Security API for Java ME: secureXdata

Remi André B. Valvik

Institute of Informatics
University of Bergen
Norway

Long Master Thesis
February 2012

Foreword

This document is the result of my masters degree in informatics at the University of Bergen.

I would like to use this opportunity to give special thanks to my supervisors Federico Mancini and Khalid A. Mughal for their dedication, support and advice throughout my masters thesis. I would also like to thank Samson Gejibo who has been working on the server side of things, for his support and cooperation during the work. Thanks to Jørn I. Klungsøyr for his collaboration and to Marc Bezem for introducing me to the project.

Thanks to my fellow Master student Are Venes for his support and cooperation in the courses leading up to my thesis, and Aleksander Vines for interesting discussions and moral support during the final weeks.

Finally, I would like to give thanks to my family and friends for their support throughout my studies, with special thanks to my better half, Jenna for her loving support, encouragement and delicious food.

Remi André B. Valvik
Bergen, February 14, 2012

Contents

List of Figures

List of Tables

Code Listings

1 Introduction

1.1 Advent of mHealth

The usage of mobile phones, PDAs and other mobile communication devices in the context of health is an emerging part of eHealth. In 2010 the American National Institutes of Health defined mHealth as "the delivery of healthcare services via mobile communication devices" [10], and while the idea of using mobile devices to provide healthcare related services is a very convenient prospect in industrialized countries, it is having a much more profound impact on the healthcare situation in low-income countries.

A report by the United Nations Foundation and Vodafone Foundation [30] suggests that close to half the population in low-income countries own or have access to mobile phones. Healthcare in these nations can be scarce or difficult to access due to restraints such as limited resources, finances and healthcare workforce, or parts of the population living in remote locations. High mobile phone penetration makes mHealth a viable option for providing better healthcare through mHealth systems.

Leveraging mHealth systems makes it possible to cost-effectively provide things like:

- General education and awareness through, for instance, SMS.
- Communicating with and training of healthcare workers, even in remote locations.
- Tracking of disease outbreaks or epidemics.
- Diagnosing and assisting in treating patients remotely when traveling to a hospital is

not an option.

In this thesis we are mainly focused on a specific aspect of mHealth, namely remote data collection. However, the work presented could prove useful for other areas as well.

Using mobile devices, healthcare workers can gather important data about the condition and trends of a country's health status not only in central areas where people have access to medical facilities, but also in remote locations. This information can then be submitted quickly to policymakers and healthcare providers, enabling them to make well informed decisions and better spend the available resources in the best possible manner.

1.2 Mobile Data Collection

As we have briefly discussed, mobile data collection is the process of gathering data using a mobile device such as a mobile phone or PDA. Since the data collected can be medical records or patient forms, it can be of a highly private or sensitive nature. These concerns are often neglected when it comes to mHealth systems. Private data is stored on the device in a non secure way, and even though most of these systems support HTTPS to secure transmissions, it is not always the case that it is used. We will have a closer look at mobile data collection in the next chapter.

1.3 mHealth Security Group

The mHealth Security Group at the Department of Informatics in collaboration with openX-data at the Centre for International Health, both at the University of Bergen, are developing a protocol for securing both data at rest on a mobile device, and data being transmitted over insecure networks. The work in this thesis is based on and part of a further development of the protocol and the systems surrounding it. The original protocol paper can be found at [34]. In the next chapter we will look closer at the protocol and its purpose.

1.4 Motivation, Objectives and Challenges

mHealth is making way for easily available and low cost improvements in health care systems, especially in low-income countries where this is needed the most. However, many systems in this field fail to systematically address the security and privacy concerns while handling private or personal information such as medical records.

Building on the existing work done by the mHealth Security Group, it is desirable to make an easy to use API that can be incorporated into existing mHealth systems to provide some acceptable level of security.

Objectives The overall objective in the thesis is to extend the current prototype into a comprehensive API that can be further developed and tested through field and usability studies, ultimately allowing for the easy creation secure Java ME applications conforming to some security requirements.

We can split this task into a number of sub-objectives:

- Redesign the prototype/proof of concept implementation into a more usable API design.
- Integrate the API into the existing openXdata client.
- Evaluate the API design based on performance and the integration with openXdata.

1.5 Chapter outline

Chapter 2: Security Issues in Mobile Data Collection Systems This chapter will give an overview of the constraints and issues around which the protocol is designed. We will also give a short overview of the protocol itself.

Chapter 3: Record Store and Serialization Performance In this chapter we benchmark the record store and serialization on the device, the results are used in chapter four.

Chapter 4: Implementing secureXdata In this chapter we will go through the requirements on which the API is based and describe the resulting API while discussing why it was designed in this way, and highlight any issues that might need to be addressed in future refinement of the API.

Chapter 5: Implementation Evaluation This chapter discusses the integration with the openXdata client. The API is evaluated in terms of how easy it is to use and integrate, and how well it performs on an actual mobile device.

Chapter 6: Experiences In this chapter the candidate evaluates the tools used throughout the work on the thesis and shares general experiences and some guidelines/pitfalls related to working with Java ME and other technologies.

Chapter 7: Conclusion and Future Work In this chapter we conclude the work that has been done and present a more in depth explanation/discussion of aspects of the API that needs further work.

Appendix A This appendix contains additional results from benchmarks performed as part of chapter three.

2 Security Issues in Mobile Data Collection Systems

2.1 Mobile Data Collection and Security

As mentioned in the introduction, mobile data collection systems (hereafter referred to as MDCS) are often used to collect medical data as a part of mHealth systems. There is more than one way to perform such data collection. For instance, automated sensors can be placed in the field and periodically transmit collected data, but in this thesis we are interested in MDCS where data is collected through forms filled out by people, such as healthcare workers.

These MDCS usually consist of mobile clients in the field and a centralized server that the clients communicate with. Using a form designer, generally not on a mobile device, a set of relevant questions is put together to comprise a form. The form is then stored on the server and can be retrieved by the appropriate collector in the field. Once a form has been filled out by a healthcare worker or other data collector, it is uploaded to the server and either analyzed or stored for later use.

2.1.1 Security Concerns

MDCS used in mHealth can clearly handle and contain highly sensitive or private data, and as such some security concerns arise. In the paper [38] based on [34], the mHealth Security

Group from the University in Bergen discusses these security concerns and evaluates how they are handled in some of the existing MDCS. It is noteworthy that the paper contains some changes in the protocol since the original [34] was published and parts of the API design discussed in the paper is based on the work described in this thesis.

2.1.2 Confidentiality and Availability

Collected data, both at rest on the device and being transmitted to the server, needs to be properly protected from unauthorized disclosure.

