Università
della
Svizzera
italiana

Faculty
of Informatics

Bachelor Thesis

June 7, 2012

Secure and Compact SMS

Android application for advanced short messages features

Lorenzo Wölckner

Abstract

Short Message Service (SMS) in the mobile network is one of the most important and used form of communication in the world. Despite its very large distribution and volume of traffic, considering other developments in near fields, the service has not been worked on or improved much since its initial deployment: much can be done to enhance the current SMS status.

This project presents the development, ideas and features of a practical application for *Android* smartphones, aimed at overcoming existing SMS network limitations and provide extended functionalities. The main areas of development are two: text compression and network security; both developed features are combined and usable in an accessible way by any user.

The objective of the project is to provide increased communication capabilities via longer messages and more secure transmission of information, improving the user experience of SMS communication.

Advisor
Prof. Marc Langheinrich
Assistant
Marcello P. Scipioni

Advisor's approval (Prof. Marc Langheinrich): Date:

Contents

1 Introduction **2**

1.1 Outline . . . 2

2 Related work **3**

2.1 SMSzipper . . . 4

2.2 CryptoSMS . . . 4

3 Project description **5**

3.1 Objectives and milestones . . . 5

3.2 Development . . . 6

3.3 Tools . . . 6

3.4 Design . . . 7

4 Compression **8**

4.1 General description . . . 8

4.2 Dictionaries . . . 8

4.3 Analyzer . . . 9

4.4 Preprocessor . . . 9

4.5 PPM compression . . . 10

4.5.1 Description . . . 10

4.5.2 Implementation . . . 11

4.6 Testing and performance . . . 11

5 Security **15**

5.1 General description . . . 15

5.2 DSA - Digital signature . . . 15

5.3 RSA - Asymmetric encryption . . . 16

5.4 AES-PBE - Symmetric encryption . . . 16

6 SC-SMS Data **17**

6.1 Structure and flags . . . 17

6.2 Generation and retrieval . . . 17

6.3 Multipart messages . . . 18

7 Android integration **20**

7.1 Receiver component . . . 20

7.2 Sender component . . . 21

7.2.1 Message tab . . . 21

7.2.2 Security tab . . . 22

7.2.3 Manage keys tab . . . 22

8 Program structure **23**

8.1 Compression component . . . 23

8.2 Security component . . . 24

8.3 Data component . . . 25

8.4 Android component . . . 25

9 Possible developments **27**

9.1 Elliptic curve cryptography . . . 27

9.2 Additional key exchange channels . . . 27

9.3 Separate encrypted storage of messages and keys . . . 27

10 Conclusion **28**

A Terminology and abbreviations **29**

1 Introduction

Short Message Service (SMS) is, despite many new communication options, still one of the most important channels of communication available; with over 6.1 trillion messages being sent in the world during 2010 [1], the SMS traffic is currently continuing to grow. With the passing of the years, some problems and limitations are however becoming more and more evident. The network and channels used for SMS communications have been demonstrated to be not secure [2], it is easy to intercept, modify and spoof messages, the latter also with no special technology. The old limit of 160 characters per message does not make much sense nowadays, yet it is still kept and keeps bringing large profits to companies for what can be at this time considered a derisory service for the volume of data used.

Some features and technologies at the base of SMS cannot be changed for backwards compatibility reasons, others are simply not going to change due to companies' decisions; the service's usage is increasing more and more, yet users can (rightfully) feel limited and not protected by the current status of SMS.

The project developed and presented here aims at utilizing the current static technologies used for SMS communication at their fullest, to build on top of them an additional layer of security and compression features, to present the user new features that allow anyone to easily overcome the limitations of the current system. The application is built for the *Android* platform, the only requirement to be able to communicate with the newly given tools is that both the sender and receiver have a compatible mobile phone with the application installed. The program has been developed with ease of use in mind: security features are optional and easily setup in a couple of taps, compression is done automatically and optimized without user input, the user only needs to select the destination of the message and write the wanted text; everyone can easily gain access to and use the new technology.

1.1 Outline

This document will begin by presenting in Section 2 the present situation of tools currently available for similar purposes, their differences and features in regard to compression and security features applied to SMS traffic. Section 3 explains the project's goals and general design guidelines, together with an overview of the development process and work done, details are given about the different target solutions to mitigate the current SMS limitations.

The next four sections deal with the description and explanation of the four main areas developed and included in the application, respectively they are: Section 4 - text analysis and compression, including benchmarks and characteristics of application, PPM model of compression description; Section 5 - security algorithms and features, symmetric encryption (AES), asymmetric encryption (RSA), digital signature (DSA), together with a general description of implementation; Section 6 - SC-SMS data structure, generation and retrieval procedures, application of multipart message features; Section 7 - integration with the *Android* platform, related features and UI details.

The report continues with Section 8, where the specific structure of the program code, classes and functionalities are described, including the relations between them.

Section 9 which explains possible expansions and further developments that can be done on the presented application, extension of the security applications, different security algorithms, and additional communication channels, the document then ends with a conclusion and review of the achieved results in Section 10.

2 Related work

Available on the web there are several dozens *Android* and *iOS* applications claiming to bring either compression or security, however very few can be considered working as advertised and well documented. Plenty of developers claim their application simply “encrypts data in the most secure way”, or lets you “fit up to 400 characters in a single SMS”. These sentences however don’t make much sense or must be proven by documentation and data to be trusted. If we think about security we should know which algorithm is used, how are keys and passwords handled, maybe see the implementation and so on. For compression, stating the maximum amount of characters compressible to one message also doesn’t say anything; for the developed application presented here for example the claim that it could fit 1’000 (obviously specially chosen) characters in a 160-character SMS would be true, but wouldn’t mean much.

Two aspects have to be clear before an analysis of eventually currently available similar tools: the area and type of application of security features, and the application of text compression. As mentioned before, the security algorithms, specifically encryption/decryption and signing/verification, are specifically applied to protect data only during transmission until they are decoded/verified on the receiving end; the same thing is valid for text compression, data is compressed to occupy less space only for and during communication. The scope of the program is focused on the data going out and in the smartphone, not on different types of storage. This set of features makes the application pretty unique, it also gives the application opportunities to incorporate itself with different programs, which instead offer features related to secure or compressed storage and retrieval of data in memory.

Two applications were found which are well documented and present similar features to the project presented in this document, *SMSzipper*[1] and *CryptoSMS*[2]; the latter must not be confused with another program equally named *CryptoSMS* [3], which presents its security features in a very questionable and absolutely not reliable way). The first two products and their characteristics and features are briefly presented in the following subchapters; it is interesting to note that both the presented programs, even though recently updated and developed, are meant for older phones (Java ME) and are not available for *Android* or *iOS*. This section focuses not much about advanced user interface, but more about algorithms used and functionalities, general ideas which can be compared with the developed application.

Figure 1. *SMSzipper* screenshot

Figure 2. *CryptoSMS* screenshot

[1] http://www.smszipper.com/en/
[2] http://cryptosms.org/
[3] http://www.cryptosms.com/

2.1 SMSzipper

While the name may suggest only compression features, this application also gives the user the security option to perform encryption and decryption of SMS, together with other features not related to the scope of this project; a screenshot of the main menu of the application can be seen in Figure 1.

The main advertised feature of the program is its compression capability: as with the project presented in this paper, the application lets the users exchange special compressed messages to spend less money. While the usual (not so useful) references to maximum compression capabilities are made, an average compression performance of 50-55% is also mentioned, which (as seen later in Section 4.6) is very similar to the results obtained with SC-SMS. The precise algorithm used for compression is not disclosed.

As mentioned before, *SMSzipper* offers a basic security feature to its own special SMS messages: encryption. More specifically the application uses AES with a password to encrypt and decrypt messages for communication, and additionally stores the SMS in encrypted form on the phone. The encryption algorithm used is the same utilized in SC-SMS for symmetric encryption, even the used block chaining mode (see Section 5.4) is the same; the size of the keys used for AES is unknown.

This application offers features very similar to the ones developed and included in SC-SMS, it also gives the users additional options which are not part of the presented application. While it is not possible to accurately comment on the compression performance, it would seem both programs would perform similarly, with however the already available product having very few supported languages (and seemingly hard to add); for security, SC-SMS will provide additional features not included in *SMSzipper*.

2.2 CryptoSMS

CryptoSMS is a very good example of a well documented free open source program; it doesn't deal with compression, but provides various security features for SMS exchanges, plus other security measures not in the scope of this project. The program is based on its own personal folders for sent and received messages, the encryption used for SMS exchange is asymmetric, more specifically using elliptic curve encryption.

The used elliptic curve cryptography algorithm is very interesting and relatively new, it may bring various advantages and be actually better suited than RSA (Section 5.3) for the purpose of this project, the main features being good security with much smaller key sizes; more details about the argument can be found in the Section "Possible developments" (9).

The application works very similarly to the own presented program's asymmetric encryption (RSA) features: the user has at his/her disposition a special contact list to manage private and public keys, once keys have been exchanged, two persons can send each other encrypted SMS by following the asymmetric encryption rules (to send and encrypt, it is necessary to have the other person's public key). *CryptoSMS* also has an additional layer of encryption used to store the messages in the application's private inbox/outbox, the encryption used in this case is AES-256, the same used in the developed program for symmetric encryption for communication (see Section 5.4). An image of the principal interface of the application can be found in Figure 2

The *CryptoSMS* program presents part of the features already present in the own developed application, and a few interesting ideas for further expansion and addition of security features; more about the relevant needed ideas and algorithms will follow in Section 9 together with other features.

3 Project description

3.1 Objectives and milestones

The generic objectives of the project were very clear from the beginning; the development process was based on first the creation and testing of most of the features as standalone small programs, then the integration between functionalities, finally the porting to the *Android* platform. The final specific objectives can be resumed by the following sequence of points representing core (needed) and additional (optional) features of the final application; details and explanations about acronyms and algorithms used can be seen in appendix A or in the relative sections.

Core objectives

- **Integration and/or development of text compression**
  An algorithm for compressing the text of messages has been created, the function is composed by multiple steps (in this case analyzer, preprocessor and compressor) and must at least support the english language. A PPMC compression library has been used, combined with a custom analyzer and preprocessor. More about this argument can be found in Section 4.
- **Implementation of security features for text encryption**
  An easy to use set of functions has been developed, granting access to symmetric and asymmetric encryption. The two different types of algorithms implemented have been respectively AES with password based encryption, and RSA with a private/public key infrastructure. For the symmetric encryption, additional data and methods is used to check password correctness, for asymmetric encryption, a system for sending, storing and managing keys was created. More details can be seen in Section 5.
- **Development of a data format for the new SMS messages**
  A new type of format and rules for generation and retrieval must be created to handle and differentiate the new SC-SMS together with normal SMS. A set of special flag options have been developed, together with other systems and rules to generate valid formatted data for SMS traffic from bytes, and special functions to support multipart messages. Section 6 contains all the explanations and schemes about this subject.
- **Port of all the features on the *Android* platform**
- **Development of a complete and easy to use user interface**

Additional objectives

- Support for additional languages for compression
- Implementation of additional security features (DSA)
- Real-time UI feedback on status of compression and security
- Integration with default *Android* messaging features

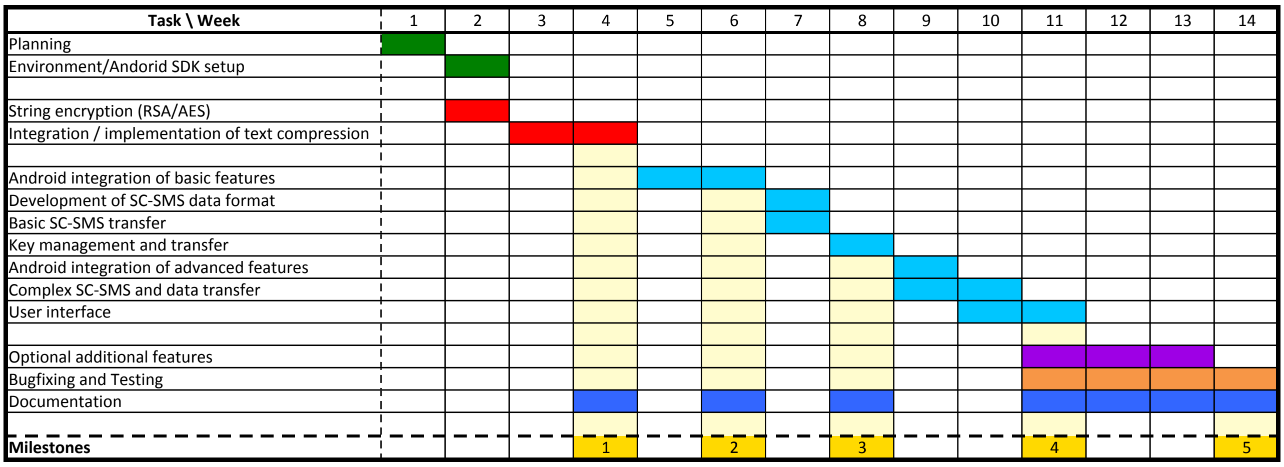
These objectives combined together in time brought to the definition and creation of the five following milestones, representing consequent stable (although limited) releases of the program. The features included are cumulative and the final milestone represents the last state of the product, with all the features implemented and ready to work.

