activities, and the `MainActivity` that starts when the app is initiated.

The Manifest file for the app is listed in Appendix C.15, *AndroidManifest.xml*.

Chapter 6

Results

In this project the results are presented in two parts. The first part is a Java packet called **no.altconsult.signcryption** that gives confidentiality, authenticity and reply detection of text by public key cryptography. This part will be found below in section 6.1, *Signcryption packet results.*

The second part of the results is **SigncryptedSMS**, an SMS client for Android phones that implements the security services from the **no.altconsult.signcryption** packet. The client is made similar to the original Android SMS client, and provides SMS data protection almost seamlessly, seen from the user perspective. The results of this part are found in section 6.2, *SigncryptedSMS app results.*

6.1 Signcryption packet results

The result of the signcryption module that has been planned and implemented for the project, is a 95kB jar file called *no.altconsult.signcryption.jar*. It supports any Java project using the core library Java 1.6 or later and has dependency on Bouncy Castles open source library[bou11].

6.1.1 Usage

Once the packet is imported into a class only a few lines are required to obtain signcryption and unsigncryption of messages. The usage is best shown with examples, as shown in the following listings.

First we configure the packet:

```
settings = new SigncryptionSettings(
    (byte)0xAA,//preamble to identify signcrypted messages
    (byte)1,//set version number to specific protocol
    FieldType.P384,//set predefined EC field
    KeyLength.key256//set key length of AES encryption
    );
```

Given that all keys are available, we signcrypt a message on the sender's side and end up with a byte array that can be sent to the recipient:

```
sc = new Signcrypt(
    getPrivateKeySender(),//Sender's Private Key
    getPublicKeyReceiver(),//Recipient's Public Key
    message, //String containing the message
    settings//The configuration we set
    );

//Signcrypts the message and returns it as bytes
byte[] bytes = sc.getSignCryptPacket().getPacketAsBytes();
```

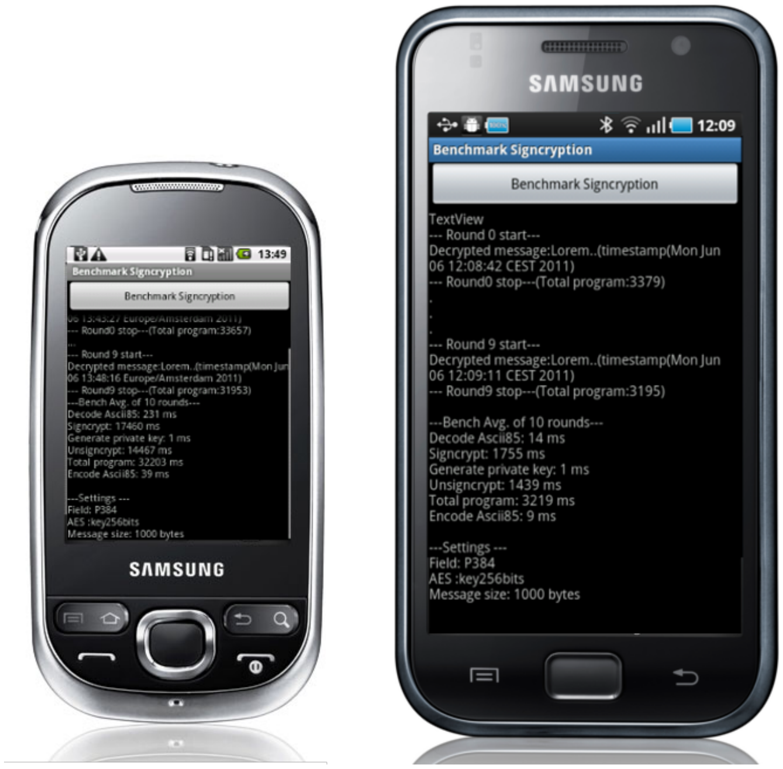
Given that all keys are available, we unsigncrypt on the recipient's side. Via the Unsigncrypt object, we get access to the cleartext message, the timestamp of the signcryption and its validity:

```
us = new Unsigncrypt(
    getPublicKeySender(),//Sender's Public Key
    getPrivateKeyReceiver(),//Receiver's Private Key
    bytes,//Signcrypted byte message
    settings);//Same configuration as sender

//Get the cleartext message
message = us.getStringMessage();

//Get the time message was signcrypted
signTime = us.getUnixTimeStamp();

//Check if the signcryption is valid
us.isAccepted();
```

6.1.2 Benchmark signcryption

Smart phones often have limited computational power, which in the worst case can make cryptographic functions unfeasible. Because of that, we have chosen to implement an EC signcryption scheme, which gives high security at low cost. To be sure our signcryption packet really can be taken into use on an Android smart phone, following the establishment of a **BenchmarkSigncryption** app in section 5.4.1, *Benchmark-Signcryption app.*