In most cases the devices are communicating using some mobile network and/or through the Internet. As such, it is possible for a third party to intercept the transmission. All the MDCS discussed in [38] support HTTPS for server communication, but as we will see later in this chapter this is not always feasible.

The same applies to the data stored on the device. Unlike a computer in a locked room, a mobile device can easily be accessed by a third party, be it intentionally (by an attacker stealing the device) or by chance (the phone is dropped/lost). Without proper confidentiality measures in place, the data will be available for anyone to read or tamper with if they should get physical access to the device.

If confidentiality is ensured on storage, then this raises another issue, availability. Should a data collector lose his or her password, the data secured on the device needs to still be recoverable. And furthermore, resetting the password or user account on one device, should be completely independent from any other devices the user might be registered on.

2.1.3 Authentication and Authorization

Regarding authentication there are two aspects to consider. The first is local authentication on the mobile device. As we will get back to later in this chapter, the device may be shared among multiple users and as such there should be some authentication in place to make sure that the application can trust the user. Be it to protect from a third party trying to gain access to the application or a registered user trying to access another users data.

Secondly, authentication on the server is needed as well. It should only be possible for a legitimate user in the system to send and retrieve data or forms to and from the server. In short: users should be authenticated on the server as well as on the device. By using HTTPS, the client can authenticate the server through its certificate. However, if HTTPS is not used, the server should be authenticated in some other manner so that the client can verify/trust that it it is talking to the genuine server.

Authorization becomes a concern when there are multiple users sharing a single device. Even though they may have different credentials for authentication, unless authorization is taken into account they will have access to each others data on the device. As discussed in [38], some of the MDCS investigated will allow users to access each others data due to lack of access control. Furthermore, as long as a someone has physical access to the device and the data stored on it is not properly encrypted, a third party tool can be used to extract the data from the device regardless of both access control and local authentication.

2.2 OpenXData

The work that this thesis is based on ([34], [36] and [38]) originated from a collaboration with openXdata [12]. OpenXdata is a open-source MDCS which has been primarily used in health related projects in low-income countries. OpenXdata follows the previously described concept of downloading forms from a server and filling them out on mobile devices, and is being used in a number of projects in low-income countries such as Pakistan [26] and Uganda [25]. A preliminary assessment of the system showed that security concerns were not systematically addressed and what's worse, this was not isolated to openXdata. A more in depth discussion regarding this can be found in [38].

2.2.1 Security Analysis Overview

Since we will be using openXdata as our reference system in this thesis, we will have a look at some of the security issues found within the openXdata system.

When work on the thesis started, the openXdata client provided local and server authentication: once a user tried to log in, a check was performed to see if the user credentials were stored hashed in a record store locally on the device. If this was not the case, the credentials would be submitted to the server as plain-text. If the user credentials were authenticated on the server, the database containing all user names, salts and salted hashed passwords would be sent to the device and stored in the record store. The initial local authentication would then be done on the newly acquired user database. If a match was found, then the user would be logged in. By downloading the entire user database, local authentication would be available to all registered users once the initial user had been authenticated.

Currently, the situation has improved slightly by the fact that they no longer download and store the entire database of users on the device. Only the data for the user trying to log in is downloaded and stored.

Regarding communication, the openXdata client, as mentioned, supports HTTPS, but, as we will discuss, this is not always applicable. The data stored on the device is not secured in any way. It is not encrypted and as such, even without being authorized, the persistent data can be dumped to a computer and read. OpenXdata also lacks authorization on the device. Any user that is authenticated has access to all the data stored, regardless of who created/saved the data. Valid user credentials are however required with every server request.

2.3 Practical Constraints and Requirements

We have briefly looked at some of the requirements and issues with current MDCS. Many of the projects using this form of MDCS are deployed in low-income countries, and as such it is not as straightforward to apply security as it could be in other parts of the world. In this section we will give a summary of what constraints exist and what impact this might have on security. For a more in depth discussion on the subject the reader is encouraged to read [38].

2.3.1 Budget

A major issue with trying to secure MDCS stem from the fact that most of them have very restrictive budgets. Without money to provide powerful devices such as smartphones, it can prove difficult to achieve good security without adding considerable overhead for cryptography. Securing the data could also add additional costs through for instance increased traffic due communication overhead.

Because of the budget limitations, we focus on low-end Java enabled devices, sometimes referred to as feature phones [60]. These devices are relatively cheap compared to smartphones.

2.3.2 Remote Working Locations and Phone Sharing

Some of the projects using MDCS are working in remote locations and in countries where the mobile communication infrastructure might not be fully developed. As a result, there may not be network coverage in certain areas or villages where the data collectors work. Having to travel many kilometers to get a network connection might be something that is done every couple of days instead of daily, and as such data could reside on the device for long periods of time. This means that offline authentication needs to be available in order to log in without a server connection, and any data stored on the device needs to be secured.

It can not be assumed that one collector has a single phone for his or her use, and as such phones might be shared. This means that in addition to user authentication, data authorization is needed as well. Otherwise, all the users of a single device will have access to the other users' data.

2.3.3 Ease of Usage of The API

Even with a system such as our API that is made to cover all the requirements and limitations discussed so far, if it is hard to use or requires the MDCS to undergo large changes in order to be secured, few people would use it. As such, any system addressing security for MDCS needs to be easy to use and implement into existing systems for the developers, but also easy to use for the end users.

2.4 Common Mobile Security Solutions

There exist some commonly used protocols and solutions that are used for securing information and communication systems. However due to the discussed constraints, these might not be applicable in the context of MDCS in low-income countries.

2.4.1 HTTPS and Certificates

The Hyper Text Transfer Protocol Secure is designed to allow secure HTTP communication using SSL/TSL [41]. There are two issues with HTTPS that makes it less than ideal for MDCS, the first being certificates. Certificates are the means by which the client can trust the server it connects to or vice-versa. However, this trust comes at a price. Depending on what certificate authority one use, the price of a single certificate can vary from being free to many thousand dollars a year.

If the MDCS is offered as a service, as is the case with Mobenzi [11], the service provider (Mobenzi) has to pay the certificates and the cost is distributed across all of their customers. Making this a more viable solution. However, for projects which run their own servers this would most likely be too expensive to be a practical solution. A second problem with certificates is that the list of certificate authorities a mobile device will accept, is not standardized and is controlled by the manufacturers. Not all devices trust the same certificate authorities, meaning that more than one certificate might have to be purchased.

The second issue with HTTPS is establishing the secure connecting with the server, also known as the handshake. The devices used in the intended MDCS are most likely of a low-end character and as such would spend a lot of time processing the different cryptographic operations involved in the initial stages of the communication. The handshake consists of a number of requests and responses, which further worsens the situation since many areas where these systems are used have unstable or limited network coverage/connectivity. HTTPS was intended for a different scenario: establishing a secure and trusted connection with a unknown server on the Internet. With MDCS however, in many cases the server should be known to the client and therefore the handshaking step can simplified, thus reducing the load on the device and increasing the tolerance for connectivity issues.