Milestones

1. Standalone program for text compression and encryption/decryption
2. *Android* platform integration and combination of compression and security
3. Basic *Android* application for simple single SC-SMS transfer
4. *Android* application with advanced UI and complex SC-SMS transfer, key management
5. Final *Android* program, fully working and debugged, with possible additional features

3.2 Development

The most important data to look at to explain the development process is the plan used during the process, shown in Figure 3; the final plan presented here is only slightly different from the initial definition. 14 weeks were available to plan, develop, test and debug the project, which made a good planning and work strategy very important to concentrate the efforts and steadily improve the product. Regularly weekly meetings has been done with the assistant to ensure constant feedback and suggestions on the developed features and next steps to take.



Task \ Week	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Planning	🟩													
Environment/Andorid SDK setup		🟩												
String encryption (RSA/AES)		🟥												
Integration / implementation of text compression			🟥	🟥										
				🟨										
Android integration of basic features				🟨	🟦	🟦								
Development of SC-SMS data format				🟨		🟨	🟦							
Basic SC-SMS transfer				🟨		🟨	🟦							
Key management and transfer				🟨		🟨		🟦						
Android integration of advanced features				🟨		🟨		🟨	🟦					
Complex SC-SMS and data transfer				🟨		🟨		🟨	🟦	🟦				
User interface				🟨		🟨		🟨		🟦	🟦			
				🟨		🟨		🟨			🟨			
Optional additional features				🟨		🟨		🟨			🟪	🟪	🟪	
Bugfixing and Testing				🟨		🟨		🟨			🟧	🟧	🟧	🟧
Documentation				■		■		■			■	■	■	■
				🟨		🟨		🟨			🟨			🟨
Milestones				1		2		3			4			5

Figure 3. The final project plan

All the core objectives have been fulfilled, as well as the majority of the additional features. Not marked on the plan have been inevitable frequent small modifications to previous components to provide a fully working application combining all of the features.

The project first begins with a planning of the features and setup of the working environment, then standalone programs for text encryption and compression have been created, reaching the first milestone. After that, the integration with the *Android* platform together with new basic application features brought to the second milestone. The process continued by developing the data structures and functions needed to transfer the data on the GSM (SMS) network, with a basic interface for the program, reaching the next milestone and a working basic SC-SMS transfer system. Following that, advanced characteristics have been implemented, such as key management and an advanced user interface, after the implementation of additional features related to the transfer of big and complex messages, the fourth milestone has been achieved. The rest of the time has been dedicated to develop eventual optional elements, debugging and testing the application, and write most of the documentation. The final result is the last milestone: the final working project.

3.3 Tools

The project has been developed on *OSX Lion*, with the *Eclipse* IDE[4] for Java; the *Android* SDK[5] has been installed and integrated in *Eclipse* and used for most of the development process; for the initial part only Java has been used, to quickly program and test the standalone applications. During the first basic *Android* features development, the phone emulators included in the SDK have been utilized. To deal with the more advanced features, however (like multithreading), real phones have been used for debugging and testing, this is due to the much faster performance and need for real testing of network SMS communication.

The final application has been developed and tested to be compatible with *Android* version 2.2 (*"Froyo"*) and higher, thus ensuring a fully working experience for more than 99% of *Android* smartphone users.[6]

[4]Eclipse IDE - `http://www.eclipse.org/`
[5]Android SDK - `http://developer.android.com/sdk/index.html`
[6]Android, Platform Versions, Current Distribution - `http://developer.android.com/resources/dashboard/platform-versions.html`

3.4 Design

The general structure and design of the program can be divided into five different parts, each with its specific role and input/output data; a general diagram of the project's design can be seen in Figure 4. To begin with, there are two simple components: compression and security; each of them is responsible to simply take a sequence of bytes, some parameters, and return a sequence of bytes transformed according to the desired output. These components are independent and are not directly related to the *Android* OS or the mobile field.

The compression part takes the bytes coming from the message text, its parameter is only the dictionary used for compression, the result is the data resulting from the combined compression (see Section 4) of the preprocessor and the compressor. This component has also a separate part responsible for loading the needed dictionary data from the filesystem, from different files.

The security part is responsible for transforming the given data depending on the security options provided, encrypting, decrypting, signing and verifying text. Like the compression component, also in this case there is a separate part responsible of interacting with the filesystem, in this case in order to store and retrieve keys. More details about this part can be found in Section 5.

The third component of the program is responsible for the transformation, in both ways, of the data in byte form, from and to data in string form, compatible with the GSM standard; it must also calculate and add (or read and remove) the needed data structures to correctly identify a SC-SMS message. More details about this component can be found in Section 6.

The remaining two parts (contained in the Android phones in the diagram) are related to *Android* and the mobile interface; each has its type of user interface. One part deals with the user writing new messages and managing the keys, it is responsible for sending and asking the appropriate data with the previously described components. The other part is usually working in background and only interacts with the user when needed, it is responsible for receiving and correctly decrypting, verifying and decompressing any new incoming SC-SMS.

More details about the specific structure of components will be given in Section 8, when talking about the structure of objects and interaction of systems used, and in the following sections, with details about the single parts composing the system.

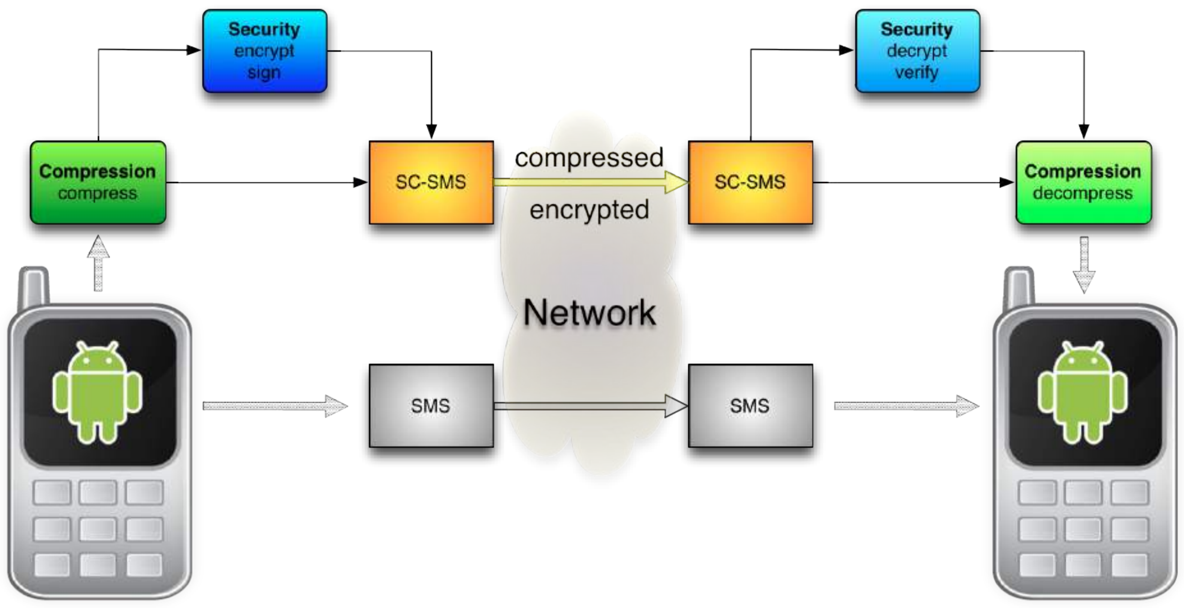


Figure 4. General scheme of the application's work

4 Compression

4.1 General description

Compression in the application is applied to the text given by the user as content of a SMS, the objective is to reduce as much as possible the message size while still operating in real time with the user input. The main benefit of compression is the overcoming of the very strict limits of SMS messaging (1120 bits per single message), letting users write more in less messages, needing less message parts and money to send long segments of text.

The compression procedure used works in multiple stages and uses text analyzers, preprocessors, dictionaries and PPMC compression (explained in Section 4.5) to ensure a very good compression rate; the general idea and scheme of the compression procedure can be found in Figure 5. The entire procedure can be divided in three parts: analysis and preprocessing of the text, pre-training of the compression model, final compression of the message. All these parts and their interactions will be explained in greater detail in the next chapters.

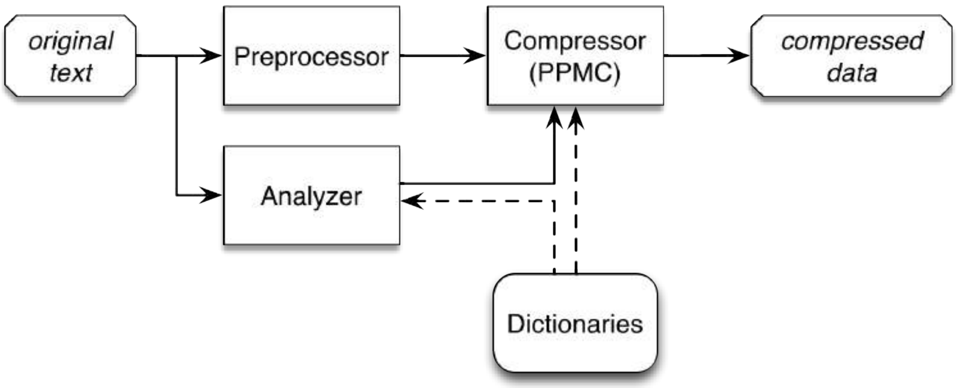


Figure 5. General scheme of text data compression

4.2 Dictionaries

The application includes dictionary files for 32 different languages; the dictionaries are lists of 5'000 words, ordered by frequency of use in the given language. The collection is composed by lists automatically built from statistics coming from movie subtitles, prepared by Dave Hermit[7]. The data is the only one of this kind and has been taken and adapted by removal of unused data to fit the needed purpose (for instance by removing single letter words). A small example of top words contained in the lists can be seen in Table 1.

While such lists may not accurately represent the real usage statistics after a certain point, they are probably the best and only collections representative of the language used in short text messages; this is due to characteristics which may seem "wrong" if the SMS context is not considered. The lists are characterized by colloquial styled words, presence of famous names of people and locations, words representing very frequent grammatical errors, which can be often found in short messages. The source of the data may seem unofficial and inaccurate; however, due to the way it was collected and the amount of different languages available, it seems the best candidate for the application.

	English	Italian	French	German	Spanish
1st	you	non	je	ich	de
2nd	the	che	de	sie	que
3rd	to	di	est	das	no
4th	and	la	pas	ist	la
5th	it	il	le	du	el

Table 1. Example dictionary top words for selected languages

[7] `http://invokeit.wordpress.com/frequency-word-lists/`

The application supports the main languages composed by characters that can be then successfully compressed by the following algorithms, namely Latin, Cyrillic, Slavic and Hebraic characters; the main languages for number of speakers (with the previously described restriction) were selected.
The full list of supported languages is the following:

- Albanian
- Bulgarian
- Croatian
- Czech
- Danish
- Dutch
- English
- Estonian
- Finnish
- French
- German
- Greek
- Hebrew
- Hungarian
- Icelandic
- Indonesian
- Italian
- Latvian
- Lithuanian
- Macedonian
- Norwegian
- Polish
- Portuguese
- Romanian
- Russian
- Serbian (Cyrillic)
- Serbian (Latin)
- Slovak
- Slovenian
- Spanish
- Swedish
- Ukrainian

The dictionaries are used both for analyzing text and recognizing the correct language to train the compression model (in reduced form). It should be noted that even if a language is not supported and present in the list, the compression will still work by selecting the default English language dictionary, granting a good working compression, although not optimal.

4.3 Analyzer

The analyzer deals with the recognition of the contents of the message, more specifically the detection of the language used; its decision is used to correctly initialize the compression model an ensure ideal compression rate for any possible message in the supported languages. It works by simply going through the given text word by word, checking if the sequence of characters exists in every language dictionary, keeping a counter. Once the message has been analyzed in all the possible languages, the results are compared: if one language dictionary had a number of hits that was superior by a certain threshold to any other language, it is selected as the text's language and used to initialize the compressor in the correct way for the given language. The procedure is very quick and is repeated in real time as the user types for every new character, to provide constant adjustments of the characteristics of the message and eventual language changes, for the best compression.

4.4 Preprocessor

The preprocessor is responsible for text analysis and optimization before the full compression, executed by analyzing and comparing the message text contents with known rules applicable to any language and modifying the text on their basis. The text optimization aims at getting a shorter sequence of characters, which can then be re-expanded and reverted back to the original message. The procedure is divided in three parts: *removal of redundant spaces*, *removal/insertion of spaces*, and *letter case transformation*, with only the last two stages executed in reverse order for the inverse process. It is important to notice that the process is not a bijection and text could be different after encoding and decoding; the process however follows syntax rules, ensuring that the final produced text, even if different, still maintains the same meaning and general structure, possibly correcting wrong spacings.