This app provides a simple implementation of the `no.altconsult.signcryption` packet and utilizes the `Benchmark` class we established in section 5.3.10, *Benchmark class*, to find the average run time, based on 10 runs with the same configuration. Screen shots of the BenchmarkSigncryption are shown for both Samsung Galaxy S and 5 in figure 6.1. Since the test phones are clean installations of Android, with no other processes



```
06 13:43:27 Europe/Amsterdam 2011)
--- Round0 stop---(Total program:33657)
...
--- Round 9 start---
Decrypted message:Lorem..(timestamp(Mon Jun
06 13:48:16 Europe/Amsterdam 2011)
--- Round9 stop---(Total program:31953)
---Bench Avg. of 10 rounds---
Decode Ascii85: 231 ms
Signcrypt: 17460 ms
Generate private key: 1 ms
Unsigncrypt: 14467 ms
Total program: 32203 ms
Encode Ascii85: 39 ms

---Settings ---
Field: P384
AES :key256bits
Message size: 1000 bytes
```

```
TextView
--- Round 0 start---
Decrypted message:Lorem..(timestamp(Mon Jun
06 12:08:42 CEST 2011)
--- Round0 stop---(Total program:3379)
.
.
.
--- Round 9 start---
Decrypted message:Lorem..(timestamp(Mon Jun
06 12:09:11 CEST 2011)
--- Round9 stop---(Total program:3195)

---Bench Avg. of 10 rounds---
Decode Ascii85: 14 ms
Signcrypt: 1755 ms
Generate private key: 1 ms
Unsigncrypt: 1439 ms
Total program: 3219 ms
Encode Ascii85: 9 ms

---Settings ---
Field: P384
AES :key256bits
Message size: 1000 bytes
```

Figure 6.1: BenchmarkSigncryption run on Samsung Galaxy 5 and Galaxy S

adding noise onto the results, the number of 10 rounds has been chosen as sufficient to get a clean average estimate of the performance. To see what impact the security level and the message size has on the performance, we have tested each device with different configurations. Tests have been performed with the EC fields P-192, P-256 and P-384 implemented in the packet. For each of the fields we change the message length between 10 bytes and 1000 bytes. When it comes to large messages of 1000 bytes, we also try to see if there are any difference in using AES with a 128 bits key compared to a 256 bits key. The results will be listed in tables, with average times given in ms. The column names of the benchmarks, tells which components of the signcryption that have been taken into consideration.

6.1.2.1 Samsung Galaxy S

If we look in the benchmark results table 6.1 for Galaxy S, the message length and the AES key size makes no difference. The Ascii85 coding increases a little with the message size, but almost negligibly compared to the overall run time of the program. Signcryption is shown to be slightly more expensive on all curves, about 7% on P-192 and P-256. The difference increases to 23% when P-384 is applied. The jump from the P-192 curve to P-256 increases the total run time by about 19%, while the jump from P-192 to P-384 increases the total run time by about 73%. The key size used in AES, does not seem to make any difference with respect to the run time.

Curve	AES	Message length	Signcrypt	Unsigncrypt	Encode Ascii85	Decode Ascii85	Total Program
P192	256 bits	10 bytes	947 ms	887 ms	2 ms	3 ms	1 842 ms
P192	256 bits	1000 bytes	965 ms	880 ms	6 ms	10 ms	1 868 ms
P192	128 bits	1000 bytes	956 ms	887 ms	8 ms	11 ms	1 865 ms
P256	256 bits	10 bytes	1 152 ms	1 029 ms	2 ms	3 ms	2 188 ms
P256	256 bits	1000 bytes	1 162 ms	1 025 ms	4 ms	18 ms	2 212 ms
P256	128 bits	1000 bytes	1 187 ms	1 025 ms	4 ms	24 ms	2 243 ms
P384	256 bits	10 bytes	1 773 ms	1 423 ms	2 ms	2 ms	3 208 ms
P384	256 bits	1000 bytes	1 755 ms	1 439 ms	9 ms	14 ms	3 219 ms
P384	128 bits	1000 bytes	1 756 ms	1 426 ms	11 ms	9 ms	3 215 ms

Table 6.1: Results from BenchmarkSigncryption app on Samsung Galaxy S

6.1.2.2 Samsung Galaxy 5

Table 6.2 shows the results for Samsung Galaxy 5. The run time for the total program is increased by a factor of 10. The signcrypt/ unsigncrypt ratio is approximately the same as in the Galaxy S. When it comes to Ascii85 coding, the performance is good for messages holding 10 bytes, which in reality is 10 bytes plus overhead of 48, 64 or 96 bytes with P-192, P256 and P-384. For messages holding 1000 bytes, the decoding part of Ascii85 seems to suffer. Here as well the key size of AES does not affect the run time.

6.1.2.3 Benchmark sum up

The benchmarks have pointed in the direction that our signcryption packet implemented in Java will have low end smart phones like Galaxy 5 struggle. On the other hand, with the high end smart phone Galaxy S, priced about three times higher on the Norwegian market, the performance makes it feasible to apply signcryption over the P-384 curve. With our security configuration of P-384 with AES-256, the sender side uses around 1,8 seconds to signcrypt and the receiver side 1,4 seconds to unsigncrypt.