2.4.2 Other Platforms

Other more powerful advanced platforms (smartphones) like Android [2], BlackBerry [4] or iOS [3] have native support for securing data stored and also run on higher grade hardware and will therefore not have as much of an issue with HTTPS handshakes and other cryptographic operations being slow. HTTPS would still suffer from connectivity issues though. However, smartphones are most likely not viable alternatives due to budget restrictions.

2.5 Quick Protocol Overview

The protocol itself is designed to be platform and application independent and will provide a secure layer on top of an existing application layer. An implementation of the protocol would generally take the form of an API providing some secure services: secure storage for securing data at rest on a device, secure transmission and user management including registration and authentication with password recovery. The aim is to be able to provide a “good-enough” level of security based on the requirements and restraints of any given application or project. The intended area of use is mainly securing MDCS in low-budget settings.

The protocol is considered a part of the public domain and is still being developed. While the working requirements and overall goal have stayed unchanged, the inner workings of the protocol have evolved since the original paper was published. The evolution and current state can be better understood by going through the papers describing the protocol. In chronological order:

- **Adding Security to Mobile Data Collection [34]** This paper introduces the issue of lacking security in mHealth systems and outlines some working assumptions based on experience from openXdata. The protocol is presented and discussed.
- **Securing Mobile Data Collection [35]** In this paper an API designed on the proposed protocol in [34] is described. This API is the basis for the work in this thesis. More implementation specific details regarding the protocol and some preliminary performance tests are also discussed.
- **Challenges in Implementing End-to-End Secure Protocol for Java ME-Based Mobile Data Collection in Low-Budget Settings [36]** This paper discusses some of the the challenges faced when making an API implementation of the discussed protocol. Some of the solutions proposed here are based on the work presented in this document.
- **End-to-End Secure Protocol for Mobile Data Collection Systems in Low-Budget Settings [37]** This paper discusses the working assumptions under which the protocol is intended to work and presents an improved version of the protocol on a high abstraction level.
- **On the Security of Mobile Data Collection Systems in Low-Budget Settings [38]** In this paper a security review of a number of MDCS is presented and available security solutions are discussed. A further evolved version of the protocol is also introduced, without any implementation details.

The candidate started working on the project shortly after “Securing Mobile Data Collection” [35], was written and has since been a part of the protocol development as well as designing the API.

2.6 Technologies and Devices

In the next chapter we will look the development of the API implementation of the protocol based on [34]. In the implementation the following technologies have been used:

Java ME [52] Java ME is the platform used for developing the API and applications that will run on the device.

Eclipse [32] Eclipse is the IDE that will be used for development and the writing of this document.

Sun Java Wireless Toolkit 2.5 [55] WTK 2.5 is one of the Java ME toolkits used during the development process.

Java ME SDK 3 [53] Java ME SDK 3 is the other toolkit used during the development process.

Bouncy Castle [8] Bouncy Castle is a third party cryptography library used in the API.

Ant [29] And is the build tool used in the development work done.

In chapter 6 the candidate will discuss his experiences with the different technologies.

Since we are developing software for mobile devices, we will be using emulators (part of the toolkits used) for testing on the computer. But in the end it is how the API works and performs on actual devices that matters. We will primarily be using the Nokia 2330c-2 [47] for benchmarks and testing. Based on preliminary performance testing that can be found in [36], this phone has fair performance while still being inexpensive. Other devices will be tested in the future.

3

Record Store and Serialization Performance

3.1 Benchmark Background

As can be seen from the preliminary tests results in [36], the performance of the Record Management Store (RMS) systems on the different phones vary. Because of this, it is essential to see which aspects or features of the RMS system is most costly and base the design of the API around these findings.

Some existing benchmark tools for Java ME enabled devices were used such as RMStress [6] which didn't work out of the box (nullpointer exception), once fixed and working it produced very inaccurate results due to display update (progress gauge) code being within the benchmarked code section. TastePhone [17] was also run which produced a lot of general information, but not enough for our purpose, as a result we created our own benchmarks.

The basic operations for working with the MIDP RMS system are listed below [40],

Opening Record Stores Open and if required to create a record store.

Closing Record Stores Close an opened record store.

Adding Records Create a record in a open record store, and add some data to it. The data can not be added in chunks, it has to be written as a whole to the record.

Updating Records Update an already existing record in an open record store. This will overwrite any data already in the record. As with adding records this cannot be done in chunks.

Reading Records Read the content of an existing record in an open record store into memory so it can be used by the application.

Deleting Records and Record Stores Delete a record or an entire record store.

Creating and deleting record stores are operations not frequently used in the API. Nor can much be done to optimize the use of these operations when designing the application. As such, these aren't benchmarked.

It is noteworthy that Java ME does not support automatic serialization, ie. you can't implement the serializable [50] interface as it is not a part of Java ME. When talking about serialization, we mean that the object have methods for turning itself into a storable format and to initialize itself from such a format. The way this is done in the API is by writing all the fields into a byte array using a stream writer.

3.2 Benchmark setup

All the RMS benchmarks are done on an empty store, more specifically, before each test, the entire record store is deleted and created with no content. The serialization benchmarks which will be discussed later, are done on newly created buffers.

All benchmarks are run on the target phone Nokia 2330c [47].

To more easily explain how the benchmark is done, we will define some terms that will be used in the following explanations:

Feature - A feature we want to test. Writing a record for instance.
Test - Performing a feature and timing (in ms) how long it takes.
Iteration - Running a test once.
Benchmark - Running a number of iterations, returns the total time.

The final result is determined in the following way: For each feature the benchmark is run 10 times with each of the different data sizes. Each benchmark consists of 5 iterations. Out of these 10, the minimum of maximum result from each data size is discarded to minimize errors. The remaining 8 values are then averaged giving the final result for 5 iterations, this is then divided by 5 to give the per iteration result.

The hypothesis is that it is more expensive, time wise, to store an amount of data in multiple records than it is to store the same data in a single record. This can be achieved by serializing the different parts of data in an object into one larger byte array and storing this in one record. Since most of the data is not static in size, we also need to look at how size variation impacts update times of a record.

At this point we are interested in determining how to design our record stores, and as such we primarily want to determine if storing all the data as one big chunk is better than multiple minor records. The additional benchmarking details regarding updating with different data sizes can be found in A.1. Since serializing data requires processing it is important to make sure that serialization does not take longer than any potential gain by using a single record, as such we benchmark this as well.