The *removal of redundant spaces* stage is very simple and, besides the obvious advantages, also ensures correct behavior of the next stages; this step is not needed when going back to the original text. What is modified in the message are the sequence of spaces: the string is trimmed and groups of two or more space symbols are reduced to a single space character.

The *removal of spaces* (respectively insertion when reversed) stage deals with symbols that necessarily must be followed and/or preceded by a space: when possible, that is when the text can be retrieved by reversing the procedure, the spaces are removed. The algorithm takes into account simple frequent symbols like dots and commas, but also smileys, an important part of SMS text, and applies rules and exceptions depending on the context.

The *letter case transformation* stage is responsible of automatically transforming characters to lowercase (or uppercase when reversed) after a specified set of symbols. This procedure, while not directly decreasing text size, ensures a better compression by reducing the size of the set of symbols used in the message.

The three specified steps in the preprocessor are executed before the compression phase; the process increases the compression ratio by decreasing the size of the message itself and better preparing the input for compression, it also has the advantage of producing "quality" text with correct syntax (related to spaces and capitals) for most communication situations, even when starting with badly formatted text typical of SMS.

4.5 PPM compression

4.5.1 Description

Prediction by Partial Matching (PPM) is a statistical method of lossless compression originally ideated by J. Cleary and I. Witten [3],[4], and further developed and implemented by A. Moffat [5]; based on symbol prediction and context modeling. Its ideas and algorithms are currently at the base of many of the best available text compression programs.

PPM works by evaluating the probabilities of a symbol to appear after other sequences of symbols; this has obvious advantages at compressing text composed by natural languages, as the rules of grammar and other properties of the letters used make possible to find exploitable structures. In PPM each encountered symbol is encoded depending on its context, the *n* symbols preceding it; the maximum size of the context is the *order* of the algorithm and is directly linked to its complexity and compression capabilities. If the symbol and context are found in the context model, the *total*, *left* and *right* probabilities on the *probability line* of the match are returned, otherwise the order gets decreased until a match is found.

To generate the compressed stream of bit, for each input symbol, the algorithm will assign a certain probability and encode it with it by an adaptive arithmetic encoder. The arithmetic encoder, given a set of symbol probabilities, returns a set of bits that encodes them; while the arithmetic decoder, given a set of bits, returns a number indicating the decoded symbol on the probability line.

The high level scheme of the whole PPM compression and decompression algorithm with the relative followed procedures, can be observed in Figure 6.

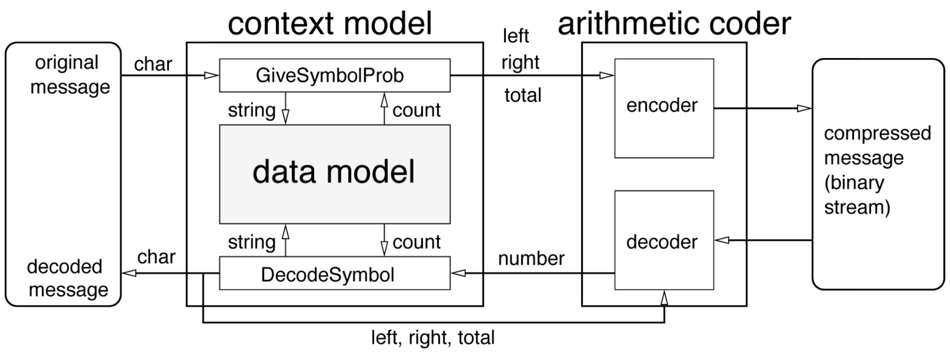


Figure 6. PPM compression model scheme (*from [6]*)

To calculate probabilities, PPM implementations can have two kinds of context-based statistical models: static and adaptive; both with the objective of assigning probabilities based on the symbol itself plus the informations about its context, depending on previously known data. The static model is simpler and less effective, it consists of fixed pre-calculated tables of sequences of symbols, each with its nonzero probability; whenever a new symbol is encountered, the entry of the symbol with its context is read from the table and the probability is assigned. The adaptive model, as the name suggests, can start with the same predefined probabilities, which could also be empty in this case, but the probabilities get updated every time a new sequence of symbols is read. This makes the model slower but leads to better compression thanks to its adaptation to the given data.

The PPM decoding part needs to know when to switch to a shorter context in order to retrieve the correct symbols, correctly using the data provided by the encoder; to do this a special *escape symbol* is used to mark a reduction in context size. The escape symbols are written in the encoded data whenever the encoder needed to decrease the size of the context, and is read by the decoder to replicate the context reduction actions and correctly retrieve the compressed data. The way to assign probabilities to the escape symbol, due to its high frequency, is an important part of the algorithm; different methods are often the reason of different names for different PPM variants, written as PPMX. Some known variants are PPMA, PPMB, PPMC, PPMP, ...

More details, including an in depth description of the algorithm with more examples and schemes can be found in [7], with a chapter entirely dedicated to this method.

4.5.2 Implementation

The PPM algorithm utilized in this project is the *PPMC* version, one of the best standard variations of the original algorithm which differentiates itself from the others by how escape symbols are handled; the model used is adaptive. The original Java library used in the application has been developed by Dr. B. Carpenter[8], minor additions have been made to provide easier access to the needed compression functions and faster training.

The way the symbols and probabilities are assigned in PPMC is as follows: each already encountered context is put in a section of the model together with the escape symbol, the frequency for the escape symbol is initially 1; whenever the same context is encountered again, a new entry with frequency 1 is created if the symbol is new, or a matched frequency value is increased by 1. The probability for each symbol with that context (including the escape symbol) is then the frequency for that symbol divided by the total frequencies. The rank (context length) used for PPMC is fixed, in this specific application set at *3*: after a series of tests this number resulted to be the best compromise between speed, memory used and compression achieved at various sizes of input text, as shown in Section *Testing and performance* (4.6).

PPM is better used as a compression algorithm for long segments of text, this due the relatively bad compression rates for very small amounts of characters, as demonstrated by S. Rein, C. Gühmann and F. Fitzek [6] together with techniques to reduce this effect. Better performance for short messages can be however achieved by utilizing the knowledge of structure and language of the input text and pre-training the model with dictionary data before the compression. Thanks to the analysis of the preprocessor it is possible to detect with a reasonable level of certainty the language used in the text; this can be updated in real time to adjust to the user input. The given language can then be used to initialize a new PPMC model, specifically built to improve the compression for the current and upcoming text by exploiting the characteristics of the selected language.

The way PPMC is initialized is as follows: we “train” the model by reading a certain amount of words contained in the detected language dictionary of most frequent words and passing them to PPMC (note that a prefix space is present); this will make the model build the tables of probabilities tailored for that specific language for most of the possible situations that can be encountered in the real message text. Utilizing 1’000 words for the training of each language seems to be the best solution for time needed for initialization of the model and compression performance.

The trained PPMC model runs continuously on a separate thread, activating whenever the text input by the user changes to provide almost real-time compression and feedback about the current size status of the message. Whenever a different language is detected the current model is discarded and the new model is built and trained for the new language. Due to the adaptive model used by the implemented algorithm, the model is saved as a copy on initialization, and reset to the initial state every time a new compression of the text is required. More details about the single objects composing the implementation of the compression method can be found in Section 8.1.

4.6 Testing and performance

To test the compression capabilities of the application, a collection composed of real SMS messages has been used: the NUS (National University of Singapore) SMS Corpus[9], composed of 51’434 English messages collected all over the world by volunteers; one of the few collections of this kind. While the corpus should contain only messages in English, a small percentage of SMS are written in other languages; the list of messages has been passed to a basic filter to eliminate duplicates and messages in obviously different languages using other alphabets, resulting in a final corpus of 41’765 SMS used for the benchmark. It is to be noted that the list still contains a small part of messages in other languages (which are all interpreted for this benchmark’s purpose always in English) and many messages full

[8]`http://www.colloquial.com/carp/`
[9]`http://wing.comp.nus.edu.sg/SMSCorpus/`

of abbreviations, a kind of worst scenario situation, that still shows very good results.

There are a few variables which are interesting for comparing input data and results, following is an analysis of their values and effects. The compression is applied to messages of length measured in bytes from text in UTF-8 encoding form, the compression amount is measured as a number representing the size of the compressed data in bytes, divided by the original length (a lower number means a better compression). There are two parameters to tweak in the algorithm used in the application: the context length/order, that is the number of characters looked at by the PPMC model; and the size of the dictionary of words passed for the training of the model. The preprocessor is a separate feature, which also has an effect on the final size and quality of compression when enabled.

The first feature looked at in detail is the order of the model used. As explained before the context/order of the PPMC model sets the maximum amount of characters looked at to make the prediction of the next upcoming character. By increasing the order the time needed for realtime compression (negligible for the small amount of text) and training increases. If the context is increased, the compression performance of the algorithm usually improves, but only up to a certain point: after the "limit" has been reached, increasing the order only brings to worse compression due to the continuous fallbacks to the escape symbol. The order also influences the complexity and size of the model object created on training; this in turn affects the time needed to restore the original trained model once a compression has been made. The context length variable clearly is a compromise between speed and compression quality; following is a chart of the varying compression capabilities at varying message sizes at different orders.

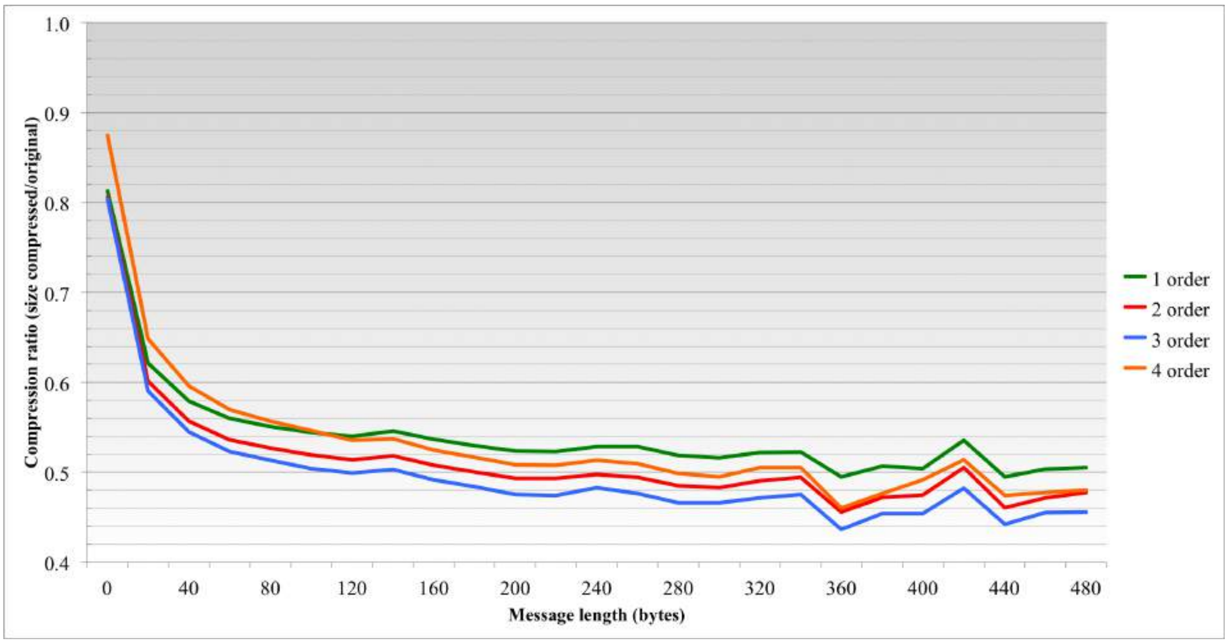


Chart 1. Average compression performance in relation to context length

From Chart 1 it can clearly be seen that, from the initial order 1, the compression performance increases by augmenting the order up to order 3, for all the message lengths. Starting from order 4 the performance degrades, bringing no benefits to the previously found best order and actually resulting in worse compression than the first order for short messages. Compared to order 1, a context of length 3 improves the compression performance by around 5%. Due to the good compression and relatively short (acceptable) time for training and small model size, order 3 has been chosen as the preferred context length for the PPMC model.

The next variable analyzed is the amount of words from the dictionaries used for the training of the model; while its effects are more limited than the choice of the order, it still has a part on the time needed to train and the final compression ratio. Starting from no words used, where the model will actually not be trained and will not know what to expect, increasing the words used for training will allow the PPMC model to build the probability tables more accurately depending on the characteristics of the dictionary used, leading to better compression. Like for the order variable there is however a "limit", which once surpassed will lead the model to have many useless and uncommon language characteristics stored, not concentrating on the common words and letter probabilities composing the language, thus leading to worse compression ratios.