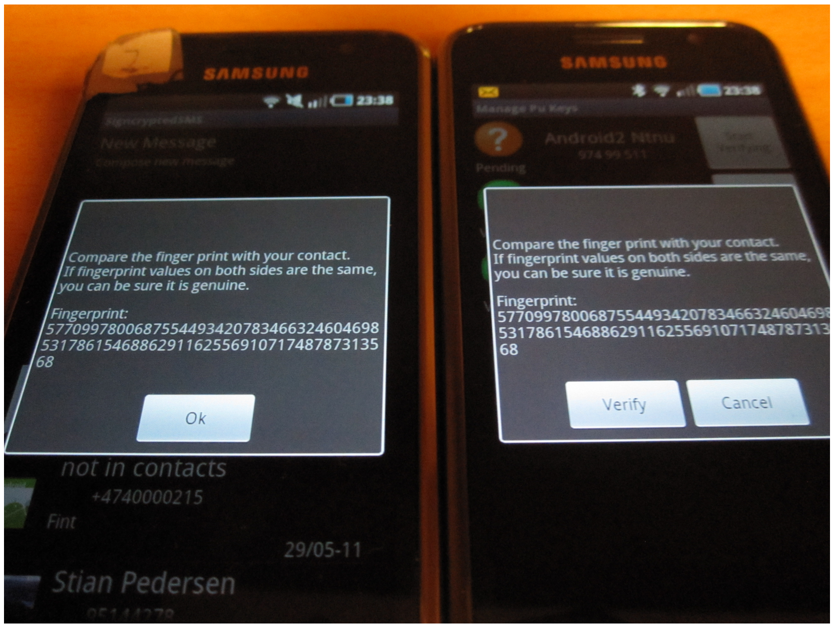
Curve	AES	Message length	Signcrypt	Unsigncrypt	Encode Ascii85	Decode Ascii85	Total Program
P192	256 bits	10 bytes	8 477 ms	7 995 ms	5 ms	8 ms	16 490 ms
P192	256 bits	1000 bytes	8 568 ms	8 038 ms	45 ms	186 ms	16 849 ms
P192	128 bits	1000 bytes	8 538 ms	8 002 ms	67 ms	213 ms	16 831 ms
P256	256 bits	10 bytes	10 701 ms	9 733 ms	8 ms	8 ms	20 456 ms
P256	256 bits	1000 bytes	10 923 ms	9 690 ms	42 ms	121 ms	20 783 ms
P256	128 bits	1000 bytes	10 904 ms	9 695 ms	33 ms	156 ms	20 796 ms
P384	256 bits	10 bytes	17 398 ms	14 572 ms	6 ms	10 ms	31 994 ms
P384	256 bits	1000 bytes	17 460 ms	14 467 ms	39 ms	231 ms	32 203 ms
P384	128 bits	1000 bytes	17 438 ms	14 463 ms	69 ms	205 ms	32 181 ms

Table 6.2: Results from BenchmarkSigncryption app on Samsung Galaxy 5

6.2 SigncryptedSMS app results

This part of the result contains everything that has been designed, implemented and analyzed until now and deployed as an Android app. All this a few clicks away for smart phone owners connected to the Android Market.

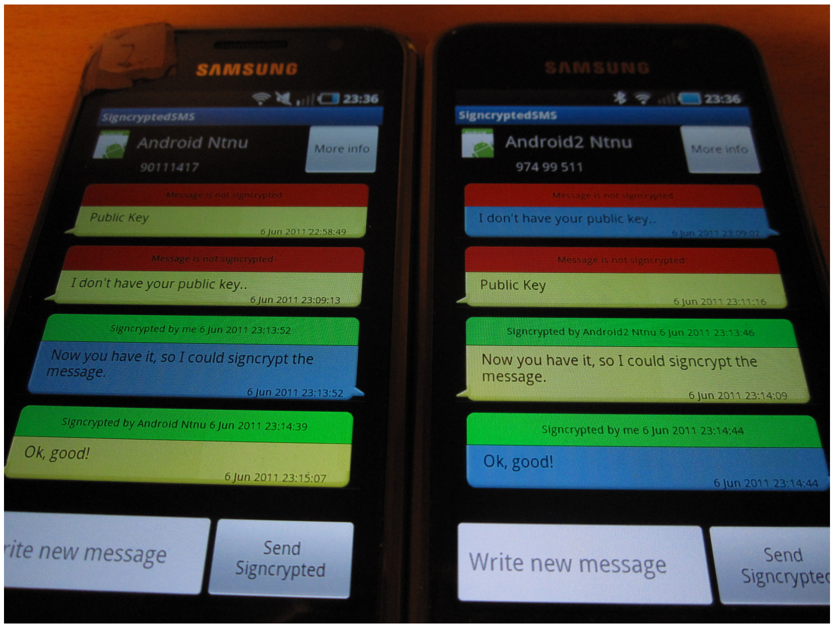
The implementation done in section 5.4.2, *SigncryptedSMS app*, covered almost all of the planning of the app, such as *use case diagrams* in section 4.2 and *mockups* in section 4.4. As we could understand in section 6.1.2, *Benchmark Signcryption*, Samsung Galaxy 5 was too slow to be able to use the signcryption packet. Fortunately The department of Telematics provided an additional Samsung Galaxy S for development and testing. Figure 6.2 shows verification of a Public key by fingerprint, sent from the left phone to the right phone. In figure 6.3 we show a test conversation between the phones before and after public keys have been exchanged. There are very few differences between the *Mockup* in section 4.4 and the GUI implemented. One difference is that the encryption of the private key has been disabled because of an unknown bug, this means non of the screens considering encryption of the private key will be shown in the app. Another difference is that the options menu in the main screen offers a choice called *My fingerprint*. The result of selecting this option is shown on the left smart phone in figure 6.2 where public key verification was demonstrated.