Benchmark A - Store record This benchmark will determine the time it takes to store a number of records of a given size in an existing record store.

Listing 3.1: Benchmark for record write time.

```java
// Recordstore rms is opened, and contains no records.
long result, total = 0;
for (int i = 0; i < itt;i++) {
  bmData = genRandomByteArray(size);
  startClock();
  rms.addRecord(bmData, 0, bmData.length);
  result = stopClock();
  total += result;
}
```

Benchmark B - Update record This benchmark will determine the time it takes to update an existing records data with some new set of data of the same size.

Listing 3.2: Benchmark for fixed sized record update time.

```java
// Recordstore rms is opened, and contains 1 record of "size"
    bytes.
long result, total = 0;
for (int i = 0; i < itt;i++) {
  bmData = genRandomByteArray(size);
  startClock();
  rms.setRecord(1,bmData, 0, bmData.length);
  result = stopClock();
  total += result;
}
```

Benchmark C - Read record This benchmark will determine the time it takes read an existing record into a byte array.

Listing 3.3: Benchmark for dynamic sized record update time.

```java
// Recordstore rms is open, contains 1 record of "size" bytes.
long result = 0, total = 0;
for (int i = 0; i < itt;i++) {
  startClock();
  bmData = rms.getRecord(1);
  result = stopClock();
  total += result;
}
```

Benchmark D - Delete a record This benchmark will determine the time it takes to delete an existing record.

Listing 3.4: Benchmark for record deletion time.

```
// Recordstore rms is open, contains no records.
long result = 0,total = 0;
// Note that once a record is deleted, the record id will never
// be reused, which is why we can’t simply refer to record 1.
int id;
for (int i = 0; i < itt;i++) {
  id = rms.addRecord(bmData, 0, bmData.length);
  startClock();
  rms.deleteRecord(id);
  result = stopClock();
  total += result;
}
```

Benchmark E - Update record unevenly Benchmark B updates with a record of the same size, as this is not always the case, this benchmark will determine the time it takes to update when the stored and new data are of different sizes.

Listing 3.5: Benchmark uneven record updating.

```
// Record store is open, contains 1 record.
// initSize is a percentage (can be over 100%) of size.
long result, total = 0;
for (int i = 0; i < itt;i++) {
  bmData = genRandomByteArray(initSize);
  rms.setRecord(1,bmData, 0, bmData.length);
  bmData = genRandomByteArray(size);
  startClock();
  rms.setRecord(1,bmData, 0, bmData.length);
  result = stopClock();
  total += result;
}
```

Benchmark F - Add multiple data This benchmark will determine the actual time it takes to fill the record store with a fixed amount of data by using multiple smaller pieces. Comparing this to the results from benchmark A’s results might give some interesting results. Ie. will the measured time for writing 512 bytes into 16 32 byte records be higher than the estimating results found by multiplying the results for adding a single 32 byte record by 16?

Listing 3.6: Benchmark multiple records.

```
// Recordstore rms is open, contains no records.
```

```
long result, total = 0;
for (int i = 0; target > (i * chunkSize);i++) {
  bmData = genRandomByteArray(chunkSize);
  startClock();
  rms.addRecord(bmData, 0, bmData.length);
  result = stopClock();
  total += result;
}
return total;
```

Benchmark G - Serialize data This benchmark will determine the time it takes to serialize a number of bytes into an array using a `ByteArrayOutputStream` and `DataOutputStream`.

Listing 3.7: Benchmark for data serialization.

```
long result, total = 0;
for (int i = 0; i < itt;i++) {
  // Close buffer/stream.
  startClock();
  SerializationHelper.writeByteArrayToStream(out, bmData);
  result = stopClock();
  total += result;
  // Close buffer/stream.
}
```

Benchmark H - De-serialize data This benchmark will determine the time it takes to de-serialize a number of bytes into an array using a `ByteArrayInputStream` and `DataInputStream`.

Listing 3.8: Benchmark for data serialization.

```
long result, total = 0;
for (int i = 0; i < itt;i++) {
  // Set up buffer/stream.
  startClock();
  bmData = SerializationHelper.readByteArrayFromStream(in);
  result = stopClock();
  total += result;
  // Close buffer/stream.
}
```

Benchmark I - Overhead serialize data This benchmark will determine the overhead time for creating and opening the buffer and stream used while serializing. `ByteArrayOutputStream` and `DataOutputStream`.

Listing 3.9: Benchmark for data serialization.

```
long result, total = 0;
for (int i = 0; i < itt;i++) {
```

```java
  startClock();
  ByteArrayOutputStream baos = new ByteArrayOutputStream();
  DataOutputStream out = new DataOutputStream(baos);
  result = stopClock();
  total += result;
  // Data is written.
  startClock();
  bmData = baos.toByteArray();
  out.close();
  baos.close();
  result = stopClock();
  total += result;
}
```

Benchmark J - Overhead de-serialize data This benchmark will determine the overhead time for creating and opening the buffer and stream used while de-serializing.

Listing 3.10: Benchmark for data serialization.

```java
long result, total = 0;
for (int i = 0; i < itt;i++) {
  // bmData contains some serialized data.
  startClock();
  ByteArrayInputStream bais = new ByteArrayInputStream(bmData);
  DataInputStream in = new DataInputStream(bais);
  result = stopClock();
  total += result;
  // The data is read.
  startClock();
  in.close();
  bais.close();
  result = stopClock();
  total += result;
}
```

3.3 Results and Conclusion

3.3.1 Record Store

Looking at table 3.1 we can see that for all operations in this table the total time is more dependant on the operation than the data size, this could indicate that the actual writing of bytes to storage takes less time than setting up the record and other book keeping. In other words, the increase of data size does not seem to effect how long it takes to complete an operation to a very large extent. When comparing the ratio between the largest (20480 bytes) and the smallest (32 bytes) record size used, the difference in size is 64 000%, but

Data Size (bytes)	Write	Update	Read	Delete
32	1,5	0,6	0,8	0,4
64	1,4	0,7	0,1	0,5
128	1,5	0,7	0,8	0,4
256	1,6	0,8	1,0	0,4
512	1,8	0,8	0,5	0,5
1024	1,9	0,9	1,1	0,4
5120	2,6	1,2	4,0	0,6
10240	3,0	1,5	8,0	0,7
20480	4,4	2,3	16,3	0,8
25600	4,8	2,8	21,8	0,9

Table 3.1: Benchmark results for basic RMS operations. The results are the time it takes to perform the different operations. All times are in ms.

Data Size (bytes)	Measured	Calculated
32	33,6	23,6
64	13,9	11,0
128	5,8	5,9
256	2,9	3,1
512	2,1	1,8

Table 3.2: Benchmark results for write operations in multiple records, a total of 512 bytes are stored in records of “data size” bytes each. The calculated value is based on the write results in table 3.1.