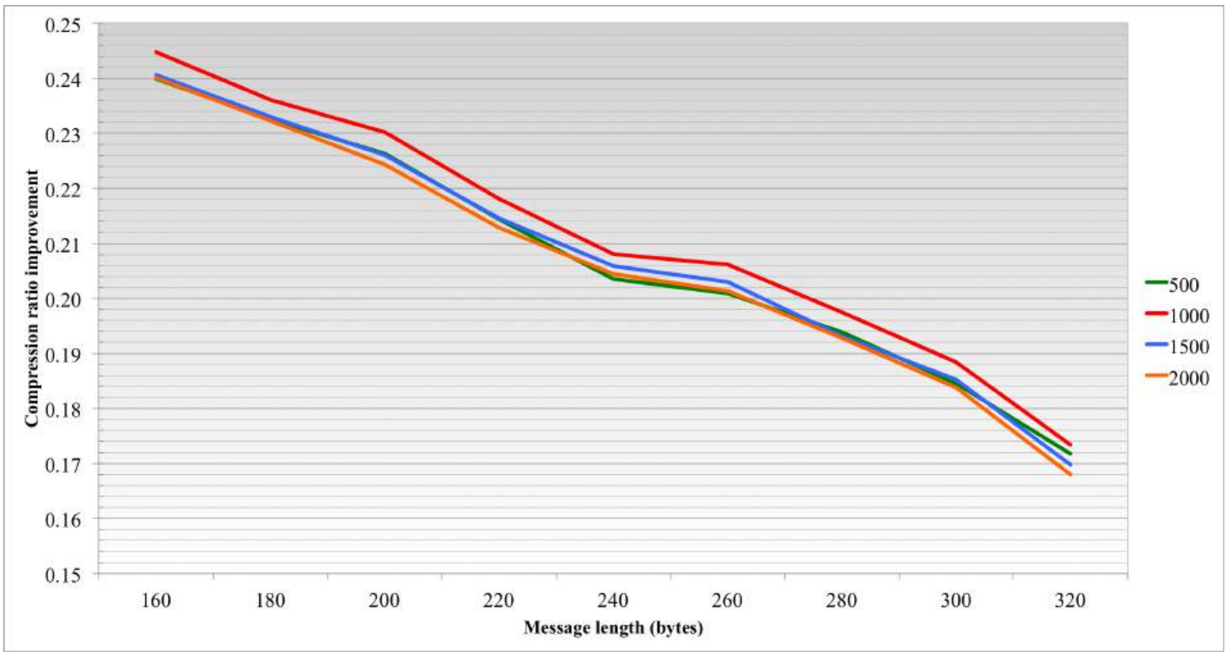


Chart 2. Average compression improvement from no dictionary in relation to dictionary size

Chart 2 represents the average improvement in compression performance with different amounts of words used for training, compared to no training at all. The chart focuses on a segment of message sizes: this is only done to better differentiate the results for visualization purposes, almost identical results can be found throughout the various sizes. Two things can easily be noticed: there is a clear improvement for all sizes if compared to absolutely no training, and the difference between the results of different amount of words being used is not big. It can be seen that the best average improvement is reached with 1'000 words taken from the dictionary and being used to train the compressor, this is the reason why this number was chosen to be used in the application, also due to the small time difference if compared to the time to load 500 words. Bigger amounts of words used only worsen the compression performance and result in more time to train the model, which also becomes uselessly bigger.

The last variable analyzed is the own developed preprocessor: the difference in performance it causes by being active or not. The preprocessor was built to increase the compression performance at almost no time cost by modifying the text before compression, it has no real drawback except a negligible amount of additional time used before the compression step.

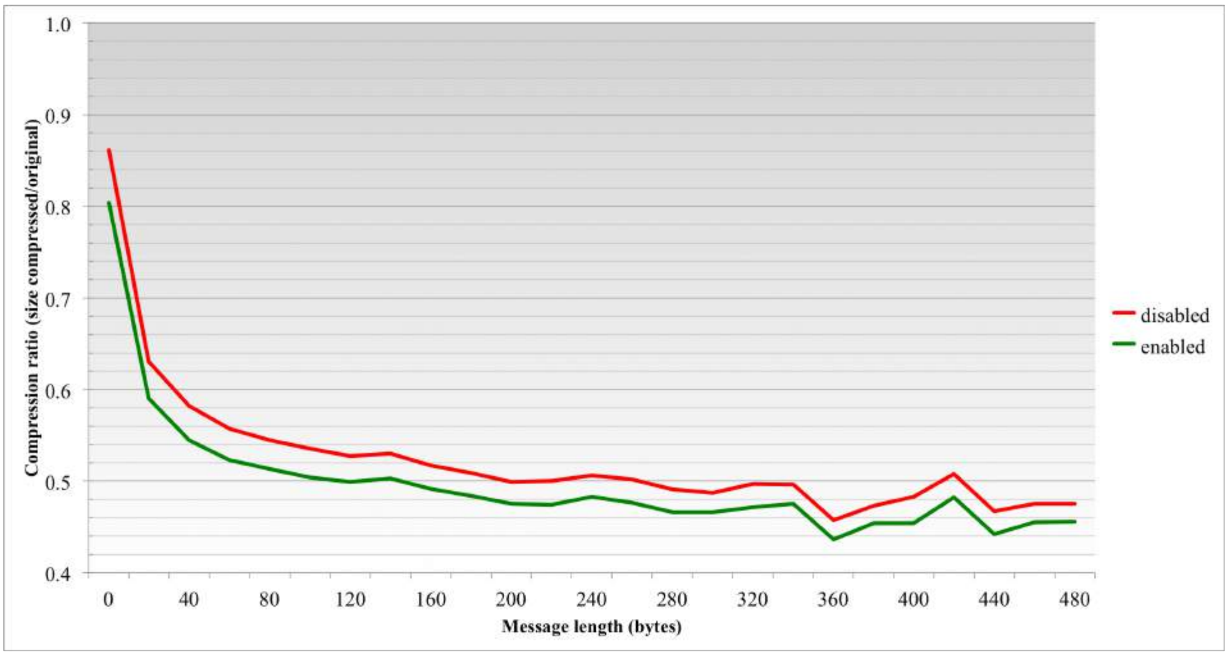


Chart 3. Average compression performance in relation to preprocessor utilization

In Chart 3 it can clearly be seen that the preprocessor improves compression rates for all message sizes, the improvement is around 2-2.5%. Given the almost inexistent disadvantages of having the preprocessor active, plus also the added (indirect) benefit of resulting in more syntactically correct messages, it is clear that the preprocessor is a positive feature to always have enabled.

We can now have a look at the resulting combined parameters and results of the compression used in the application. The PPMC model is initialized with a context of length 3 and trained with a list of 1'000 words for the needed language; it then receives text data modified by the preprocessor and performs the compression. The following chart shows the minimum, average, and maximum compression results for the messages in the corpus.

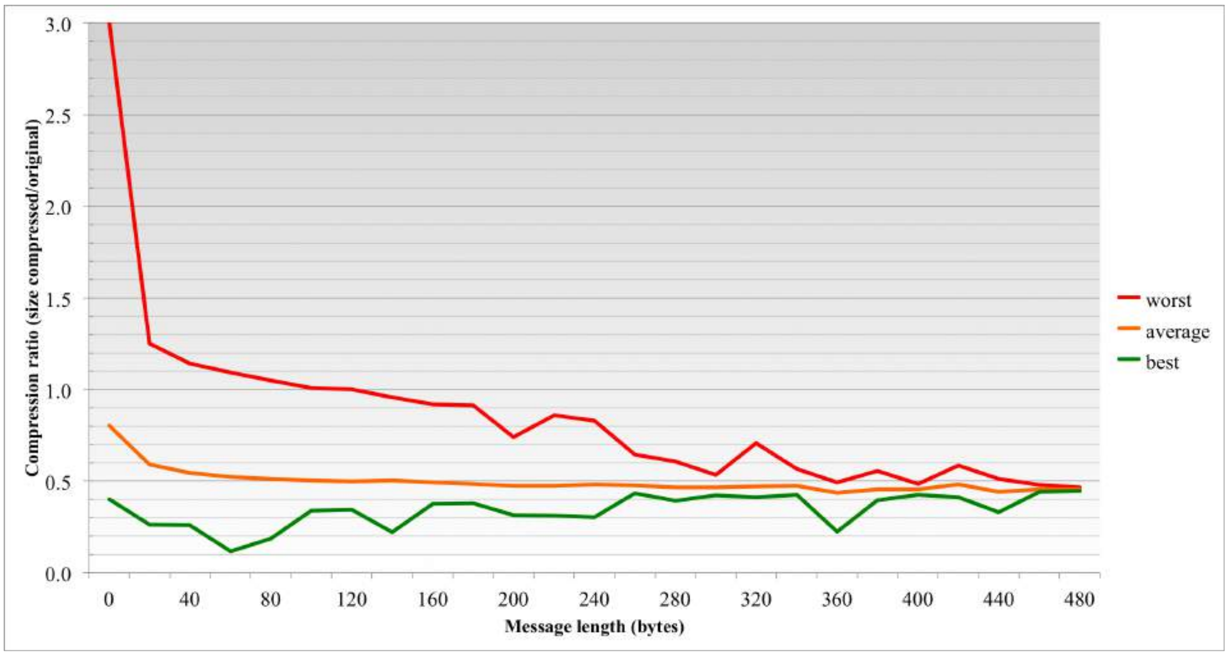


Chart 4. Compression performance of the final algorithm

By looking at Chart 4 the compression performance behavior can be noticed pretty well: the compression rate starts with bad results at very low message sizes (although still lower sizes than the original on average) but quickly improves. An average compression ratio of 0.5 can be found already at around 80-100 bytes, this compression performance then very slowly but steadily improves the more the messages get bigger. It should be noted that the alternating results at larger sizes are due to the corpus having a lot more short messages compared to the long ones. The best and worst lines also slowly converge the more the SMS gets longer.

Thanks to these performance results, the compression can be said to really help to reduce the amount of messages needed to write and send long text communications, with an usual compression of 50% already achieved after 100-120 characters. It is common to be able to now fit around 320 characters of text in each single SMS (where before the limit was 160), the longer the message, the better the compression.

5 Security

5.1 General description

The security features implemented in the project are all aimed at preventing attacks on the not-so-secure communication channel. The objective is not to encrypt data stored on the device, but to provide security features for the transmission of the data, protecting from eavesdropping, modifications, and spoofing. The network used for SMS transmission has already been demonstrated as not secure [8]: it is possible to intercept, read and modify messages; while faking SMS sources is even easier and widespread. There is currently no plan to upgrade the current structures and protocols, in any case the developed features aim at providing an additional layer of security independently from the underlying structure.

As mentioned before, there are two main types of “attacks” that must be prevented, each one with its own related security measure. The first one is intercepted or modified data, with loss of privacy; the solution to that lies in encryption through one of the two main methods: symmetric encryption via a known password, or asymmetric encryption via private and public keys. The second one is message source spoofing, or authentication; this is solved thanks to a digital signature and verification algorithm.

The security features are implemented so that they can be combined together as the user wants, providing the needed security for each message sent. The full security process goes through digital signature, asymmetric encryption, and symmetric encryption in this order when creating a new SC-SMS, as also shown in the diagram in Figure 7; the process is reversed when the data is received. All the specific algorithm implementations used for encryption, signatures and hashes are the ones already built into Java’s security packages, utilizing *BouncyCastle*. For more details about the single class components, refer to Section 8.2.

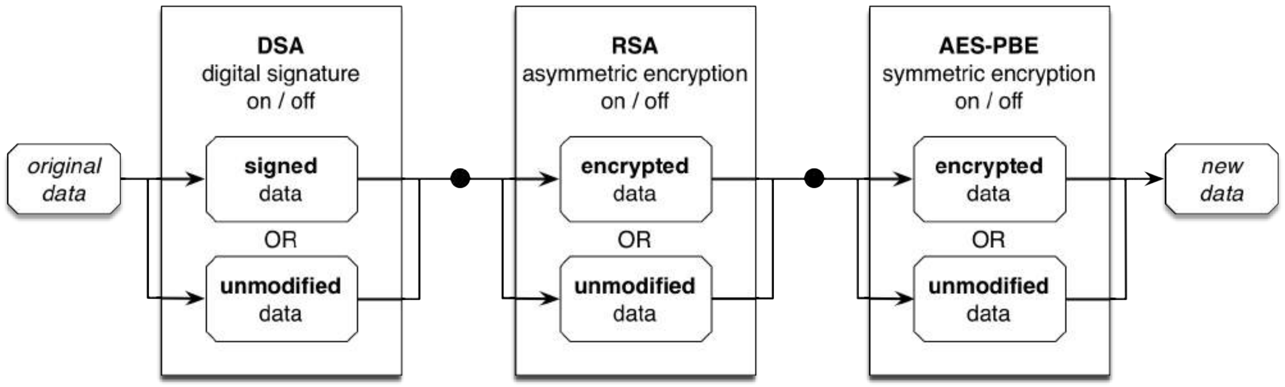


Figure 7. Data security implementation scheme

5.2 DSA - Digital signature

The Digital Signature Algorithm (DSA) is a standard for digital signature originally proposed by the National Institute of Standards and Technology (NIST) in 1991 and inserted in the Digital Signature Standard (DSS) in 1993, its last revision was published in 2009 [9]; it is based on the mathematical difficulty of calculation of discrete logarithms.

