Secure SMS messaging using Quasigroup encryption and Java SMS API

Marko Hassinen[1]⋆ and Smile Markovski[2]

[1] University of Kuopio, Department of Computer Science, P.O.Box 1627, FIN-70211 Kuopio, Finland
Marko.Hassinen@uku.fi,
[2] Faculty of Sciences, Institute of Informatics, p.f.162 Skopje, Republic of Macedonia
smile@ii.edu.mk

Abstract. SMS (Short Message Service) is a widely used service for brief communication. Occasionally the data sent using SMS services is confidential in nature and is desired not to be disclosed to a third party. In this paper an application for sending encrypted SMS messages using cryptographic methods based on theory of quasigroups is proposed. The encryption algorithm is characterized by a secret key. The application is developed using programming language Java and the J2ME environment available in several mobile devices on the market today.

1 Introduction

SMS messages are sometimes used for the interchange of confidential data such as social security number, bank account number, password etc. A typing error in selecting a number when sending such a message can have severe consequences if the message is readable to any receiver. Most mobile operators encrypt all mobile communication data, including SMS messages but sometimes this is not the case, and even when encrypted, the data is readable for the operator. Among others these needs give rise for the need to develop additional encryption for SMS messages, so that only accredited parties are able to engage communication.

Our approach to this problem is to develop an application that can be used in mobile devices to encrypt messages that are about to be sent. Naturally decryption for encrypted messages is also provided. The encryption and decryption are characterised by a secret key that all legal parties have to posses.

Several mobile device manufacturers have adopted Java as their platform offered for software developers. To certain extent Java applications are portable between devices of differend vendors. Some mobile device manufacturers provide an application programming interface (API) for SMS services, which can be used for our purposes. These facts makes Java a natural choice for our application.

In addition to cryptographic strength, things to consider when developing this type of an application for mobile devices are limitations in memory and processing capacity. Quasigroups are well suited for encryption of this type of

⋆ Supported by a grant from the University of Kuopio data. The cryptographic strength of quasigroup based encryption has been examined in earlier papers [1], [2] to some extent. In Section 2 we give definitions of encryption and decryption methods. The amount of memory needed for these operations as well as the amount of processing power is examined in Sections 3 and 4. Memory allocation can be examined both on algorithm level by calculating the memory needed for variables used and by using memory allocation logging when running the application on an emulator.

The suitability of current Java platforms for this type of an application is examined in Section 5 focusing on two major vendors, Nokia and Siemens. The application for sending encrypted messages as well as receiving and decrypting them is presented in Section 6.

2 Definitions

2.1 Quasigroups

A *groupoid* is a finite set Q that is closed with respect to an operator $*$, i.e., $a * b \in Q$ for all $a, b \in Q$. A groupoid is a *quasigroup*, if it has unique left and right inverses, i.e., for any $u, v \in Q$ there exists unique x and y such that $x*u = v$ and $u * y = v$.

This means that all operations are invertible and have unique solutions, which implies their usability as cryptographic substitution operations. With this in mind we can define inverse operations for $*$, call them \(left inverse) and /(right inverse) . The operator \ (resp. /) defines a new quasigroup $(Q, \backslash)$ (resp. $(Q, /)$) and for algebra $(Q, \backslash, *)$

$$x * (x \backslash y) = y = x \backslash (x * y) \tag{1}$$

A quasigroup can be characterised with a structure called Latin square. A *Latin square* is an $n * n$ matrix where each row and column is a permutation of elements of a set. In our case $|Q| = n$.

Several other operations can be derived from the operation $*$ [2], but for our purposes operations $*$ and \(right inverse) are sufficient.

2.2 Encryption

The encryption primitive e_l on sequence $x_1x_2 \ldots x_n$ is defined as $e_l(x_1x_2 \ldots x_n) = y_1y_2 \ldots y_n$ where

$$\begin{cases} y_1 = l * x_1, \\ y_{i+1} = y_i * x_{i+1} (i = 1, \ldots n-1) \end{cases} \tag{2}$$

The variable l is called the *leader* and in our application it is derived from the secret key (password). Tansformation e is a mapping $A^+ \to A^+$. Elements $x_i, y_i \in A$ are usually not characters, but entities of bits, where the bitlength of the element is defined by the size of the Latin square associated with the quasigroup.