Compare the finger print with your contact.
If fingerprint values on both sides are the same,
you can be sure it is genuine.

Fingerprint:
577099780068755449342078346632460469853178615468862911625569107174878731356 8

Figure 6.2: Verification of a public key in SigncryptedSMS app



Android Ntnu
90111417

Message is not signcrypted
Public Key
6 Jun 2011 22:58:49

Message is not signcrypted
I don't have your public key..
6 Jun 2011 23:09:13

Signcrypted by me 6 Jun 2011 23:13:52
Now you have it, so I could signcrypt the message.
6 Jun 2011 23:13:52

Signcrypted by Android Ntnu 6 Jun 2011 23:14:39
Ok, good!
6 Jun 2011 23:15:07

Android2 Ntnu
974 99 511

Message is not signcrypted
I don't have your public key..
6 Jun 2011 23:09:02

Message is not signcrypted
Public Key
6 Jun 2011 23:11:16

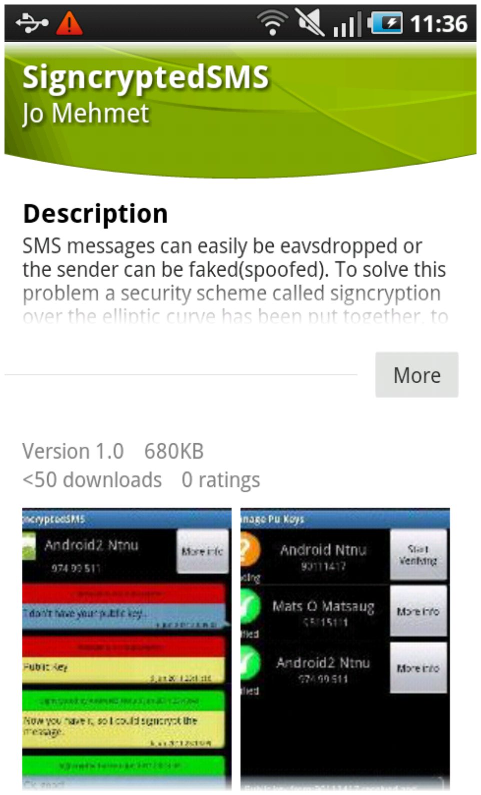
Signcrypted by Android2 Ntnu 6 Jun 2011 23:13:46
Now you have it, so I could signcrypt the message.
6 Jun 2011 23:14:09

Signcrypted by me 6 Jun 2011 23:14:44
Ok, good!
6 Jun 2011 23:14:44

Figure 6.3: SMS conversation utilizing the SigncryptedSMS app

6.2.1 Try the app

The SigncryptedSMS app has been posted on Android Market as shown in figure 6.4 and can be found by searching for the name SigncryptedSMS. The installation takes two clicks, **Install** and **Accept permissions**. Since this is the alpha version of the app and it has only been tested on two devices, the installation page is restricted to Norway with no promotion on the Android Market front page.



SigncryptedSMS
Jo Mehmet

Description
SMS messages can easily be eavsdropped or the sender can be faked(spoofed). To solve this problem a security scheme called signcryption over the elliptic curve has been put together, to

More

Version 1.0 680KB
<50 downloads 0 ratings

Figure 6.4: The Android Market installation page for SigncryptedSMS

Chapter 7

Discussion

As we now have gone through the full process of developing a system that protects SMS messages from end-to-end, we are going to discuss some important aspects of what we have achieved.

In section 7.1, *Choices of Design and Implementation*, we will discuss potential other choices of design. In section 7.2, *Security Analysis of Our Solution*, we will point out possible security flaws in our solution. At the end of this chapter we show in section 7.3, *Our solution compared to existing solutions*, we give a brief summary of what we have achieved in our solution, compared to other offers on the market.

7.1 Choices of Design and Implementation

7.1.1 Key length

In section 6.1.2, *Benchmark signcryption*, we saw that low end smart phones run signcryption more slowly. The difference in performance comparing P-192 to P-384 is approximately twice the run time using P-384 versus P-192 , this makes many seconds in difference when even the simplest security configuration runs slowly. *Top Secret* as security level can in many cases be an overkill. One time passwords are often valid for 15 minutes only, therefore secured content for many decades is not required. Our signcryption module makes it very easy to change the security configuration, and the users could be given the choice of what security level they want to use by sharing public keys having the desired security level. This can be implemented in the SigncryptedSMS app by using the version flag available in the `SigncryptionSettings` object to switch between security levels. The overhead would then consist of an extra set of key pair, one for each EC field.

7.1.2 Protection of message in the sent folder

When SMS messages are sent with the SigncryptedSMS app, they are stored in the *sent messages* folder as cleartext. The reason is that messages signcrypted with the sender's private key and the receiver's public key, only can be unsigncrypted by the recipient. The choice of storing the content as cleartext still holds for end-to-end protection of SMS messages. We could as an extra security option have chosen to protect the signcrypted messages in the *sent messages* folder by signcrypting them with the user's own private and public keys. That would double the amount of processing resources required for every sent message, but would secure the content with the private key encryption password if the phone gets stolen.