The algorithm can be divided in two parts: key generation and message signature/verification. In the first part a special couple of private and public keys (numbers) is generated; in the second one the available key is used for signing (private key) or verifying (public key) a message. The overhead caused by this security mechanism is fixed at 46-48 bytes and does not depend on the length of the message.

The sender of the message takes the original text he/she intends to send and calculate a hash of it, in this case SHA-1 is used, to get a 160-bit number; the hash is then used with the private key in a series of mathematical operations to calculate and obtain the signature of the message. The signature can only be calculated by the true sender, and is concatenated with the original message, proving the authenticity of the message by any receiver which can use the sender’s public key to verify it.

5.3 RSA - Asymmetric encryption

The RSA algorithm is a public-key cryptography method of encryption, originally ideated by R. Rivest, A. Shamir and L. Adleman in 1978 [10]. It is based on the mathematical difficulty of the large number factorization problem and "RSA problem"; it is nowadays the standard for asymmetric cryptography and communication of data on insecure channels.

Like DSA, the RSA function can be divided in two parts: private and public key (numbers) generation, and data encryption/decryption. In this case the public key (given by the receiver beforehand) is used to encrypt a message, while the private key is used to decrypt the data. RSA lets users distribute public keys, which can only be used for encryption, so that any message encrypted with them can be decrypted only by the correct receiver: the initial user which generated the keys. The overhead due to this type of encryption is of 11 bytes for every 117 bytes of initial data, with (eventually padded) blocks of 117 bytes generating blocks of 128 bytes.

Since RSA is used to encrypt data, hashes cannot be used; instead all the message is passed one block at a time to the RSA function, in order to be encrypted with the private key. The result, in the implemented case with RSA keys of 1024 bits, is a sequence of blocks of 128 bytes generated from blocks of 117 bytes (due to data used by the standard PKCS#1 padding [11]), representing the encrypted original data, which can only be decrypted with the correct corresponding private key. RSA-1024 is nowadays the most used algorithm for asymmetric encryption, with currently no known cracking of key; it is considered secure for at least several years.

5.4 AES-PBE - Symmetric encryption

The Advanced Encryption Standard (AES) is a worldwide used specification/standard for encryption of data, and equally indicates the *Rijndael* algorithm for symmetric-key encryption. Originally developed by J. Daemen and V. Rijmen in 1998 and submitted in 1999 [12], the algorithm resulted the best of all the submitted algorithms in the public AES selection process; it was officially announced as the new block encryption standard in 2000 by USA's National Institute of Standards and Technology (NIST).

Password-Based Encryption (PBE) is used in the application, combined with AES, to provide stronger security. Part of *RSA Laboratories' Public-Key Cryptography Standards* [13], PBE is meant to generate a cryptographic key from a text password, which is then used to actually perform the encryption by AES. The PBE used implements PBKDF2 (Password-Based Key Derivation Function), a standard function for key derivation which utilizes, in this case, HMAC-SHA1 with a fixed 8-byte salt value to produce a 256-bit cryptographic key. HMAC-SHA1 is a Hash-based Message Authentication Code, which stands for the construction of a MAC [14] using the SHA1 [15] hash function.

As explained before, everything starts with a user-supplied password string: the string goes through a fixed amount of cycles of HMAC-SHA1, generating a 256-bit key which is then used by AES. The encryption works by taking blocks of 128 bits of data, for each block AES encryption is applied for 14 rounds with the 256-bit key, chaining operations between blocks together using the Cipher-Block Chaining mode (CBC). If the data passed to the algorithm is already of the correct multiple-length due to previous RSA encryption, padding is not needed; otherwise PKCS#5 padding [13] is used for filling the remaining data in the last block. Due to the implementation and characteristics of the padding, it can happen that the padding is correctly reconstructed on decryption, even with a wrong key, incorrectly generating wrong bytes of decrypted data (this is not rare with few padding bytes). In order to avoid that, before encryption the initial data is passed to a digest function, calculating the SHA-256 hash of the given data; then the initial 4 resulting bytes are taken and appended to the original bytes and the resulting byte array is encrypted. On decryption the checksum bytes are taken and compared to a newly calculated hash of the decrypted data, testing if the decrypted data is correct, thus if the given key was the right one.

The total overhead caused by this form of encryption is as follows: 16 bytes for the IV and 4 bytes for the checksum data are fixed and always added, data must be then padded to get full 32 byte blocks. The final result of the procedure is a sequence of 128-bit blocks of data, representing the encrypted original source (plus checksum). AES-256 is considered one of the most secure algorithms for symmetric encryption, with many tests executed and still in progress, there is currently no known practical attack or even sign of a possibility of it; this form of security is thought usable and secure for at least a couple of decades.

6 SC-SMS Data

6.1 Structure and flags

A Secure and Compact SMS (SC-SMS) by itself is a sequence of bits, not directly referable to a string of characters; the composition of the data contained in a SC-SMS is as follows. Two bytes of special data precede a sequence of bytes of indefinite length: the actual contents of the message (plus additional multipart markers, seen later); these first bytes compose the flags of the message and define what the following bytes are and how they should be read, the structure of the flags is shown in Figure 8.

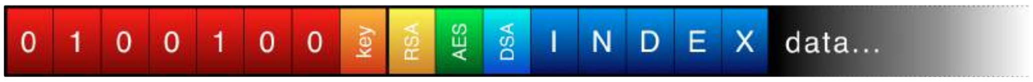


Figure 8. Structure of the SC-SMS flag bits and data

The first seven bits compose the special sequence that indicate that the message is a SC-SMS and that the following data should be read and processed accordingly. The reason why seven bits are used lies in the default SMS encoding for text, the GSM 03.38 standard [16], which uses 7 bits per character. The chosen 7 bits represent the character “¤” (`0x24`), a very rare character which should usually never be used as first character in a normal SMS; when this character is detected, the next bits are interpreted consequently as a SC-SMS message. The eighth bit in the first byte tells if the upcoming data represents a text message or a public key, used for RSA and DSA security; in that case the next flag bits are ignored and the incoming bytes are read and stored as a key object for the source phone number.

In the second byte, the first three bits are used to respectively tell if the data uses RSA encryption, AES encryption, and DSA signature; all the different modes can be used together in any combination. The last five bits indicate the index of the dictionary used for compression (basically the main language of the original text), this data is used to correctly initialize the PPMC model to decompress the given data and retrieve the original message.

6.2 Generation and retrieval

The user has two main choices that decide what will be sent in the SC-SMS: he/she can send a public RSA or DSA key, or send a text message then compressed and with added optional security features.

In the first case the generation of the SC-SMS containing the key bytes is easy, the flag section will simply contain the marker symbol and the eighth bit set to **`1`**, to indicate that the upcoming bytes represent a key, the second flag byte is left at a default of **`0`** and is ignored. The next bytes will contain the key in encoded form, more precisely the key transformed by following the X.509 standard for public key formats.

In the second case the user will write his/her message text and then choose which security options, if available, to use; if AES is selected a password must also be chosen. Once the message is finished, the text first goes through the aforementioned full compression routine (preprocessor and compressor), where the initial string is transformed to a sequence of bits representing the compressed content. The resulting bytes go then through a series of transformations, depending on which security options were chosen, in a fixed order; after each stage the bytes are transformed depending on the security option, or remain the same if that option was not selected. First a DSA signature is added, then RSA encryption is calculated, then AES is applied with the supplied user’s password.

The SC-SMS flags are then created as follows: the marker symbol is placed as usual, and the eighth bit is set to **`0`** to indicate that a SC-SMS with text is encoded. The three bits representing the possible application of security measures are set to their values, depending on the choice of the user; the final five bits are set to represent the dictionary index used to compress the text.

The resulting SC-SMS (sequence of bytes) is then transformed to a 7-bit encoded string of characters following the GSM 03.38 standard, and sent as one (or more) SMS over the default network; this is needed in order to correctly pass the data to the *Android* methods for SMS sending. The charset definition contains a special symbol (the *escape symbol*) and the carriage return symbol which cannot be represented as normal characters in a string (at least not to be handled correctly by the *Android* API). Due to that the program has three special cases which use a custom special symbol “@”: “@1” to represent “@”,“@2” to represent the carriage return, and “@3” to represent the escape symbol. The whole described procedure of SC-SMS generation for the different types of data is resumed and schematized in Figure 9.

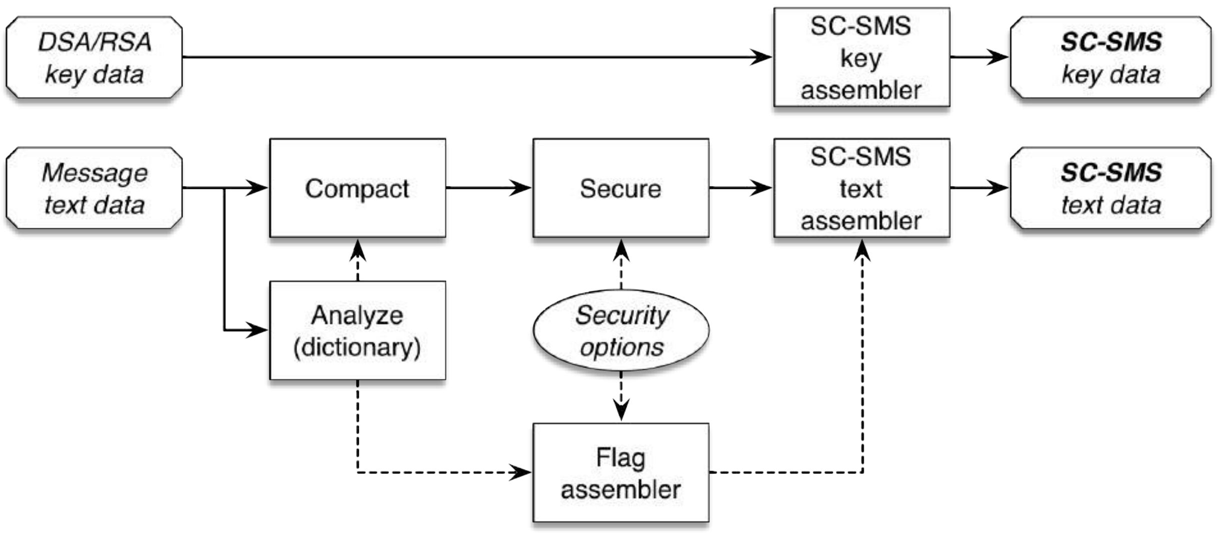


Figure 9. SC-SMS generation scheme

The retrieval of data from an incoming SC-SMS is basically similar to the previous procedure, but in reverse, and works as follows: for every incoming SMS we look at the first character, if it matches the special marker character, the program tries to read and decode the data as an SC-SMS message. Firstly the GSM 03.38 charset is used to go back from a string of characters to a sequence of bytes, this gives back the exact same sequence of bits which composed the original SC-SMS message. The application then looks at the eighth key bit, if it is **1** then it knows the bytes after the flags represent a key; the other flag byte is ignored and the data is read and interpreted as a key object, which also contains the details about the type of algorithm of the key. The bytes are then correctly classified and interpreted as a RSA or DSA public key and saved for the specific incoming phone number after the receiving user’s decision. If the eighth bit is instead **0** the program knows a text SC-SMS has arrived and reads the next flags to learn how to proceed. The first thing to do is to revert the security measures taken (if any): the next three bits coming from the second flag byte are read, and the relative security decryptions/checks applied in reverse order in respect to the generation phase; if AES was used, a password is requested to the user.

Once the previous procedure is over, we have a sequence of bits representing the compressed text string, the application will then read the final five bits and initialize PPM for decompression by passing the specified dictionary, the text is then decompressed and a string is produced. At this point the last step is to run the text preprocessor (in this case post-processor) in decoding mode, in order to get the original message text back (with possible syntax space corrections); the result is the text string originally input by the sender.

6.3 Multipart messages

Multipart (concatenated) SMS is a method already supported by most mobile phones and network providers, which enables the user to send a message containing more than 160 characters split in multiple SMS of 153 characters each (space is lost due to headers). While using the already existing feature might have been easier, a custom, simpler multipart support for SC-SMS has been implemented, which reduces the data overhead and allows for more useful data to fit on each message sent. The conversion of a SC-SMS to a multipart SC-SMS happens as last step, if needed, just before the actual sending of data through the network.