2.3 Decryption

Decryption $d_l : A^+ \to A^+$ is naturally a reverse operation of encryption. It is defined as $d_l(y_1 y_2 \dots y_n) = x_1 x_2 \dots x_n$, where

$$\begin{cases} x_1 = l \backslash y_1, \\ x_{i+1} = y_i \backslash y_{i+1} (i = 1, \dots n-1) \end{cases} \tag{3}$$

In essence, the decryption primitive as well as the encryption primitive are table lookup methods, where the Latin square associated with the quasigroup is used. In encryption the previous encrypted element and the next not yet encrypted element are used to find corresponding value from the table. This value is then used as the new encrypted value.

In decryption the values used for lookup are the previous already decrypted value and the next not yet decrypted value. With these a new decrypted value is found from the latin square.

2.4 Composition of encryption and decryption

One application of the encryption e_l defined earlier usually doesn't provide adequate security. For this reason the encryption method is repeated, using different leaders. The encryption

$$E_L \;=\; e_{l_1} \circ e_{l_2} \circ \dots e_{l_k},\; l_i \in L \tag{4}$$

is a composition of consecutive applications of the encryption primitive e_l where leaders $l_1, l_2, \dots, l_k \in L$ are used. Also different quasigroup operations can be used in distinct primitives e_l. The decryption function

$$D_L \;=\; d_{l_k} \circ d_{l_{k-1}} \circ \dots d_{l_1},\; l_i \in L \tag{5}$$

is composed similarly except that the leaders $l_i \in L$ are used in reverse order. The quasigroup operation used in primitive d_l must correspond to the one used in e_l. In earlier papers [1], [2] it has been proved that mappings E and D are bijections and $E_L(D_L(\beta)) \;=\; D_L(E_L(\beta)) \;=\; \beta$.

For encryption purposes, primitives e_l and d_l can be incorporated to produce a more complex encryption method. Then encryption is defined as

$$E_L \;=\; h_{l_1} \circ h_{l_2} \circ \dots h_{l_k},\; l_i \in L, h_{l_i} \in \{e_{l_i}, d_{l_i}\} \tag{6}$$

In this construction, primitive e_l is clearly used for decryption if corresponding primitive d_l has been used in decryption.

Construction of quasigroups and Latin squares is out of scope of this paper. Neverthelss, it is worth mentioning that some Latin squares are not suitable for encryption, for example when the square is symmetric.

3 Description of the application

3.1 Requirements

The requirements that were set on the application were the following:

1. Encrypt an SMS message using quasigroup encryption and send it
2. Receive encrypted messages and decrypt them
3. Operate fast enough to provide acceptable service in an environment with very limited processing power
4. Be compact enough for use in mobile phone memory space

Clearly, for testing these requirements an application development platform is needed. Many mobile phone manufacturers offer some type of support for developing applications on their products, but by far the most standardized and also the most portable solutions are based on Java.

A more restricting requirement is the need to be able to send and to receive SMS messages by the application. There are Java solutions that support this functionality to some extent. Both Siemens and Nokia offer support for these demands in their Java enabled phones.

Obviously, demands on restricted pocessing power as well as memory consumption are tackled with tools of software engineering and are somewhat undependent on the selection of the development environment. With these things in mind the decision to choose Java as the tool for developing this application was made.

3.2 J2ME and MIDlets

J2ME (Java 2 Micro Edition) is a runtime environment specifically designed for devices with very limited resources such as mobile phones or handheld computers. J2ME is comprised of CLDC (Connected Limited Device Configuration) and Mobile Information Device Profile (MIDP). A program developed for J2ME is called a MIDlet. MIDlets use classes defined in APIs of CLDC and MIDP. There is no straight interaction between a MIDlet and the device itself, since MIDlets are run by the Java virtual machine (JVM). The architecture of J2ME is depicted in Figure 1. This architecture limits the functionality a MIDlet can have into those provided by the runtime environment.