Data Size (bytes)	Measured	Calculated
32	273,8	236,2
64	144,6	110,1
128	71,6	59,0
256	36,1	31,0
512	18,0	17,8
1024	10,1	9,3
5120	2,0	2,6

Table 3.3: Benchmark results for write operations in multiple records, a total of 5120 bytes are stored in records of “data size” bytes each. The calculated value is based on the write results in table 3.1.

the time difference however is 320% for writing, 466% for updating, 2 725% for reading and 225% for deleting.

Furthermore, when looking at tables 3.2, 3.3 and 3.4, we can see that the actual time it

Data Size (bytes)	Measured	Calculated
32	1037,0	944,6
64	497,4	440,3
128	253,0	236,2
256	135,4	124,0
512	72,6	71,0
1024	37,8	37,0
5120	8,6	10,4
10240	5,8	6,0
20480	3,3	4,4

Table 3.4: Benchmark results for write operations in multiple records, a total of 20480 bytes are stored in records of “data size” bytes each. The calculated value is based on the write results in table 3.1.

takes to write an amount of data as many small pieces is, on average, even larger than what figure 3.1 suggests. This is either because it takes longer to write once the number of records go up, or because the values in figure 3.1 are lower than the actual average. Regardless we can see that the accumulated overhead for smaller pieces of data becomes nontrivial once the number of pieces go up.

It is noteworthy at this point that because the operations are timed in milliseconds, and some of them on average take 1 ms or less, some inaccuracy is to be expected in the findings.

In addition to the above presented data, some benchmarks have been done on updating under different scenarios (the existing record is larger / smaller than the data it is update with). It seems that regardless of the size or configuration of the record, not surprisingly, it is quicker to update than to write the same amount of data to a new record using the `setRecord()`. The results and descriptions can be found in appendix A.1.

3.3.2 Serialization

We have seen in the record store benchmarks that using a single record for storing a chunk of data can be far more effective than storing the same data in multiple pieces. However, if transforming the many pieces of data into a single array of bytes is more costly (time wise) than what we gain by storing this way, then it ultimately becomes meaningless.

Table 3.6 shows the total time it takes to either serialize or de-serialize a byte array of a given size. Comparing this to the results in table 3.1, we can see that for data sizes below 5kb, time spent serializing is negligible compared to the time it takes to write the data to the store, this means that for many small records, serialization is considerably faster.

For larger data sizes let us look at an example: To calculate the estimated time it takes to store something serialized we need to multiply the serialization time with the number

Data Size (bytes)	Serialize	De-Serialize	Overhead S.	Overhead DS.
32	0,2	0,1	0,1	0,1
64	0,1	0,1	0,1	0,0
128	0,1	0,1	0,1	0,0
256	0,1	0,1	0,1	0,1
512	0,2	0,2	0,2	0,0
1024	0,5	0,2	0,3	0,1
5120	1,1	0,9	0,9	0,1
10240	1,8	1,6	1,6	0,1
20480	3,4	3,3	3,1	0,1
25600	3,9	3,7	3,9	0,1

Table 3.5: Benchmark results for serialization. S is short for serialization and DS for de-serialization. Times are in ms.

Data Size (bytes)	Total S.	Total DS.
32	0,3	0,2
64	0,1	0,1
128	0,2	0,1
256	0,2	0,2
512	0,4	0,2
1024	0,7	0,2
5120	2,0	0,9
10240	3,4	1,6
20480	6,5	3,4
25600	7,8	3,8

Table 3.6: Calculated total time for serialization. The values are the sum of (overhead S + serialize) or (overhead DS + de-serialize) from figure 3.5. Times are in ms.

of elements, add the overhead for serializing some data of that total size (returning the `ByteArrayOutputstream` as an array takes time) plus the time it takes to write the serialized data into the record store. Storing 20480 bytes of data consisting of four 5120 byte records would then take an estimated $(1,1ms * 4) + 3,1ms + 4,4ms = 11,9ms$ storing this as four records (without serialization) would take $2,6ms * 4 = 10,4ms$, compared to the obvious gains when serializing smaller arrays, it is slightly faster to store a few larger records than to use serialization. In other words, if we want to store some large pieces of data, it might be beneficial to write them as separate records, at least if there aren't many of them.

3.3.3 Conclusion

Based on the results from the benchmarks it is pretty clear that our hypothesis holds true. It can quickly become costly to store data in separate records when the number of records goes up, the amount of data that is being stored seems to matter less than the overhead for performing any given operation. Furthermore, the cost of serializing multiple data sets into a single larger array does not exceed the benefits of storing in single records. If the data sizes exceed approximately 5120 kb each, it might be more viable to store them as separate records, however, this comes down to the number of data pieces to store. It would be fair to say that in most cases it is better to store data in a single record than split into multiple ones. This also holds true for updating data, in this case however, the threshold for number of records to be updated separately before serialization being better is slightly higher.

4 Implementing secureXdata

4.1 Implementation Criteria and Challenges

Before starting any work, it is imperative that some goal or ground rules are laid down regarding what we are trying to accomplish. The ultimate goal of this work is obviously to create a working API implementation of the protocol discussed in section 2.5, this will be referred to as secureXdata or simply the API. There are a number of solutions to this problem however, so in order to be more precise, we determined some criteria for the implementation. These are loosely based on the talk "How to Design a Good API & Why it Matters" [28].

4.1.1 Ease of Use / Transparent Design

This is the most important of the criteria we have decided on. As mentioned in the introduction chapter 1 there are multiple MDCS in use today, many of which lack proper security 2.1. If we are to be successful in getting any of these to adopt the API into their applications, integrating the API into an existing system needs to be as hassle free as possible.

Trying to address this, we have focused on making the API design as transparent as possible. This means that a programmer using the API to the least possible degree needs to know anything about what's going on inside the API, ideally any programmer with Java ME experience should be able to use the API without having to read any documentation.

As we will discuss later in this chapter, it is not possible to extend the existing Java ME classes we're interested in. Instead we will mimic the way they work down to the method signatures. This makes integration into existing systems very easy, since the only changes needed to be done is one or two lines of initialization and changing the object type.

4.1.2 Functionality Decoupling and Flexibility

The full API provides more than one functionality, primarily there are three different services provided. Since each of these cover different areas (server/user authentication, secure storage, secure communication), they should be as decoupled as possible, preferably completely independent of each other.

This makes the API very flexible since a programmer can use only the aspects of the protocol that is needed, and not being forced to add other unwanted functionality. Further more, it is desirable to provide more than one way of using each of the parts or modules of the API this is to accommodate any different requirements applications might have.