The result from all the previous operations is a string of characters following the standard GSM charset, meaning a maximum of 160 characters per message, or (160 times 7 bits) 140 bytes. The custom multipart building procedure works as follows. A custom marker symbol is chosen, for simplicity the same symbol used to mark a SC-SMS (“¤”) is used. If the string is 160 characters (140 bytes) or shorter and the last symbol is not the marker, we simply take everything and send the string, as shown in Figure 10. If it is not the case, the first 159 characters of the string are taken, the marker symbol is appended at the end of the taken string and the whole string is sent; Figures 11 and 12 show the two cases with exactly 160 characters. If there are characters left we then repeat the procedure with the remaining characters of the string until everything has been sent (Figure 13). The receiver will then react in the following way when receiving a new SC-SMS: if the message is 160 characters long and ends with a marker symbol, he will store the string (firstly removing the marker), if another part is already stored, the new part is appended; if the message is shorter than 160 characters or doesn't end with a marker symbol, the part is appended to any existing part if needed, composing the final full message.

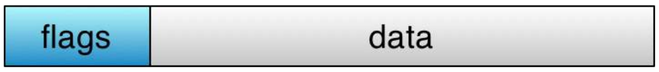


Figure 10. Multipart SC-SMS case: less than 140 bytes

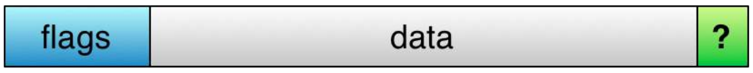


Figure 11. Multipart SC-SMS case: 140 bytes, last character not marker

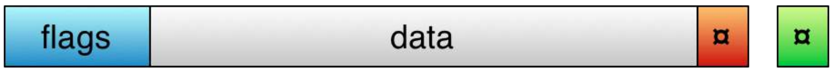


Figure 12. Multipart SC-SMS case: 140 bytes, last character is marker

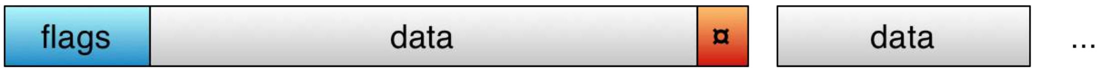


Figure 13. Multipart SC-SMS case: more than 140 bytes

The four different base cases for multipart SC-SMS splitting are shown in Figures 10, 11, 12, and 13; each of them shows how data is divided across messages depending on the length and symbols contained in the original block of bytes. Longer concatenations can be obtained by combining the last case with the remaining data by repeating the procedure (it should be noted that flags are only needed in the first message part).

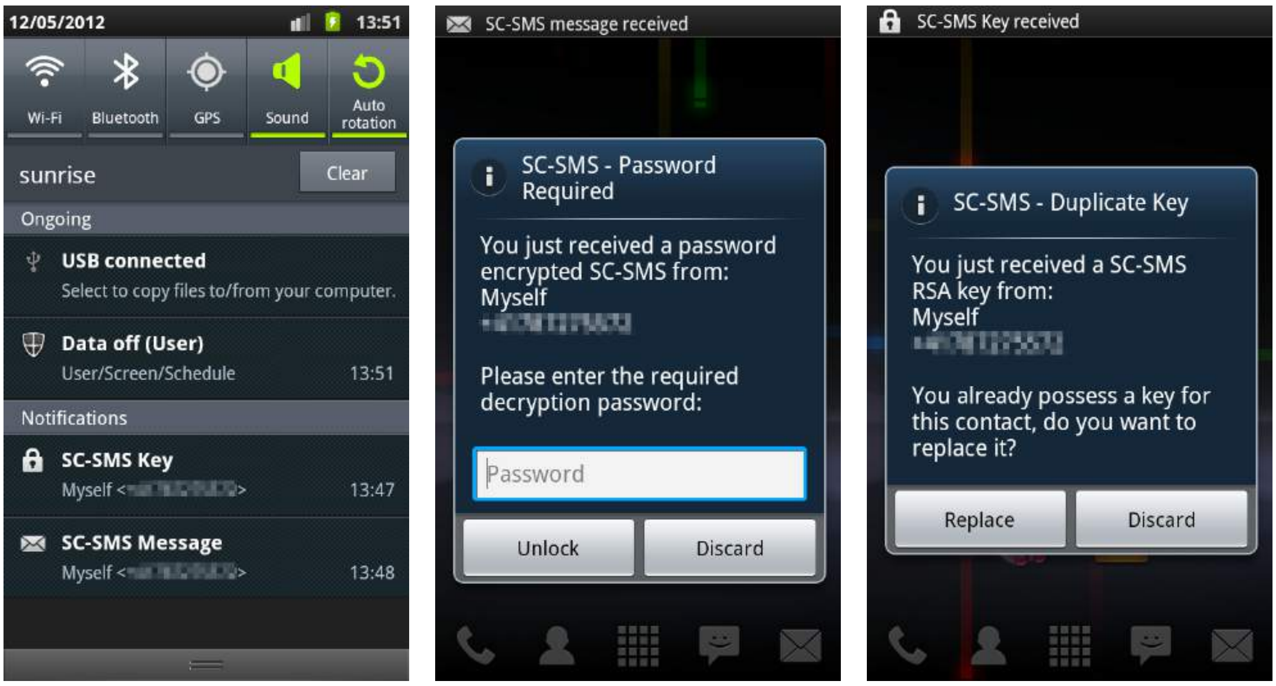
The advantage of this scheme is a very low overhead for multipart messages, with only one character used to mark the multipart nature of the message, not even used in the final part of the message. The procedure also results in single part SC-SMS being correctly handled without waste of space.

7 Android integration

The developed application has various parts well integrated with *Android* OS, in order to provide the user the best possible experience combining SC-SMS and normal messaging. One part of the program is more “abstract”: it deals with compression, security and similar features, handling strings and bytes with no direct user interaction, but instead providing simple functions to access. The program part directly related to SC-SMS and the user within *Android* can be divided in two sections: receiver and sender, with each one having different types of relations with the operating system and the user, interacting directly with program windows or dialogs.

7.1 Receiver component

The receiver part is responsible for intercepting and reacting on incoming SC-SMS, it must differentiate them from normal messages, combine multiple parts, and interact with the user only when needed. It is always silently waiting in background, activating whenever any kind of new SMS arrives; if the incoming message is a normal one, it is simply passed to the OS’s default system for handling new incoming SMS, otherwise, a new thread is spawned in order to process the message and the background receiver goes back to wait for other new messages. Example screenshots of this component at work can be seen in Figure 14; showing from left to right: different alerts in the notification history, a dialog requesting a password to decrypt an incoming SC-SMS, and a dialog requesting the user’s choice regarding a newly received public key.


12/05/2012 13:51

sunrise

Ongoing

USB connected
Select to copy files to/from your computer.

Data off (User)
User/Screen/Schedule 13:51

Notifications

SC-SMS Key
Myself <[redacted]> 13:47

SC-SMS Message
Myself <[redacted]> 13:48

SC-SMS message received

SC-SMS - Password Required

You just received a password encrypted SC-SMS from:
Myself
[redacted]

Please enter the required decryption password:

SC-SMS Key received

SC-SMS - Duplicate Key

You just received a SC-SMS RSA key from:
Myself
[redacted]

You already possess a key for this contact, do you want to replace it?

Figure 14. Notifications and alerts of the Android receiver component

When a new SC-SMS arrives, initially a notification is shown in any case to the user to let him/her know that there’s a new message, the notification is very similar to the default SMS ones: it is stored in the status bar together with informations about the SC-SMS which just arrived, letting the user rapidly see and select any new message for viewing. If an incoming SC-SMS contains data which requires user interaction, dialogs depending on the situation are shown, prompting the user for the action to take; examples for this case are new DSA certificates, password-encrypted messages and so on. The action selected is applied to the relative data and the user can then proceed with his/her normal phone utilization.

The processed SC-SMS, meaning decrypted and decompressed, is composed at this stage by a simple string representing the text of the message, plus a possible string header indicating the security features. The thread will store the message together with the sender’s informations in the default inbox of the messaging application, so that the user can easily access it like any other received message.

7.2 Sender component

The sender part deals with the direct wanted user interaction with his/her contacts, more specifically with the entire SC-SMS composing action, real time interaction with compression, security options and management of keys. The program is started intentionally by the user (it is not always active in background) and provides a user interface divided in three parts to give access to different features. The three tabs are responsible respectively for message composition, security options of the message, and key management with the contacts. The application also implements a background thread that deals with the compression of the message, started and ran automatically when needed by the user to provide real time feedback and fluid experience.

When a message is sent, the application takes care of putting a copy of the message in the OS's default outbox folder, completing the standard messaging experience, providing SC-SMS conversations integrated with all the other existing and future messages of the user.

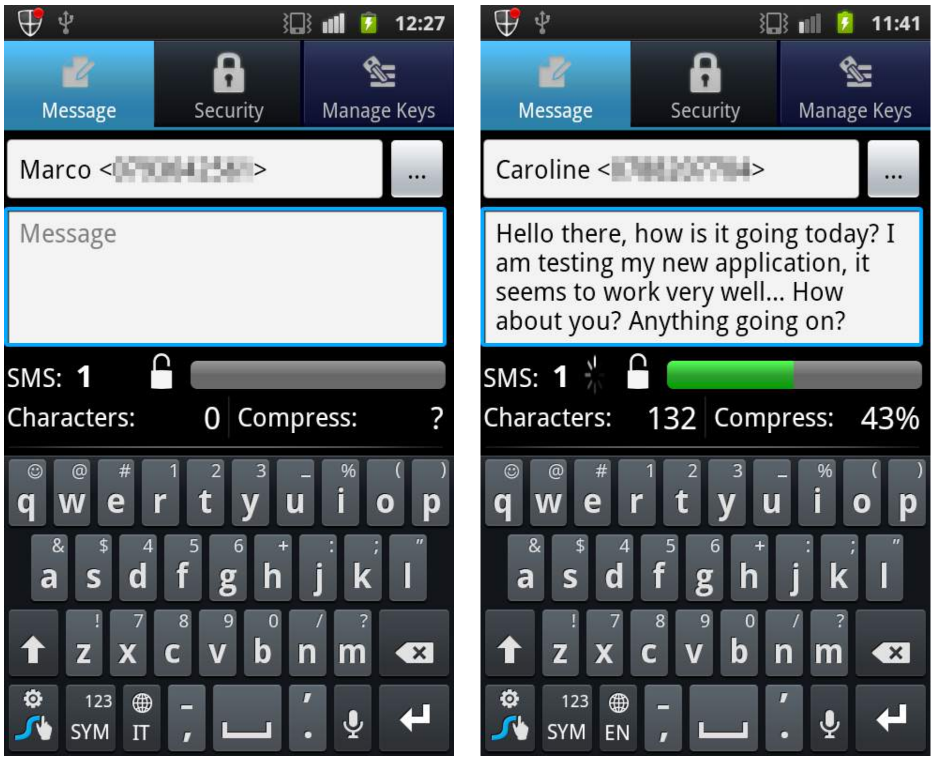
7.2.1 Message tab

In the "Message" tab interface the user can select the target phone contact or number, manually (with suggestions) or by selecting from the personal contacts; text can then be written. Whilst the user inputs text, the background thread continuously analyzes and performs compression on it to provide real-time feedback about compression and message size, while a spinning icon is displayed to indicate this activity.

Images of the tab in an example usage situation can be found in Figure 15: on the left is the view with an empty message; on the right the tab situation with a partial message, while the user is busy typing text.

A counter of currently used number of SMS is shown; the message size is displayed as a simple bar to give a quick visual feedback about the used and remaining part of the current SMS, this is also done because providing an exact count of remaining characters is impossible due to compression. The bar has at its side a lock icon which changes to indicate whether security options are active, if it is the case the bar also changes color, this indicates that there are overheads due to security and that only an estimation of the size is possible.

Under these described features there are two additional informations displayed: the effective characters written and the current compression percentage. Finally a button is given at the bottom for the user to select when he/she is ready to send the SC-SMS.


Marco <[redacted]>

Message

SMS: 1

Characters: 0 Compress: ?

Caroline <[redacted]>

Hello there, how is it going today? I am testing my new application, it seems to work very well... How about you? Anything going on?

SMS: 1

Characters: 132 Compress: 43%

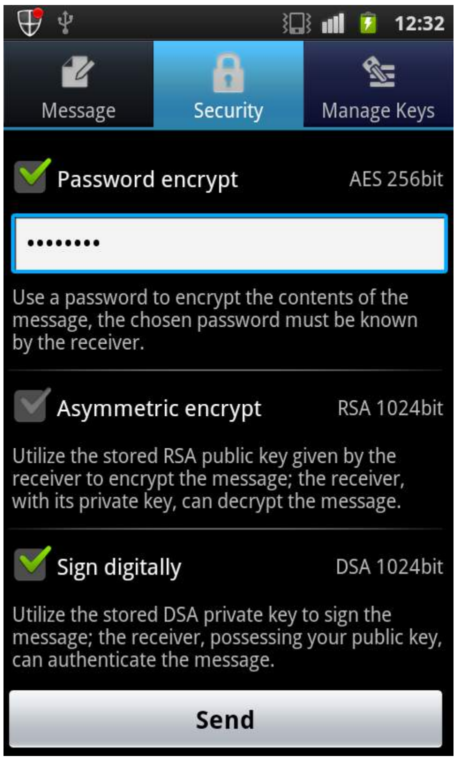
Figure 15. The "Message" tab in two different moments of message composition

7.2.2 Security tab

Security options for the message are provided in the “Security” tab, shown in Figure 16: simply activated as checkboxes, the three types of protections can be enabled easily by the user and applied to the message when it is sent. The password protection option reveals an input field to write the password, and a short description is provided for all the security features.