3.3 Java CLDC

Basic java language features and language libraries are contained in the CLDC (Connected Limited Device Configuration). It contains application programming interface (API) classes in the very heart of the Java language. These APIs are:

* java.io (Basic input/output functionality)
* java.lang (Basic language constructs)
* java.util (Basic utilities)
* javax.microedition.io (Basic networking)

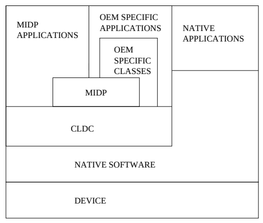


Fig. 1. J2ME architectural structure

3.4 Java MIDP

Built on top of CLDC, MIDP provides classes that give additional function for a specific type of device. MIDP has the following packages:

* javax.microedition.midlet (The base class for MIDlets and the MIDlet lifecycle)
* javax.microedition.lcdui and javax.microedition.lcdui.game (Classes for user interface)
* javax.microedition.rms (Persistent storage)
* java.util.Timer and java.util.TimerTask (Timers)
* javax.microedition.io.HttpConnection (Class for Http connection)
* javax.microedition.media and javax.microedition.media.control (Sound support)
* javax.microedition.pki (Support for X.509 certificates)
* java.lang.IllegalStateException

3.5 Nokia SMS functionality

The functionality of sending and receiving SMS messages is implemented in Nokia phones with the package com.nokia.mid.messaging. This package includes the following classes:

* BinaryMessage
* Message
* MessageConnection
* MessageListener
* TextMessage

This API is based on the Generic Connection Framework (javax.microedition.io package). Sending and receiving messages is implemented by a Connection interface called MessageConnection that inherits javax.microedition.io.Connection. Only a documentation of this API is available by Nokia for the emulator, so if one wants to use it for developing applications and testing them on the emulator, one needs to implement these interfaces.

3.6 Siemens SMS functionality

Also Siemens provides some APIs of their own that are

* com.siemens.mp.game (Images, Graphic objects, Melodies)
* com.siemens.mp.gsm (SMS, Call, PhoneBook)
* com.siemens.mp.io (File handling routines, Connection)
* com.siemens.mp.ui (Image)

Of these com.siemens.mp.gsm and com.siemens.mp.io were used for sending and receiving SMS messages. Also com.siemens.mp.io was used to read contents of the SMS inbox. This makes it possible to decrypt and read messages that have been received when the application was not running.

3.7 Application architecture

Architectural structure of our application is presented in Figure 2.

The application includes the following classes:

* CryptSMS.java (The source code for CryptSMS MIDlet)
* SMSScreen.java (Application main screen)
* WriteSMSScreen.java (Screen for writing a message)
* NUMScreen.java (Screen for typing a phone number for sending a message)
* PASScreen.java (Screen for typing passwords)
* QX.java (Class for encryption and decryption)
* SmsHandler.java (Abstract super class for handling SMS connections)
* WmaSmsHandler.java (WMA implementation for SmsHandler.java)
* NokiaSmsHandler.java (Nokia implementation for SmsHandler.java)
* OutgoingTextMessage.java (Class that describes an outgoing message)
* SiemensSmsHandler.java (Siemens implementation for SmsHandler.java)
* Receiver.java (A class used in SiemensSmsHandler.java for receiving SMS)
* DebugLog.java (A static class used for logging events and errors)
* CommonLog.java (A super class for platform specific log objects)
* SiemensDebugLog.java (A Siemens platform specific log object)
* NokiaDebugLog.java (A Nokia platform specific log object)

The MIDlet resides in CryptSMS.java. It creates and makes visible screens when needed. The main screen of the application is the class SMSScreen, where incoming messages and information about operations are shown. When a new message is initiated, the content of the message is written on screen WriteSMSScreen after which, a number where to send the message is prompted in screen NUMScreen. Finally, a password used to encrypt the message is asked on screen PASScreen.

The core classes depicted in Figure 2 with a dashed line are used in all three environments (WMA, Nokia and Siemens). They have no platformdependent code. The SMS handler is selected by the application runtime depending on the environment. This makes it possible to use the same core classes in all three environments. Specific source code files are included when the application is compiled for a target platform. This affects a little on the size of the resulting JAR (Java Archive) file that is loaded into the device. The JAR size and optimatizations are discussed more detail in the next Section.

4 Memory requirements

For mobile phones, as for all mobile devices, there are strict demands for application memory management. Usually the amount of memory available is very restricted, which has to be taken into consideration in application design. Two aspects in this field, the storage needed to store the application jar file and the runtime memory needed for running the application are considered in corresponding subsections.