4.1.3 Low-end Device Requirements

As discussed in 2.3.1 the primary target for the API is low end (in terms of memory and CPU) devices or feature phones [60], as such it is obvious that the API needs be designed to support such devices. On a strictly technical note, this would be using a Java ME configuration that the device is capable of running.

The other side of the matter would be to accommodate the varying capabilities of different devices in terms of hardware capabilities. Cryptographic algorithms are heavy processes and as such it should be possible to make trade offs between better security and better performance. This should of course have some lower limits to ensure at least a minimal level of security.

The user experience or usability also comes into the picture when working with low-end devices. Many devices have small screens and limited capabilities for inputting passwords etc, and this needs to be taken into consideration. How well the system performs also affects the usability. Long wait times can make users oppose securing their system, and heavy processing can be draining on the battery which in turn means the device needs to be charged often which while in the field might be a serious problem.

4.2 SecureXdata Architecture

Before going into detail on the different packages, this section will give an overview of the packages and the intended ways for integration, as well as some of the package dependencies. This should give a basic understanding of the API architecture, and how things fit together

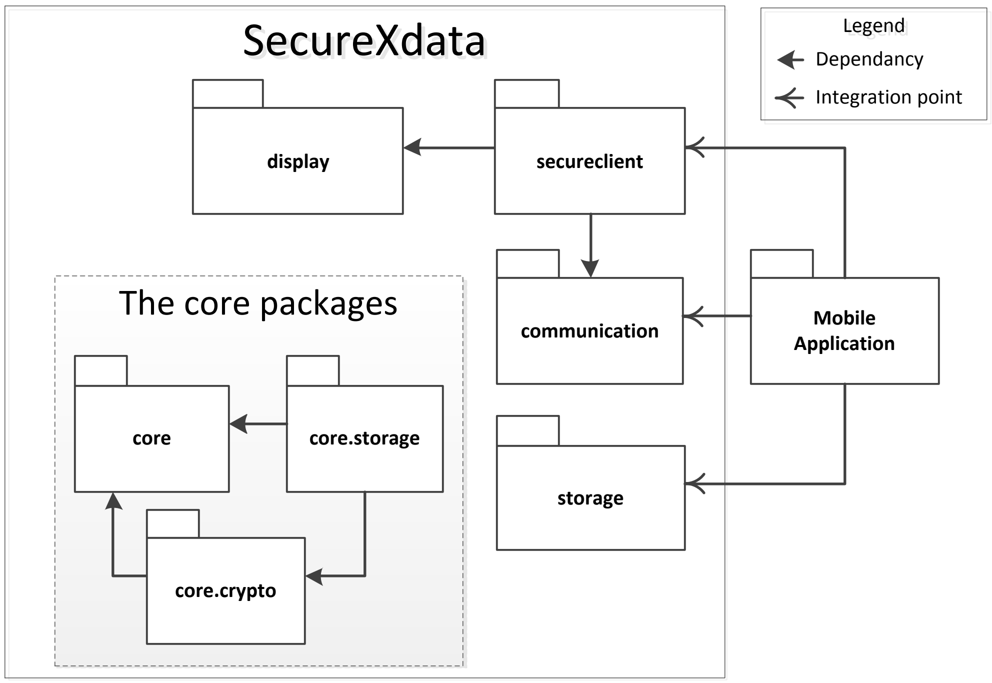


Figure 4.1: High level overview of package dependencies and API entry points.

before going more in depth later in the chapter. Figure 4.1 shows the high level API architecture.

One of the criteria for the design of the API is as described in 4.1.2 function decoupling, and as we can see from the dependency diagram 4.1 the API can be split into four primary parts. The `core` packages which are used by the entire system, the `communication` package, the `storage` package and the `secureclient` and `display` packages. In the following sections we will summarize the purpose of the different packages and see how much they depend on each other. There will be a more detailed description of the different parts later in this chapter.

4.2.1 Core packages

These packages contain the core functionalities of the API that are used by the other packages such as the `ApplicationStore` which contain the application settings. The core also contains support/helper classes for array manipulation or serialization.

4.2.1.1 The `core` Package

This is where all the support classes are, for instance the `Arrays` class which provides features similar to `java.util.Arrays` in Java SE. The custom exceptions thrown by the API are also defined in this package.

4.2.1.2 The `core.crypto` Package

When attempting to provide a secure system, cryptography is essential. As such most of the functionality provided by the API uses cryptography to some extent. This package contains the cryptographic classes used throughout the API, this provides flexibility with regards to how cryptography is implemented. This will be discussed in more detail in a later section.

4.2.1.3 The `core.storage` Package

This package contains the classes that relate to data storage. Classes of interest here are `ApplicationStore`, `UserKeyStore` and `SecurityPolicy`, which store application settings, user settings and security related settings respectively. Helper classes for interaction with the RMS system is also within this package.

4.2.2 The `secureclient` Package

Later we will described how the `AbsSecureClient` can be leveraged by a programmer to create a "secure gateway" into his or her application that will handle server authentication, user registration and logging in users.

The `securelient` can be considered a template or basic client that provides some common functionality though its abstract class, and as such it depends on most of the other packages. More specifically all but the `storage`.

4.2.3 The `communication` Package

The purpose of this package is to provide secure communication. It depends on `core.crypto` for cryptography and on the `ApplicationStore` for protocol related settings such as the ApplicationId. It is, however, completely independent from the `SecureClient` and the storage.

The main class of interest in this package is the `SecureHttpConnection` which provides the means to create a secure communication channel between the client and server.

4.2.4 The `storage` Package

This package provides facilities for storing data securely on the device. The classes of interest here are `SecureUserRecordStore` and `SecureRecordStore` which will be discussed in greater detail later in this chapter.

As with the `communication` package, the `storage` package depends on the core components for cryptography and application settings, but is independent from `communication` and `secureclient` as well. So if an application uses HTTPS for communication, it would be able to use `storage` to secure data saved locally without changing its transport layer.

4.2.5 Decoupling and Flexibility

Even though we have not yet gone into any details about how the different parts of the API function, we have seen that the main functionalities are decoupled and can be used as separate modules. In the next subsection we will look at three possible approaches that can be used to integrate an application with the API. These have little to do with and should not be confused with the three integration points as show in figure 4.1.

4.2.6 Integration Approaches

There are three main approaches for integration, these provide different degrees of control and responsibility for the programmer. That is, as with most other things, with more control comes greater responsibility. If the programmer is knowledgeable when it comes to security, he or she can control how keys are stored and managed. If the programmer is not familiar with security design then this task can be taken care of by the API. The approaches will go from programmer control to API control.