The different security options are directly linked and dependent on the selected message recipient: while the password encryption option is always active, the two other features are enabled or disabled depending on the relative security key presence or absence.

All the security features can be enabled/disabled at the same time, any combination of preferences is possible; the current status of the options is updated and reflected in the “Message” tab, where it is used to make an estimate (if needed) of the current size of the data occupied in the message and displayed in the security icon (security options on/off) and the message size bar.



✓ Password encrypt AES 256bit

••••••••

Use a password to encrypt the contents of the message, the chosen password must be known by the receiver.

✓ Asymmetric encrypt RSA 1024bit

Utilize the stored RSA public key given by the receiver to encrypt the message; the receiver, with its private key, can decrypt the message.

✓ Sign digitally DSA 1024bit

Utilize the stored DSA private key to sign the message; the receiver, possessing your public key, can authenticate the message.

Figure 16. The “Security” tab options

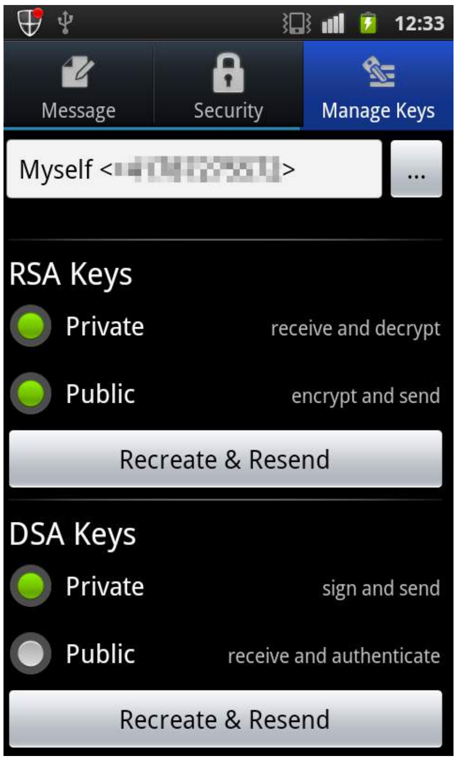
7.2.3 Manage keys tab

The private and public key management used by RSA and DSA is provided in the “Manage Keys” tab, shown in Figure 17. Here the user can select any contact (or also a custom number) and see the current status of availability of keys for that specific person; by default the recipient of the current message is selected.

The interface shows a quick overview of the existing keys and respective features available, so that the user can quickly understand what it is possible to do. Two different buttons are provided for each of the key types, to generate (replacing the existing keys if necessary) a new key pair and transferring it via SC-SMS to the target contact.

The recipient contact will then see an alert by the receiver component, meaning that a new key has been received, he or she will then have the choice to accept it and replace the existing keys if needed.

The key status obviously directly updates the available security options for sending a message to a contact, enabling or disabling the different features depending on the availability of the keys related to the different algorithms.



Myself <[redacted]>

RSA Keys

● Private receive and decrypt

● Public encrypt and send

DSA Keys

● Private sign and send

○ Public receive and authenticate

Figure 17. The “Manage Keys” tab view

8 Program structure

8.1 Compression component

The compression component of the application is responsible for the transformation of a string of text to and from a sequence of bytes representing the text lossless compressed; it is separated in different sections, each with a specific role, described in the following paragraphs. A diagram of this component is shown in Figure 18.

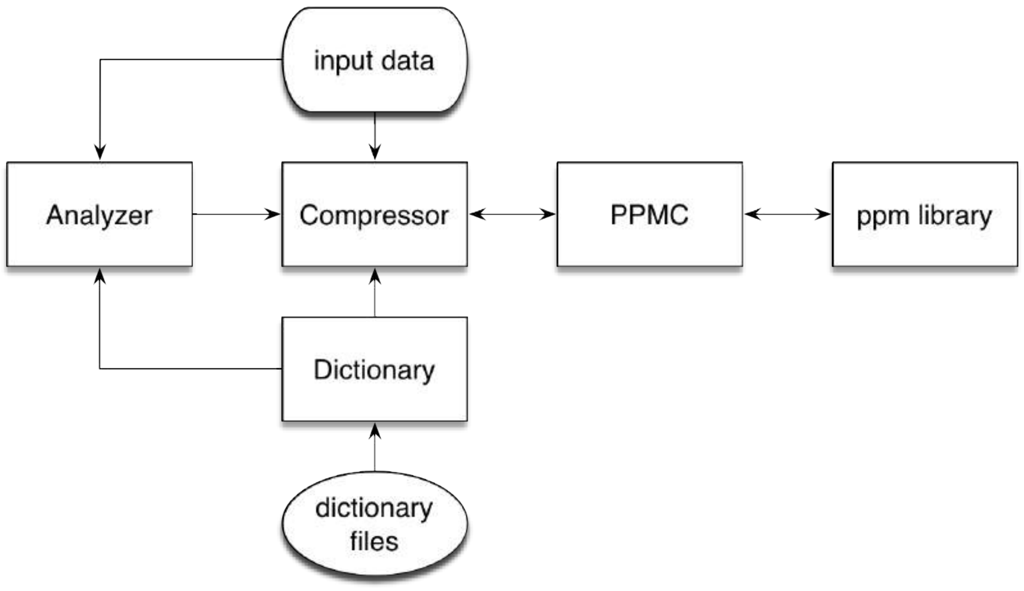


Figure 18. Compression component scheme

- **`Dictionary.java`**
  This object is responsible for the interaction with the filesystem, in regard to compression; it deals with the initial loading in memory of all the existing dictionary files. The loaded dictionary entries are put in a single structure shared by all the other compression components, which contains all the loaded strings divided per dictionary. In the case of *Android*, the dictionary files are included directly in the application's package.

- **`Analyzer.java`**
  Its purpose is to analyze the text: it takes as input a string and tries to determine with a certain reliability which is the dictionary language better representing the given text. Its output is used to initialize the other components which will compress the text differently depending on the given dictionary.

- **`Preprocessor.java`**
  The class responsible for the preprocessing part of compression is independent from any other part; it simply takes as input a string and, depending on the parameters, performs the compression or decompression (pre-processing or post-processing), increasing the final compression ratio.

- **`Compressor.java`** and **`PPMC.java`**
  These objects combined act as link and callers of the used PPM compression library; the compressor object is responsible for parameter setting of the compression and the training of the model. It also deals with the copy and retrieval of the state of the model for each consequent compression.

- **`ppm`** (package)
  This is the library used for PPMC compression, it contains the model state and is responsible for the compression and all the underlying model modifications. The files are a limited selection of the original library, with small modifications to enable model copy and retrieval for different compressions.

8.2 Security component

The security component deals with the transformations of sequences of bytes related to encryption, decryption, signatures and verifications. Like the compression component, it is also divided in four clear parts responsible for different security features, explained as follows. A scheme of this component is shown in Figure 19.

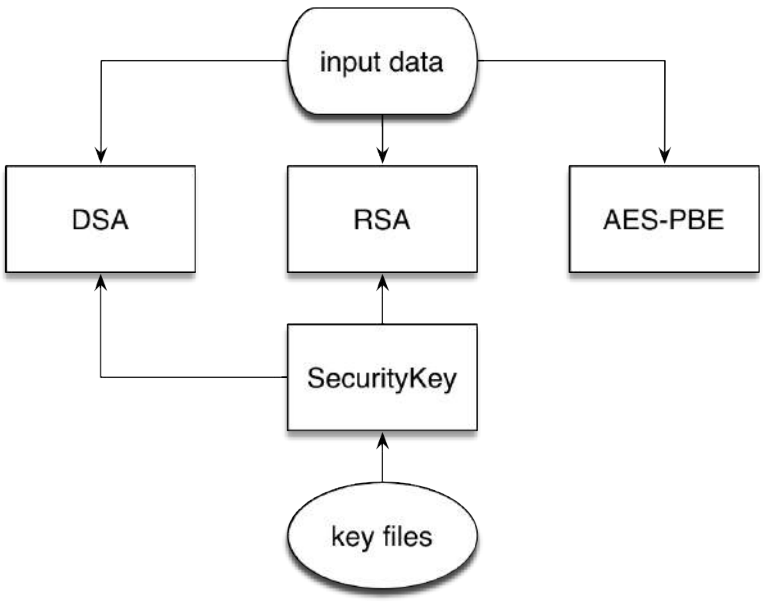


Figure 19. Security component scheme

- **`AESPBE.java`**
  This object is responsible for symmetric encryption and decryption of data. It works by taking as input a sequence of bytes representing data and a password string; the results of the functions are the encrypted or decrypted data, using the AES-PBE algorithms specified in the previous chapters. The result of the functions is *null* if the password is wrong or an error occurred. It should be pointed out that this object is independent from the rest of the objects.

- **`SecurityKey.java`**
  The purpose of this part is to interact with the filesystem, similarly to the compression's dictionary component. In this case the object has the goal of generating, storing and retrieving public and private asymmetric keys used by RSA and DSA. Key storage standards are used for saving data, in the case of *Android* the keys are stored in the application's private memory, with names depending on the relative phone number.

- **`DSA.java`**
  This object deals with digital signing and verification using the DSA standard, it interacts with the *SecurityKey* object for key management and is simply composed of two functions for the two purposes.

- **`RSA.java`**
  This object's structure and function is very similar to the previous one, it however deals with encryption and decryption using the RSA standard, it also interacts with the *SecurityKey* object in order to manage private and public keys.

8.3 Data component

The data component shown in Figure 20 is responsible for all the transformations needed to convert a byte array of data from and to a SC-SMS in string form. The byte data is used by the security and compression features, representing the text of the message. The string data is the GSM compatible form of the SC-SMS data format, to be sent and received on the mobile network. This component can be divided in three parts as follows.

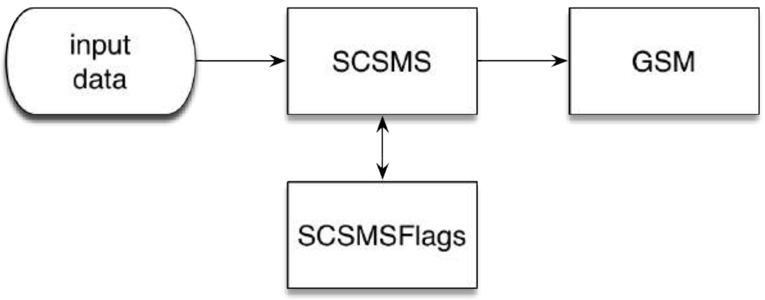


Figure 20. Data component scheme

- **`SCSMSFlags.java`**
  This part deals with the encoding, creation and interpretation of SC-SMS flags, it has functions to get and set options from and to a set of bytes, found at the beginning of every SC-SMS message.
- **`SCSMS.java`**
  This object is responsible for the generation of SC-SMS and retrieval of data from them, it interacts with the part responsible for flag management described before. When used for text (instead of keys), the component is responsible for the sequential calls and transformations through the different security algorithms.
- **`GSM.java`**
  The separate GSM class represents the object that converts byte data to/from strings in GSM format, it simply offers two functions for the two different conversions.

8.4 Android component

The Android component, with scheme in Figure 21, contains all the different parts necessary for the application to reside and work on the *Android* OS, including various UI and multithreading features. It is also responsible of combining all the previously described components to build the complete working application.

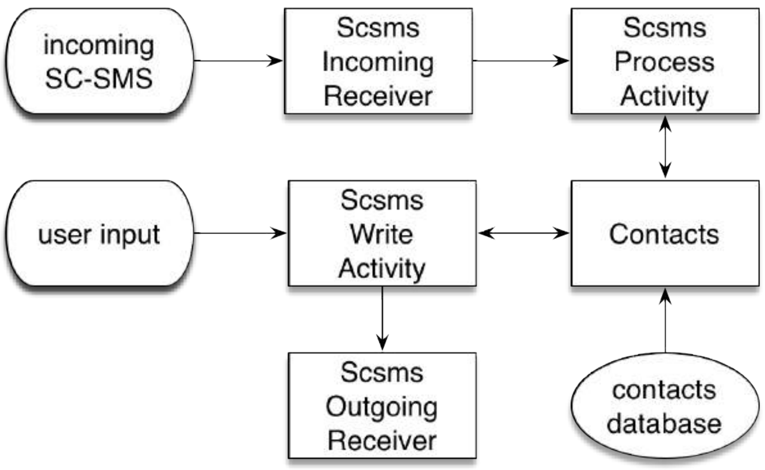


Figure 21. Android component scheme

- **`Contacts.java`**
  This independent static class is used by other objects in the Android component to get informations about contacts, more specifically it is used to get a contact name (if known) from a given phone number.