7.1.3 Aligning security and usability

We have chosen to design the SigncryptedSMS app to be in compliance with "security by designation", by interpreting the existence of a specific contact's public key as a request from the user to send signcrypted SMS messages. This way of interpreting the user's approach to security is concluded to be more secure by the article Aligning security and usability[Yee04]. A problem with this interpretion could be that some users would prefer to send unimportant messages as cleartext, as they wish to not waste money on overhead, or for any other reason. Another solution could be to give the user the choice between sending the message cleartext or signcrypted every time a new message is initiated.

7.1.4 Application of replay detection

In section 3.7, *Replay detection*, we reasoned that caching of message hashes is a low cost indicator to check for uniqueness. This was not implemented in the signcryption packet since disk storage of data is platform dependent. The application of replay-detection thereby has to be implemented by the developer of the app. To force developers using the package to implement a uniqueness check, the check could have been defined as an abstract method. In the implementation of the SigncryptedSMS app, only the presentation of the signcryption timestamp has been implemented. This was a matter of available time for implementation, and should be implemented in later versions of the app.

7.2 Security Analysis of Our Solution

Our solution consists of two parts. A signcryption module with implementation of an EC signcryption scheme which is proven to be secure, and an SMS client for Android that utilizes the module to protect SMS messaging. We want to analyze our final results for possible security flaws.

7.2.1 Public parameters

Our implementation has predefined EC fields from the standard FIPS 186-3[NIS09]. The curves defined in the standard has not been tested nor analyzed in this thesis, and may be considered secure since it is a US Federal Information Processing Standard.

7.2.2 Fuzz testing

Our implementation has not been tested for automated inputs of invalid, unexpected or random data called *Fuzz testing*. Consequently, there may be inputs that makes our solution crash.

7.2.3 Securing the private key

The private key is stored in a folder restricted by Android to be accessible only to the app. The flaw is that someone in possession of the smart phone can get *root access* and read the cleartext key from the restricted app folder.

A possible solution would be to encrypt the private key by symmetric encryption, as implemented in section 5.4.2, *SigncryptedSMS app*, but not enabled due to a bug.

7.2.4 Memory leakage

In the current version of the package, there are objects holding the private key as cleartext in the short time memory. Depending on the JRE configuration or the Operating System (OS), malicious programs could read the key from memory.

This problem is hard to solve, because security policies differ from one OS to another. A solution that improves the security would be to erase the considered objects from the memory once calculations with the private key are done.

7.2.5 Securing public keys

Public keys are stored as cleartext in the note field of a Contact. The key status is marked with cleartext <*pending*> or <*verified*> in front of the public key. These tags or the public key can be altered by malicious apps with Contact writing permission, an intruder that physically tampers with the smart phone, or via the default Contacts app. Most Android phones are by default synchronized with Google Contacts, this means public keys can be changed via a broken Google account as well.

A solution is to signcrypt public keys with the user's own private and public keys.

7.2.6 Public key renewal flood Attack

When new public keys are received, the current public key is replaced by the new one, and the status set to `pending`. If an attacker sends a bogus public key by SMS spoofing, the user will see that it is fake by verification and ask for the real key from the concerned contact. The attacker can repeat this replacement an unlimited number of times, which in itself classifies as Denial of Service (DoS) attack. The result could be that the user gives up verifying, and sends SMS messages in cleartext.

A solution is to disable automatic replacement of new keys, but that could imply that old keys does not get replaced.

7.2.7 Replay attack

Every signcrypted SMS message is marked with the timestamp for when it was signcrypted, but the user has to pay attention to the timestamp and compare it with the local received time of the message. Another problem is that there is no check to see if a received message is unique.

In case of a large time span between the signcryption time stamp and the received time, a notification should be made to the user. The messages should be tested for uniqueness as analyzed in section 3.7, *Replay detection*, and notified for every non-unique message to the user.

7.3 Our solution compared to existing solutions

We use [Str10, Section 3.2, Short Review of Existing Solutions] as an overview of existing solutions. We find that almost no implementations or commercial products are open source, which means there is no way to verify the alleged characteristics of the security services.

There is one open source project called ParandroidMessaging[Wal11] which serves secure SMS messaging by public key cryptography. Their project is based on checking out the original Android source tree and altering the original messaging client to include security. This app uses Diffie-Hellman key exchange with 1024 bits keys to establish a shared symmetric key between two contacts. The same shared symmetric key encrypts every message with AES-256. This solution will perform faster than our SigncryptedSMS app, since public key cryptography only is used as initial key exchange. A security flaw compared to our solution, is lack of forward secrecy, which means that if the shared secret key is disclosed all previous and future messages will be disclosed. Another lack of security compared to our solution is the inability to verify public keys. The proposal of ParandroidMessaging is to only reject public keys received at an unexpected moment.

There are many proposals of how to secure SMS, but none of these have been implemented, and there are many commercial products claiming to provide protection of SMS messaging, but few are open source. We consider it as a minimum to open the source to let users scrutinize the implementation, and be sure about what happens behind the scenes.

Chapter 8

Conclusion

We have analyzed different security mechanisms that can give authenticity, confidentiality and replay detection, in order to provide the best solution for our domain of SMS messaging.

With constrains on message length in SMS and computational power in smart phones, we have chosen to serve authenticity and confidentiality with a new EC signcryption scheme that has been proven secure, fast and with small overhead.

With the desire of providing a solution which is sufficiently secure for any private or public institution in the society, we have chosen a security configuration that complies with the definition of Top Secret content.