QX.java
NokiaSMSHandler.java
OutgoingTextMessage.java
SiemensDebugLog.java

4.1 Storage memory and JAR size

Storage memory is usually quite limited in mobile devices. This limits the amount of applications a user can store in the device at one time. For this reason it is significant how much storage an application needs when loaded into the phone. With MIDlets the JAR size is the predominant factor when considering how much storage space is needed. JAR size also affects OTA(Over the Air) download time and cost, which is an important factor, since more often the user pays for the downloads per kilobyte.

JAR size can be affected at design time with some design decisions. In object oriented programming, separating various levels of functionality in separate classes has some merit. However, in MIDlets corporating more functionality in a class is worth considering, as it reduces overhead that comes with numerous class definitions. Also one should use as few interfaces as possible, since they take space without offering any functionality.

In Java programming it is common to use package names in applications, a feature that can unnecessarily increase the JAR size. Also, when using self-made libraries it is crucial to make sure that everything in a library is really needed, since unnecessary classes and methods are waste of storage memory.

JAR sizes for our appliacation are as follows:

Nokia: 16265 bytes
Siemens: 17129 bytes
WTK: 16167 bytes,

where WTK is the Wireless Toolkit provided by Sun MicroSystems.

4.2 Obfuscation

Although Java byte code (.class file) is unreadable for human, several reverse engineering tools can decompile compiled byte code into source code. This is especially important issue for companies whose revenue is generated through developing and selling Java applications. Programs called Obfuscators have been developed for preventing this problem by rendering the byte code in a way that makes the decompiled bytecode more difficult to understand. Several different schemes [11] for obfuscation have been developed, but they are out of scope of this paper.

For MIDlets goal of obfuscation is instead of harder reverse-engineering, in decreasing the JAR size. Obfuscation usually shortens the names of classes, variables and methods resulting in smaller amount of bytecode. Obfuscators called Zelix KlassMaster and Retroguard were used to obfuscate applications JAR package. Both produced an approximate 10% decrease in JAR size, which is consistent with results reported by Nokia.

After obfuscation (with Zelix) the JAR sizes were:

Nokia: 15236 bytes
Siemens: 15736 bytes
WTK: 15165 bytes

4.3 Runtime memory usage

Even more limited than the amount of storage memory in mobile devices is often the amount of available heap memory (i.e. runtime memory). Developing MIDP applications there are several ways to consider this issue design time. On Java platforms, the garbage collector plays a very important role in keeping the memory from filling with objects that are no longer needed. As an application developer has very limited control over the garbage collector it is important that garbage collection system in Java is understood and supported by the application design. In our program the screens are by far the biggest consumers of memory, which means that if one wants to save memory one should create them only when they are needed and release them as soon as they are not needed. However no problems were noted even when screens were created and kept in memory throughout application execution. Keeping screens in memory allows the user to send the same message to several recipients without retyping. Also it is possible to send several messages to the same recipient without retyping the number and password. Runtime memory usage was measured using a following test:

```
int iAmount = 100;
Object[] objects;
long lFree = 0;
long lUsed = 0;

objects = new SMSScreen[iAmount];
Runtime.getRuntime().gc();
lFree = Runtime.getRuntime().freeMemory();
for(int i = 0; i < iAmount; i++)
   objects[i] = new SMSScreen(this, ‘‘TEST’’,1);
Runtime.getRuntime().gc();
lUsed = lFree - Runtime.getRuntime().freeMemory();
DebugLog.makeEntry(iAmount+‘‘ SMSScreen) used: ‘‘+lUsed);
for(int i = 0; i < iAmount; i++)
   objects[i] = null;
```

In this method a variable number (20 - 100) of each object were created and the amount of memory needed was recorded. This way the effects of normal fluctuation of memory allocation were minimized. The memory allocated to each object is given in Table 1. Sun Microsystems WTK provides a tool for monitoring the memory allocation of objects. These results are shown with WTK* in Table 1.

The object (QX.java) for encryption/decryption is rather small, consuming only 5-10% of the amount of memory needed for one screen.

5 Requirements on processing power

The main phase where processing power is needed is the encryption/decryption method. Time needed for encrypting or decrypting a message of a certain length was measured with following way:

Table 1. Memory requirements in bytes of each object used.