4.2.6.1 Manually Manage Keys

The `SecureRecordStore` and `SecureHttpConnection` can be initialized by the programmer directly, this means that the keys used will be managed by the program using the API and the API only provides secure transmission (ie. securing the communication from the device to the server) and secure storage, meaning the data stored on the device is secure. This is the most flexible way of using the API. The programmer has full control of how user credentials and data is stored and handled. However, this means that the programmer has to take care of keeping any keys used safe both from a potential attacker, and from being lost. This could be done by using the `SecureRecordStore` class provided by the API for instance.

4.2.6.2 Using The `UserKeyStore`

The API provides a user key store 4.5.2 which has the responsibility of handling user specific data and keys. The key store is protected by a password and identified by a user name, if the correct log in information is provided, the key store will be unlocked and the data made available to the program. This means that the programmer does not need to manage user keys and data, they will be stored in the `UserKeyStore`. Furthermore, once a `UserKeyStore` is unlocked, the `SecureUserRecordStore` 4.5.3.3 and `SecureHttpConnection` 4.6.2 are both initialized with data from the key store. As a result the programmer need only be concerned with one or two keys: a users password (the programmer needs to retrieve it from the user), and as a contingency should the users lose their passwords, backing up the key used by the `SecureUserRecordStore` (called the storage key, discussed in greater detail later) is advised. Failing to do so could result in not being able to decrypt data should a user forget his/her `UserKeyStore` password.

This approach takes care of most of the key management required, but still leaves the programmer with a decent amount of control. This approach, as well as the first one, requires no server interaction what so ever and could be used to secure local data on a offline application, for example. The last approach, however, relies on server connectivity.

4.2.6.3 Using the `SecureClient`

By extending the `AbsSecureClient` class an application can delegate the task of handling users entirely over to the API. Once the application starts, the screen control would be given to secureXdata by running the `SecureController` (Discussed later in section 4.3.2). Depending on the configuration, the controller may prompt users to authenticate the server on the first run, ask them to register or just log in. Once the registration or authentication is successful, control is given back to the application. At this point the UserKeyStore 4.5.2, `SecureUserRecordStore` section 4.5.3.3 and `SecureHttpConnection` section 4.6.2 have been initialized and are ready for use. This will be discussed in greater detail in the next section. By using this approach, the programmer is completely free from any security concerns. The API will take care of server authentication, user registration, account recovery, logging users in, and initializing the different components with the correct keys. This however comes at the cost of the programmer being confined to the system provided by the API, and having little to no control over how parts of the application works.

4.3 The `secureclient` Package

In order to more easily create a secure application, the API provides an abstract client that can be used as a starting point or template. It easily provides features like server authentication, user registration and log in facilities.

4.3.1 The **`AbsSecureClient`** Class

As discussed in the last of the integration approach sections 4.2.6.3, the abstract secure client can be leveraged by the programmer to take care of server authentication, user registration, account recovery and logging users in.

4.3.1.1 The **`userMenu()`** Method

One important thing to pay attention to when developing Java ME applications is the system thread, more specifically not to run your code in it. In short, the system thread won't be able to do its job, such as updating the current screen until it is not in use. This will be discussed in greater detail in chapter 6. Because of this, the `AbsSecureClient` has an abstract method called `userMenu()`, the purpose of this method is to give control back to the application once the API has logged a user in.

The logging in steps are run inside a number of threads, this will be discussed in the next section. Once these threads have logged in a user, they call the `userMenu()` method and the application can run its code. The `secureclient` threads will then wait until the application calls `logout()` at which point they will restart the log in process.

There would be nothing wrong with using a normal method instead of the `userMenu()` callback method. One such method could be `logInScreen()` for instance, which would return once the user has been logged in. However, the downside of this approach is that the programmer would be responsible for correctly running this method in a separate thread to avoid application deadlock. This would be a nuisance, goes against good API design as proposed by Josh Bloch [28] and would result in boilerplate code [20].

The following code listing shows how an application extending the `AbsSecureClient` would be implemented.

Listing 4.1: Example: Using the `AbsSecurClient`.

```java
protected void startApp() {
  try {
    // Initialize the SecureController
    initSecureModule(this,true,null,null);
  } catch (ProtocolException e) {
    e.printStackTrace();
  }
  // Hand over screen control, since this runs a
  // thread, it returns immediately.
  startSecureModule();
} // System thread exits the method and is now free
  // to do screen updates etc.

public void userMenu() {
  // A user is now logged in. Application code
  // goes in here.
}
```

As shown in the above listing 4.1 once the application starts, control is handed over to the API which performs the necessary steps to log a user in. Once the user authentication is successfully completed, `userMenu()` is called by the API and control is returned to the application. At this point the programmer can assume three things.

1. The user logged in is authenticated.
2. The users `UserKeyStore` is unlocked.
3. `SecureHttpConnection` and `SecureUserRecordStore` are initialized with the data for the logged in user and are ready to be used.

This means that the programmer can proceed as if making a normal Java ME application, and by using `SecureUserRecordStore` instead of `RecordStore` and `SecureHttpConnection` instead of `HttpConnection`. These secure classes have the same methods, but secure the data or communication behind the scenes.

4.3.2 The **`SecureController`** Class

The `SecureController` and its private class the `DisplayController` are what controls the screen and program flow for the `SecureClient`. These have both been a part of the system since before the candidate joined the project. As this is not something the candidate has made many changes to, these will not be discussed in great detail. We will however overview the basic design and more importantly the role it plays in the API. There will be a section on this class in the future work section 7.1 as well, since it is in need of updating.

The `SecureController` is in charge of running/controlling the `SecureClient`, that is, set the correct first screen depending on whether or not the server has been authenticated or there are any registered users. Once the initial screen has been set the `DisplayController` takes over, this is an inner class inside the `SecureController`. This takes care of screen flow within the `SecureClient`. Once a user is logged in, the `DisplayController` signals the `SecureController`, which in turn calls `userMenu()` in the mobile application and waits until a call to `logout()` is made.

Both these controllers are threads, and the command handler for any user input in the `SecureClient` is also run in a thread of its own. This is, as previously discussed, essential when it comes to making a responsive Java ME applications. If not done correctly, screen flow will be wrong at best, or the application deadlocked. Details regarding the system thread are discussed in chapter 6.

Even though the `SecureController` is running in a separate thread, the command handler code inside it will be run by the system thread upon user input. Since the command handler signals the `SecureController` to do most of the heavy work and it does not require any screens to be set to proceed, the application would not be deadlocked had the command handler not been in a thread of its own. However, as was the issue in the early stages of he API, any screen updates to give the user feedback of progress etc, would be queued until the command handler code returns, at which point all the screens will be set at once.