We implemented the scheme into a Java packet, called no.altconsult.signcryption, providing modularity to be imported into any Java application, a module that can easily be configured to different security levels.

With our signcryption packet, we made an SMS client for Android, called SigncryptedSMS, that applies authenticity, confidentiality and replay detection from end-to-end. This in a manner in which, if public keys are exchanged, the system interprets the user to require secure communication. Seamlessly the user achieves protected communication, with the same work flow as in unprotected SMS messaging.

All the work done has been made open source on `http://github.com/jomehmet`, this means anyone can inspect the code for security flaws as well as contribute functionality and ideas that may elevate the project to a higer level.

8.1 Future work

We have implemented a security package using a set of proven security schemes. Even if these schemes are proven to be secure, the implementation may contain undiscovered errors and security flaws. The package needs to be inspected by a third party with knowledge of information security.

The SigncryptedSMS app is in alpha version on Android Market and testing on different Android smart phones has not been done. With the crash report and feedback system provided by Android Market, the app can be improved for errors on a variety of devices. The most important security weaknesses such as saving the private key in cleartext and public keys without signatures should be fixed with high priority.

In the context of enterprises and public entities that want secure SMS communication with customers or citizens, importing the `no.altconsult.signcryption` packet to their systems, thus adding end-to-end protected SMS services will provide huge potentials for value-added services. If a two way authenticated web portal is available, the verification of public key fingerprints for both parts can be handled by the user. In this way, already established portals can increase in value, without adding labor, with public keys administered by the user himself.

Bibliography

[AS11] NeoSoft AS, Last accessed April 4, 2011.
http://www.neosoft.ch/products/emerg_tracking/detail.php?ID=1017&IBLOCK_ID=39.

[BF92] N. Borenstein and N. Freed. Mime (multipurpose internet mail extensions), June 1992.
http://tools.ietf.org/html/rfc1341.

[Blu11] BlueKrypt, Last accessed May 3, 2011.
http://www.keylength.com/.

[bou11] bouncycastle.org, Last accessed Mar 3, 2011.
bouncycastle.org.

[Cer69] Vint Cerf. Ascii format for network interchange, October 1969.
http://tools.ietf.org/html/rfc20.

[DH98] S. Deering and R. Hinden. Internet protocol, version 6 (ipv6) specification, December 1998.
http://www.ietf.org/rfc/rfc2460.txt.

[dif11] Last accessed June 15, 2011. Agency for Public Management and eGovernment (Difi)
http://www.difi.no/elektronisk-id/slik-skaffer-du-deg-elektronisk-id.

[ecl11] eclipse.org, Last accessed Feb 1, 2011.
http://www.eclipse.org/downloads/.

[fac11] facebook, Last accessed Mars 2, 2011.
http://www.facebook.com/.

[fra11] framtidshuset.no, Last accessed Mars 30, 2011.
http://www.framtidshuset.no/catalog/product_info.php?products_id=250.

[Gon93] Li Gong. Variations on the themes of message freshness and replay - or the difficulty in devising formal methods to analyze cryptographic protocols. In *In Proceedings of the Computer Security Foundations Workshop VI*, pages 131–136. IEEE Computer Society Press, 1993.

[Inc11a] Google Inc., Last accessed Feb 1, 2011.
Android SDK
`http://developer.android.com/sdk/index.html`.

[Inc11b] Google Inc., Last accessed Feb 1, 2011.
Install the ADT plugin for Eclipse
`http://developer.android.com/sdk/eclipse-adt.html#installing`.

[Inc11c] Google Inc., Last accessed Feb 1, 2011.
`http://www.android.com/`.

[Inc11d] Google Inc., Last accessed Feb 1, 2011.
`https://market.android.com/`.

[Inc11e] Unicode Inc., Last accessed April 5, 2011.
`http://www.unicode.org/`.

[KGP$^+$09] E. Karpilovsky, A. Gerber, D. Pei, J. Rexford, and A. Shaikh. Quantifying the extent of ipv6 deployment, 2009. S.B. Moon et al. PAM 2009, LNCS 5448, pp. 13 - 22.

[ME09] E. Mohamed and H. Elkamchouchi. Elliptic curve signcryption with encrypted message authentication and forward secrecy, January, 2009. IJCSNS International Journal of Computer Science and Network Security VOL.9 No.1 395-398.

[Mic11] Sun Microsystems, Last accessed Feb 1, 2011.
`http://java.sun.com/javase/downloads/index.jsp`.

[NIS01] NIST. Security requirements for cryptographic modules, May 2001.
`http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf`.

[NIS08] NIST. Secure hash standard (shs), October 2008.
`http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf`.

[NIS09] NIST. Digital signature standard (dss), June 2009.
`http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf`.

[NIS11] NIST. Digital signature standard (dss), May 2011. http://csrc.nist.gov/publications/drafts/800-57/Draft_SP800-57-Part1-Rev3_May2011.pdf.

[NSA10] NSA. Nsa suite b cryptography, Mars 2010. http://csrc.nist.gov/publications/drafts/800-57/Draft_SP800-57-Part1-Rev3_May2011.pdf.

[rev85] GSM Doc 28/85 rev2, June 1985. Services and Facilities to be provided in the GSM System.

[SMS11] SMSGlobal, Last accessed Mars 1, 2011. http://www.smsglobal.com/.