- **`ScsmsIncomingReceiver.java`**
  This object is an *Android BroadcastReceiver*, that is a special entity which is registered to the OS and gets activated and called whenever certain events happen, in this case the object reacts to new incoming SMS messages. It takes care of filtering between normal SMS messages and special SC-SMS messages: in the first case nothing is done and the message is passed on wherever is needed by the system; in the second, the message is intercepted and special processing is done by starting a new activity. This object is also responsible for the receiving, managing and appending multipart SC-SMS messages.

- **`ScsmsProcessActivity.java`**
  This activity gets started with the incoming SC-SMS message data as parameter by the previously described receiver. It performs all the needed functions to decode the data, save the message in the system and display feedback to the user.
  The function begins by displaying a standard notification in the *Android* bar, depending on the message contents (text or public key), and displaying the contact which sent the message with the time of arrival. The activity then proceeds to decode the flags of the SC-SMS to know how to deal with the data; if no user input is needed and no error is detected, the message gets decrypted, decompressed and saved in the default inbox. In the case of errors or input needed from the user (for instance duplicate keys, password encrypted SC-SMS, ...), different dialogs are shown and the answer registered to continue with the execution; once the SC-SMS is saved or discarded, the activity terminates.

- **`ScsmsOutgoingReceiver.java`**
  This simple class is another *BroadcastReceiver*, its purpose is to react to the status reports (given by the network) of sent SC-SMS messages and display feedback to the user; it is used by the SC-SMS write activity to show if a message has been successfully sent or not.

- **`ScsmsWriteActivity.java`**
  This object is the main activity of the application and incorporates other smaller objects like threads. Its main purpose is to handle everything related to the program started by the user, which is used to send a new text SC-SMS and manage security keys. It is responsible for calling and communicating with all the previously described security and compression components.
  The class contains all the functions used by the interface for the interaction with the user. It also defines and keeps track of the running background threads, specifically two different types: real time compression and security, and final message creation and sending. The threads are created and used so that the activity of the user will not be interrupted during the operations, and interact with the separate UI thread to give feedback on the status. Besides private functions used for generating SMS data and sending the messages on the network, the class also has a method to call and receive data from the default contact list of the *Android* OS, feature that can be utilized by the user when selecting the recipient for the text or key.
  The majority of this component deals with listeners for user input: security options, recipient selection, text typing, and so on; all lead to the respective functions being called or variables being changed. Initially the activity starts by loading the dictionaries into memory, once this is setup the actions of the application entirely depend on user interaction: it will keep running sending messages when needed, terminating only when the user wants to exit.

9 Possible developments

9.1 Elliptic curve cryptography

Elliptic Curve Cryptography (abbreviated ECC) was originally proposed in 1985 and 1987 by V. S. Miller[17] and N. Koblitz[18] as a new method for asymmetric key encryption/decryption and signatures. As explained in the previous sections, the utilized RSA algorithm is widely used, known and certified; it however doesn't mean it is the best, or that it doesn't have disadvantages. Recently much research has been (and is being) done about the argument; ECC seems to bring many advantages over the currently used algorithms for encryption and signature generation.

Especially in the field of this project, SMS transmission, key sizes and block sizes used are very important, as they determine respectively the number of messages needed to send keys or text; Elliptic Curve Cryptography reportedly offers the same security as RSA or DSA at much lower key and block sizes.

Since the security component is independent from the rest of the program, switching the algorithm used for a certain cryptographic function is very easy, implementing ECC would certainly bring benefits to the user, who would need to utilize less messages, and/or be protected by a better security mechanism. If and when this innovative cryptography system will become standardized and secure with a certain reliability, the modification or addition to the existing application can be done without many difficulties.

9.2 Additional key exchange channels

In the current state of the application, keys for RSA and DSA are exchanged via SMS, the same channel used by SC-SMS text messages, which has the drawback of being not secure and costly, besides other negative features. Being public keys simple short byte data, it is easy to imagine other ways to easily transfer and communicate to a target user the wanted key, especially if thinking about physically close communication.

If we consider two users wanting to exchange keys meeting somewhere, there are two currently widely available alternative network technologies which could be used for the purpose. The first one is *bluetooth*, available on mobile phones since many years, it is an ideal candidate due to its low communication range and easy to setup connection; another possibility would be to use *Wi-Fi*, more precisely *ad hoc* connection, to exchange data directly between to connected devices.

Both the previously mentioned technologies could enhance user experience by allowing users to easily exchange keys without having to rely on the GSM network.

9.3 Separate encrypted storage of messages and keys

The developed application has been built to have a very simple and intuitive integration with *Android*'s default messaging system: it doesn't have its own inbox or outbox, but rather integrates the messages with the relative locations already present in the OS. This is in line with the proposed security features, which has been ideated to purposefully protect the data only during transmission, considering the phone itself secure.

It is however understandable, and maybe worth further work, to enable the user to setup private storage for the data used by the program, message and keys alike. The envisioned situation would be an application with its own private inbox and outbox folders, containing all the needed messages (maybe setup by filter by the user) separated by the normal SMS traffic. The stored data would be encrypted by a user set master password, which would deal with encryption and decryption of all data stored on the memory of the mobile phone.

This expansion of the program, while requiring probably more work than the two previously mentioned solutions, can benefit from the already present component of security to handle encryption of data; it would bring benefits to the user by providing additional security with complete encryption of data in all situations.

10 Conclusion

A new application for *Android* called SC-SMS has been developed and presented, with the aim of improving security features and overcoming length limitations of the current mobile SMS service. For the security problems, the program offers three different algorithms for different purposes: AES password based symmetric encryption, RSA asymmetric encryption via private/public keys, DSA digital signing and verification. For the size limitations, a combination of a newly developed preprocessor and modified compression library utilizing the PPMC model has been used, to reduce the size of the sent text.

The security features are applied to data during transmission, ensuring protection of messages on the network, but assuming safe sender and receiver mobile phones. All the utilized algorithms are used following the current standards of security, ensuring data protection and authentication for at least several years. The utilization of the security features can be combined together in all possible ways and doesn't require any particular knowledge by the user. The compression applied to the text of messages directly supports the main 32 different languages spoken around the world, composed by Latin, Cyrillic, Slavic and Hebraic characters; the compression can be also applied to different languages or alphabets but would lose efficiency. The average compression ratio is around 1:2 starting from about 80 characters of given text data; the compression quality improves with quantity of data and usage of common words. Thanks to compression it can be said that in most cases the new text contents that can be sent in a single message are twice as long as before.

Android OS has been used as the platform for the deployment of the application, any phone compatible with the system starting from version 2.2 (*"Froyo"*) is able to run the application. The developed full application running on the mobile phone has been built to provide features of message receiving and composing as easy to use as possible for the user. A tight integration with the operating system and its messaging components has been made, in order to give an experience as close as possible to the default communication applications.

SMS messaging is and will be for many years to come one of the main forms of communications around the world; although its systems and infrastructures have limitations, it is however possible to overcome most of these problems. The presented application can be seen as a solution (even if not complete) to what probably are the major drawbacks of the current communication service: security and text length limits. There are still vast areas of improvement, research to be done and applications to be developed; this project wants to be a practical example on how to utilize the already existing solutions to provide more advanced features of communication, thanks to smart software.

A Terminology and abbreviations

- **SMS**
  Short **M**essage **S**ervice: the service provided on the mobile phone network for transfer of short amounts of text data (140 bytes). The abbreviation is also used to indicate the single messages sent on the network.

- **SC-SMS**
  Secure and **C**ompact **SMS**: the name of the presented application, also name of the utilized messages, incorporating compression and security features.

- **UI**
  User **I**nterface: the layer of software responsible for interaction between the user and the program, it deals with reaction to the input and output of the application status and results.

- **GSM**
  Global **S**ystem for **M**obile Communications: a set of standards defined by the European Telecommunications Standards Institute which deal with mobile technologies, including voice and SMS communication network.

- **OS**
  Operating **S**ystem: the software responsible for the link between hardware and other software, for the basic operations and running other applications; in the case of the developed project the operating system is *Android*.

- **PPM**
  Prediction by **P**artial **M**atching: the adaptive statistical compression algorithm used in the application, uses previously seen symbols to predict the upcoming ones; the specific variant used is PPMC.
  See Section 4.5.

- **DSA**
  Digital **S**ignature **A**lgorithm: a current standard for digital signing and verification of data, utilizing a private/public key system to ensure authenticity.
  See Section 5.2.

- **RSA**
  R. **R**ivest, A. **S**hamir, L. **A**dleman: the name of the developers and name of the algorithm now widely used standard for asymmetric cryptography. It utilizes a private/public key system for data encryption/decryption.
  See Section 5.3.

- **AES**
  Advanced **E**ncryption **S**tandard: the current accepted widely accepted standard for symmetric encryption of data, the used version with 256-bit keys is currently considered very secure for many years to come.
  See Section 5.4.

- **PBE**
  Password **B**ased **E**ncryption: method to derive a more secure key to be used by an encryption or decryption algorithm, utilizing recursive mathematical operations on a starting password. Can be appended to the algorithm name to indicate its usage (ex. AES-PBE).
  See Section 5.4.

List of Figures

1 *SMSzipper* screenshot 3

2 *CryptoSMS* screenshot 3

3 The final project plan 6

4 General scheme of the application's work 7

5 General scheme of text data compression 8

6 PPM compression model scheme (*from [6]*) 10

7 Data security implementation scheme 15

8 Structure of the SC-SMS flag bits and data 17

9 SC-SMS generation scheme 18

10 Multipart SC-SMS case: less than 140 bytes 19

11 Multipart SC-SMS case: 140 bytes, last character not marker 19

12 Multipart SC-SMS case: 140 bytes, last character is marker 19

13 Multipart SC-SMS case: more than 140 bytes 19

14 Notifications and alerts of the Android receiver component 20

15 The “Message” tab in two different moments of message composition 21

16 The “Security” tab options 22

17 The “Manage Keys” tab view 22

18 Compression component scheme 23

19 Security component scheme 24

20 Data component scheme 25

21 Android component scheme 25

List of Charts

1 Average compression performance in relation to context length 12

2 Average compression improvement from no dictionary in relation to dictionary size 13

3 Average compression performance in relation to preprocessor utilization 13

4 Compression performance of the final algorithm 14

List of Tables

1 Example dictionary top words for selected languages 8

References

[1] "The world in 2010," tech. rep., International Telecommunication Union, 2011.

[2] M. Toorani and A. A. Beheshti, "Solutions to the GSM security weaknesses," in *Proceedings of the 2008 The Second International Conference on Next Generation Mobile Applications, Services, and Technologies*, 2008.

[3] J. G. Cleary and I. H. Witten, "Data compression using adaptive coding and partial string matching," *IEEE Transactions on Communications*, vol. 32, no. 4, 1984.

[4] T. Bell, I. H. Witten, and J. G. Cleary, "Modeling for text compression," *ACM Computing Surveys*, vol. 21, no. 4, 1989.

[5] A. Moffat, "Implementing the PPM data compression scheme," *IEEE Transactions on Communications*, vol. 38, no. 11, 1990.

[6] S. Rein, C. Gühmann, and F. Fitzek, "Compression of short text on embedded systems," *Journal of Computers*, vol. 1, no. 6, 2006.

[7] D. Salomon, *Data Compression: The Complete Reference*. Springer, 2007.

[8] P. Traynor, W. Enck, P. McDaniel, and T. L. Porta, "Exploiting open functionality in SMS-capable cellular networks," *Journal of Computer Security*, vol. 16, 2008.

[9] "Digital Signature Standard (DSS)," tech. rep., National Institute of Standards and Technology, 2009.

[10] R. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, 1978.

[11] "PKCS #1 v2.1: RSA cryptography standard," tech. rep., RSA Laboratories, 2002.

[12] J. Daemen and V. Rijmen, "AES proposal: Rijndael, Version 2." Submission for the new AES Standard, 1999.

[13] "PKCS #5 v2.1: Password-based cryptography standard," tech. rep., RSA Laboratories, 2006.

[14] M. Bellare, R. Canetti, and H. Krawczyk, "Keying hash functions for message authentication." 1996.

[15] "Secure Hash Standard (SHS)," tech. rep., National Institute of Standards and Technology, 2008.

[16] "Alphabets and language-specific information (GSM 03.38)," tech. rep., European Telecommunications Standards Institute, 1996.

[17] V. S. Miller, "Use of elliptic curves in cryptography," *CRYPTO*, vol. 85, 1985.

[18] N. Koblitz, "Elliptic Curve Cryptosystems," *Mathematics of Computation*, vol. 48, no. 177, 1987.