Name of the class	Siemens	Nokia	WTK	WTK*	Remarks
CommonLog.java	N/A	N/A	N/A		Abstract super class
CryptSms.java	87	258	-	-	
DebugLog.java	N/A	N/A	N/A	N/A	Static class
NokiaDebugLog.java	N/A	41	N/A	N/A	
NokiaSmsHandler.java	N/A	264	N/A	N/A	
NUMScreen.java	2595	1255	1465	104	
OutgoingTextMessage.java	24	52	44	24	
PASScreen.java	2590	1257	1469	108	
QX.java	132	164	172	52	
Receiver.java	20	N/A	N/A	N/A	
SiemensDebugLog	16	N/A	N/A	N/A	
SiemensSmsHandler	284	N/A	N/A	N/A	
SmsHandler.java	N/A	N/A	N/A	N/A	Abstract super class
SmsScreen.java	448	603	400	-	
WmaDebugLog.java	N/A	N/A	12	12	
WmaSmsHandler.java	N/A	N/A	128	48	
WriteSmsScreen.java	2592	1254	1465	104	

```
long lMillis = System.currentTimeMillis();
// Operation that needs to be measured...
long lTimeSpent = System.currentTimeMillis() - lMillis;
DebugLog.makeEntry(‘‘Encryption time’’ + lTimeSpent);
```

Clearly, this doesn't give exact values, but a rather good estimate. The results seen in Figure 3 go well hand in hand with what was expected from the encryption. As it should be, the complexity grows linearly with respect to the length of processed data. Encryption times for messages with length from 1 to 33 characters are shown in Figure 3. Two passwords of different length were used and with the shorter one the encryption was repeated 4 times, while with the longer password it was repeated 16 times.

If the latin square used is of order n, holding n^2 values in the range $1 \dots n-1$ then $log(n)$ bits can be encrypted at once. On the other hand the seek time to find a result value corresponding given input values from the square is at most $O(2n)$. This means that a bit string with k bits can be encrypted in time

$$t = \frac{k}{log(n)} 2n. \tag{7}$$

The Latin square can be sorted either by row or column in implementation. This will decrease the seek time into $O(n)$. As a result, the time needed to encrypt k bits would be

$$t = k \frac{n}{log(n)}. \tag{8}$$

The length of the password has direct effect on both the speed and the security of the algorithm. As usual, this is a tradeoff between security and speed. Encryption with an m bit password is repeated $r = \lfloor \frac{m}{log(n)} \rfloor$ times. On every round $log(n)$ bits of password material is used. The total time needed to encrypt k bit data using a m bit password is

$$t = k\frac{2n}{log(n)}\lfloor \frac{m}{log(n)} \rfloor. \tag{9}$$

6 Conclusions

The fact that characters of the password are limited to printable characters would suggest that a less costly operation with the same level of security could be achieved by using only some bits of each password byte. The state of the encryption object could be used to determine these bits for the next round after each round.

In our application a fixed size Latin square of order 4 was used, which means that 2 bits of key material are used on each round. This means that even a quite short password of 4 characters will result in 16 rounds of encryption. A brute force attack on password space will in this case be probably more successful than attacking the algorithm itself. Possibly a key schedule that would result in one round of encryption for each password character would be more cost effective.

The research on crytographic strength of quasigroup encryption is still in early stages. The cryptosystem has not yet undergone much scrutiny from the cryptographic community. Several widely used cryptosystems today are conjectured to be safe, hence after extensive study by cryptographers they seem to be safe. This research has not yet been done to satisfactory extent on quasigroups. The application design itself doesn't restrict using any suitable encryption algorithm.

Quasigroup encryption seems to be well suited for applications such as SMS encryption. The algorithm is compact and needs quite a small amount of memory, which is an important aspect on mobile devices. Also its performance fulfills set requirements for this application, the encryption delay experienced by user is bearable. If needed, this delay can be hidden completely by doing encryption at the backround in a separate thread at the same time the user is prompted for a number where the message is to be sent. In light of research results, an estimate for encrypting a full length SMS message of 160 characters using 16 rounds of encryption would be 4 seconds. The user will easily spend this time typing a phone number.