[Sta05] William Stallings. *Cryptography and Network Security (4th Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2005.

[Str10] Petter Andreas Strøm. End-to-end confidentiality and integrity protection of short message service messages over cellular wireless networks, December 2010.

[TBS08] M. Toorani and A. A. Beheshti Shirazi. SSMS - A Secure SMS Messaging Protocol for the M-payment Systems. In *Proceedings of the 13th IEEE Symposium on Computers and Communications (ISCC'08)*, pages 700–705. IEEE, 2008.

[Wal11] Erik Wallentinsen, Last accessed June 13, 2011. https://github.com/erikwt/ParandroidMessaging/.

[WT99] Alma Whitten and J. D. Tygar. Why johnny can't encrypt: a usability evaluation of pgp 5.0. In *Proceedings of the 8th conference on USENIX Security Symposium - Volume 8*, pages 14–14, Berkeley, CA, USA, 1999. USENIX Association.

[XxKfSq04] L. Xiang-xue, C. Ke-fei, and L. Shi-qun. Cryptanalysis and improvement of signcryption schemes on elliptic curves, May, 2004.

[Yee04] Ka-Ping Yee. Aligning security and usability. *Security & Privacy Magazine, IEEE*, 2(5):48–55, 2004.

[Zhe97] Yuliang Zheng. Digital signcryption or how to achieve cost(signature & encryption) (double left equal) cost(signature) + cost(encryption). In *Proceedings of the 17th Annual International Cryptology Conference on Advances in Cryptology*, pages 165–179, London, UK, 1997. Springer-Verlag.

[ZI98] Y. Zheng and H. Imai. How to construct efficient signcryption schemes on elliptic curves, October, 1998. Information Processing Letters 68 (1998) 227-233.

[ZXI11] ZXING, Last accessed May 15, 2011. http://code.google.com/p/zxing/.

Appendix A

no.altconsult.signcryption

This appendix covers relevant source code for the Signcryption packet.

The files can be accessed through the zip-file attachment or on GitHub(free open source host):

zip-file: no.altconsult.signcryption

https://github.com/jomehmet/no.altconsult.signcryption

The zip-file can be found by searching on the thesis name at http://daim.idi.ntnu.no/soek/index.php

A.1 AbstractSigncrypt.java

zip-file: src/no/altconsult/signcryption/AbstractSigncrypt.java

https://github.com/jomehmet/no.altconsult.signcryption/blob/master/src/no/altconsult/signcryption/AbstractSigncrypt.java

A.2 Signcrypt.java

zip-file: src/no/altconsult/signcryption/Signcrypt.java

https://github.com/jomehmet/no.altconsult.signcryption/blob/master/src/no/altconsult/signcryption/Signcrypt.java

A.3 Unsigncrypt.java

zip-file: src/no/altconsult/signcryption/Unsigncrypt.java

https://github.com/jomehmet/no.altconsult.signcryption/blob/master/src/no/altconsult/signcryption/Unsigncrypt.java

A.4 SigncryptionSettings.java

zip-file: src/no/altconsult/signcryption/SigncryptionSettings.java

https://github.com/jomehmet/no.altconsult.signcryption/blob/master/src/no/altconsult/signcryption/SigncryptionSettings.java

A.5 SigncryptionManager.java

zip-file: src/no/altconsult/signcryption/SigncryptionManager.java

https://github.com/jomehmet/no.altconsult.signcryption/blob/master/src/no/altconsult/signcryption/SigncryptionManager.java

A.6 FieldType.java

zip-file: src/no/altconsult/signcryption/FieldType.java

https://github.com/jomehmet/no.altconsult.signcryption/blob/master/src/no/altconsult/signcryption/FieldType.java

A.7 Fields.java

zip-file: src/no/altconsult/signcryption/Fields.java

https://github.com/jomehmet/no.altconsult.signcryption/blob/master/src/no/altconsult/signcryption/Fields.java

A.8 KeyLength.java

zip-file: src/no/altconsult/signcryption/KeyLength.java https://github.com/jomehmet/no.altconsult.signcryption/blob/master/src/no/altconsult/signcryption/KeyLength.java

A.9 SigncryptPacket.java

zip-file: src/no/altconsult/signcryption/SigncryptPacket.java

https://github.com/jomehmet/no.altconsult.signcryption/blob/master/src/no/altconsult/signcryption/SigncryptPacket.java

A.10 Cryptogram.java

zip-file: src/no/altconsult/signcryption/Cryptogram.java

https://github.com/jomehmet/no.altconsult.signcryption/blob/master/src/no/altconsult/signcryption/Cryptogram.java

A.11 AES.java

zip-file: src/no/altconsult/signcryption/AES.java

https://github.com/jomehmet/no.altconsult.signcryption/blob/master/src/no/altconsult/signcryption/AES.java

A.12 Ascii85Coder.java

zip-file: src/no/altconsult/signcryption/Ascii85Coder.java

https://github.com/jomehmet/no.altconsult.signcryption/blob/master/src/no/altconsult/signcryption/Ascii85Coder.java

A.13 Benchmark.java

zip-file: src/no/altconsult/signcryption/Benchmark.java

https://github.com/jomehmet/no.altconsult.signcryption/blob/master/src/no/altconsult/signcryption/Benchmark.java

Appendix B

no.altconsult.BenchmarkSigncryption

This appendix covers relevant source code for the BenchmarkSigncryption app for Android.

The files can be accessed through the zip-file attachment or on GitHub(free open source host):

zip-file: no.altconsult.BenchmarkSigncryption

https://github.com/jomehmet/no.altconsult.BenchmarkSigncryption

The zip-file can be found by searching on the thesis name at http://daim.idi.ntnu.no/soek/index.php

B.1 no.altconsult.BenchmarkSigncryption

B.1.1 BenchmarkSigncryption.java

zip-file: src/no/altconsult/BenchmarkSigncryption/BenchmarkSigncryption.java

https://github.com/jomehmet/no.altconsult.BenchmarkSigncryption/blob/master/src/no/altconsult/BenchmarkSigncryption/BenchmarkSigncryption.java

Appendix C

no.altconsult.SigncryptedSMS

This appendix covers relevant source code for the SigncryptedSMS app for Android.

The files can be accessed through the zip-file attachment or on GitHub(free open source host):

zip-file: no.altconsult.SigncryptedSMS

`https://github.com/jomehmet/no.altconsult.SigncryptedSMS`

The zip-file can be found by searching on the thesis name at `http://daim.idi.ntnu.no/soek/index.php`

C.1 MainActivity.java

zip-file: src/no/altconsult/SigncryptedSMS/MainActivity.java

`https://github.com/jomehmet/no.altconsult.SigncryptedSMS/blob/master/src/no/altconsult/SigncryptedSMS/MainActivity.java`

C.2 TypePrivateKeyPasswordActivity.java

zip-file: src/no/altconsult/SigncryptedSMS/TypePrivateKeyPasswordActivity.java

`https://github.com/jomehmet/no.altconsult.SigncryptedSMS/blob/master/src/no/altconsult/SigncryptedSMS/TypePrivateKeyPasswordActivity.java`

C.3 ChangePrivateKeyPasswordActivity.java

zip-file: src/no/altconsult/SigncryptedSMS/ChangePrivateKeyPasswordActivity.java

https://github.com/jomehmet/no.altconsult.SigncryptedSMS/blob/master/src/no/altconsult/SigncryptedSMS/ChangePrivateKeyPasswordActivity.java

C.4 ManagePublicKeysList.java

zip-file: src/no/altconsult/SigncryptedSMS/ManagePublicKeysList.java

https://github.com/jomehmet/no.altconsult.SigncryptedSMS/blob/master/src/no/altconsult/SigncryptedSMS/ManagePublicKeysList.java

C.5 ConversationListActivity.java

zip-file: src/no/altconsult/SigncryptedSMS/ConversationListActivity.java

https://github.com/jomehmet/no.altconsult.SigncryptedSMS/blob/master/src/no/altconsult/SigncryptedSMS/ConversationListActivity.java

C.6 SmsReceiver.java

zip-file: src/no/altconsult/SigncryptedSMS/SmsReceiver.java

https://github.com/jomehmet/no.altconsult.SigncryptedSMS/blob/master/src/no/altconsult/SigncryptedSMS/SmsReceiver.java

C.7 Signcryption.java

zip-file: src/no/altconsult/SigncryptedSMS/Signcryption.java

https://github.com/jomehmet/no.altconsult.SigncryptedSMS/blob/master/src/no/altconsult/SigncryptedSMS/Signcryption.java

C.8 ModelSMSThreadRow.java

zip-file: src/no/altconsult/SigncryptedSMS/ModelSMSThreadRow.java https://github.com/jomehmet/no.altconsult.SigncryptedSMS/blob/master/src/no/altconsult/SigncryptedSMS/ModelSMSThreadRow.java

C.9 ModelSMSmessage.java

zip-file: src/no/altconsult/SigncryptedSMS/ModelSMSmessage.java

https://github.com/jomehmet/no.altconsult.SigncryptedSMS/blob/master/src/no/altconsult/SigncryptedSMS/ModelSMSmessage.java

C.10 ModelPublicKeyRow.java

zip-file: src/no/altconsult/SigncryptedSMS/ModelPublicKeyRow.java

https://github.com/jomehmet/no.altconsult.SigncryptedSMS/blob/master/src/no/altconsult/SigncryptedSMS/ModelPublicKeyRow.java

C.11 ContactInfo.java

zip-file: src/no/altconsult/SigncryptedSMS/ContactInfo.java

https://github.com/jomehmet/no.altconsult.SigncryptedSMS/blob/master/src/no/altconsult/SigncryptedSMS/ContactInfo.java

C.12 main.xml

zip-file: res/layout/main.xml

https://github.com/jomehmet/no.altconsult.SigncryptedSMS/blob/master/res/layout/main.xml

C.13 row_message_thread.xml

zip-file: res/layout/row_message_thread.xml

https://github.com/jomehmet/no.altconsult.SigncryptedSMS/blob/master/res/layout/row_message_thread.xml

C.14 row_public_key.xml

zip-file: res/layout/row_public_key.xml

https://github.com/jomehmet/no.altconsult.SigncryptedSMS/blob/master/res/layout/row_public_key.xml

C.15 AndroidManifest.xml

zip-file: AndroidManifest.xml

https://github.com/jomehmet/no.altconsult.SigncryptedSMS/blob/master/AndroidManifest.xml