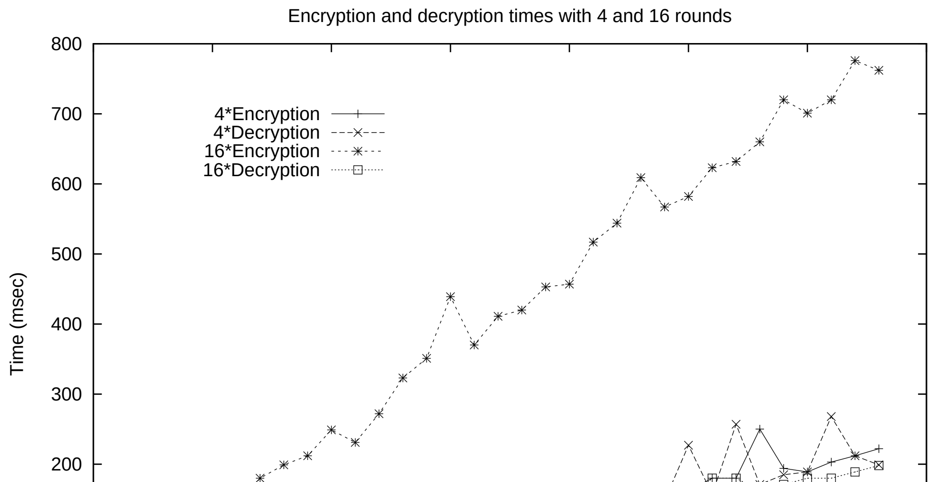
A clear difference on Nokia and Siemens environments is the way one can handle incoming messages. On Nokia platform the application has to be running and messages are received by the application. An incoming encrypted message can then be decrypted and shown. It was reported in [12], that MIDP 2.0 will define a mechanism called 'Push', which makes it possible to register a certain port for a certain application. Further, it is possible to launch a specific application when a message is received on a registered port. Currently there are no Nokia devices that support MIDP 2.0.

On Siemens platform on the other hand, receiving incoming messages by the application is not possible. However, on Siemens incoming messages are received by the phone and saved to inbox, from where the application can read and decrypt an encrypted message.

The usability of the application would benefit from a phonebook where one could store names, numbers and passwords (secret keys). Naturally this phonebook should be protected from unauthorized use.

References

1. Markovski, S.: Quasigroup string processing and applications in cryptography: Proceedings of the Conference MII 2003, 14 - 16 April, Thessaloniki, Greece (pp. 278-290)
2. Markovski, S., Gligoroski, D., and Bakeva, V.: Quasigroup String Processing: Part 1, Contributions, Sec. Math. Tech. Sci., MANU, XX 1-2(1999) 13-28
3. Markovski, S., Kusakatov, V.: Quasigroup String Processing: Part 2, Contributions, Sec. Math. Tech. Sci., MANU, XXI 1-2(2000) 15-32
4. Java Community Process JSR-000118 Mobile Information Device Profile 2.0 http://jcp.org/aboutJava/communityprocess/final/jsr118/
5. Sun Microsystems User Guide, Wireless Toolkit, Version 2.0. Java Platform, Micro Edition http://java.sun.com/
6. Nokia: A MIDlet Example Using the Wireless Messaging API and the Nokia SMS API: Chat v1.0
http://www.forum.nokia.com/
7. Nokia Java MIPD application developer's guide for Nokia devices, Version 1.0 http://www.forum.nokia.com/
8. Nokia Efficient MIDP Programming http://www.forum.nokia.com/
9. Siemens: Programmer's Reference Manual
http://communication-market.siemens.de/
10. Siemens: OTA Specification Release 1.0.2
http://communication-market.siemens.de/
11. Hongying Lai: A comparative survey of Java obfuscators available on the Internet: Project Report 22 February 2001, Computer Science Department, University of Auckland. http://www.cs.auckland.ac.nz/ cthombor/Students/hlai/
12. Chiam Poh Guan: Introduction to Nokia's MIDP Extensions and Development tools: Forum Nokia Developer Conference, 17-19 June 2002


Encryption and decryption times with 4 and 16 rounds
Time (msec)
800
700
600
500
400
300
200
4*Encryption
4*Decryption
16*Encryption
16*Decryption