4.4 The **crypto** Package

As we discussed in section 4.1.3 the main Java ME platform does not provide any functionality for cryptography. Though, in some cases some cryptographic primitives might be available through optional libraries. However, this will vary from device to device. In this section we will discuss how to deal with this.

4.4.1 The **CryptoTools** Interface

Because of this lack or difference in functionality, anything related to cryptography needs to be very flexible to be able to accommodate any requirements or shortcomings a device might have. Through the `CryptoTools` interface the API tries to address this by enabling the developer to make their own implementation using whatever algorithms or libraries they have available. In other words, if there exists some implementation of the used cryptographic primitives that works on the device, the API will be able to use these. Some devices support native Java ME libraries based on the JSR177 [56] specifications, using these might be beneficial with regards to memory and CPU/battery usage compared to using third party libraries such as BouncyCastle [8]. Using native code will also help keep the binary sizes of the resulting application down.

Listing 4.2: Interface: Overview of The `CryptoTools` interface.

```java
public interface CryptoTools {
  /**
   * Returns a String describing which security provider is used
   * for the implementation. In this case either "BC" or "JSR"
   *
   * @return A string describing the provider used for
   *         implementing the cryptographic functions
   */
  public String getProvider();

  /**
   * Sets the public key used by asymmetric ciphers.
   */
    public void init(byte[] publicKey) throws ProtocolException;

  /**
   * Encrypts the given message with the stored public key.
```

```
 *
 * @param message The message to encrypt
 * @return The encrypted message
 */
public byte[] publicKeyEncryption(byte[] message)
    throws ProtocolException;

/**
 * Method used to add seed material to the random number generator
 *
 * @param seed The seed material
 */
public void seed(byte[] seed);


/**
 * Encrypts the message by using some symmetric encryption and
 * the given key.
 *
 * @param message The message to ecrypt
 * @param key The key for the encryption
 * @param iv The initialization vector if the cipher works with
 *       CBC mode
 * @return The message encrypted with the given key
 */
public byte[] symmetricEncryption(byte[] message, byte[] key,
                                  byte[] iv)
                  throws ProtocolException;

/**
 * Decrypts the given encrypted message with the given key.
 *
 * @param message The message to decrypt
 * @param key The key to use for the decryption
 * @param iv The initialization vector if the cipher works with
 *       CBC mode
 * @return The message in clear
 */
public byte[] symmetricDecryption(byte[] message, byte[] key,
                  byte[] iv)
                  throws ProtocolException;

/**
 * Generates a random byte array of given size, which can be
 * either 16 or 32 or 64.
 *
 * @param i The size of the array (key)
 * @return A random generated sequence of {@see i} bytes
 */
```

```
  public byte[] generateRandomKey(int i) throws ProtocolException;

  /**
   * Creates a digest of the input message based on the
   * implementation used. Most likely SHA1
   *
   * @param message - The message to create a digest of
   * @return - The digest
   */
  public byte[] digest(byte[] message) throws ProtocolException;

  /**
   * Function to compute a key given the password and the salt.
   * The implemented PBE scheme depends on the underlying
   * implementation.
   *
   * @param password the user password provided at the login
   * @param salt The random 16 bytes to append to the password in
   *      order to generate the key
   * @param iterations the number of iterations the PBE algorithm
   *      will use to generate the key
   * @return The encryption key
   */
  public byte[] pbe(String password, byte[] salt,int iterations)
    throws ProtocolException;

  /**
   * Method to produce a HMAC.
   * @param data - The text used to generate the digest
   * @param key - The key used to authenticate the digest
   * @return - The HMAC for data and key
   */
  public byte[] hMac(byte[] data,byte[] key);
}
```

All of the cryptographic operations done by the API use an implementation of this interface.

4.4.1.1 Bouncy Castle and the Default Implementation

The API comes with a default implementation of `CryptoTools`, this implementation uses BouncyCastle [8]. Since BouncyCastle is a third party API it should support any Java ME enabled phone with the capacity to load it. So the default implementation should work on any device. It also provides a large arsenal of cryptographic algorithms and primitives, making different implementations possible.

On the other hand, since it is not a native implementation there are down sides. Computational time and memory usage might be higher than what would be the case with a native implementation, this indirectly effects battery usage. This is because native code can be in a more powerful language or even hardware. Furthermore, the BouncyCastle library is huge (the version used in this project is roughly 1mb), and as such the memory footprint of the API will take a hit, obfuscation helps keeping the size down, but it will still be larger than if using native libraries. In order to use some of the features of Bouncy Castle obfuscation is required, which can add complexity to the testing and debugging aspect of things.

The default implementation uses, as described in the protocol paper [36]: RSA for public key encryption. AES in padded CBC mode with initializing vector (IV) for symmetric cryptography. SHA1 digest. HMAC based on SHA1 digest. Password Based Encryption based on PKCS#12. A random generator based on the `SecureRandomGenerator` class provided by Bouncy Castle. During the next phase of the development of the secureXdata system, these and other algorithms will be tested extensively on a number of different devices to get an overview of what works better on which devices and so on.

4.4.2 The `CryptoToolsFactory` Class

The `CryptoToolsFactory` is the means by which the API gets access to the cryptographic primitives it should use. Once initialized with a cryptographic provider (an implementation of `CryptoTools`) it will be accessible by the API through the `public static CryptoTools getCt()` method.

4.5 The `storage` Package

In this section we will look at how the API stores application settings and manages user keys. We will also explain how the API provided secure storage works.

4.5.1 Application store

The application store is, as the name suggests, the store object that contains information for the actual application. Unlike the `UserKeyStore` 4.5.2, the data in the Application Store contains mostly constant data once it has been created. This data is shared between all the users on the device and consists of among other things the public key of the secure server, the URL to said server and the security policy for the system.

4.5.1.1 The Java Application Descriptor (JAD) file

The JAD file [46] is used to describe a corresponding JAR [46] file and its content. The application manager on the device can use the JAR files content to check whether or not it is capable of running the application prior to loading the JAR. It is also possible to specify custom attributes, for instance to define configuration settings for the device in the JAD file. This will be relevant in the next section.

4.5.1.2 The `SecurityPolicy` Class

As one would expect, the `SecurityPolicy` contains various settings regarding the security of the system. These dictate things like how many iterations should be used in generating some keys within the API, if HTTPS should be used, the size of the AES keys and so on.

These values are read from the JAD file during the first run of the system, and is since persisted as a part of the `ApplicationStore`. When reading the values from file, some of the attributes are checked to ensure they are not below some minimum requirement, this is to ensure that even if with a badly configured JAD file, security isn't compromised.

4.5.1.3 Implementation of the `ApplicationStore`

The `ApplicationStore` is a static class. The reason for this is that its data is used by the majority of the other classes in the secureXdata system and as already mentioned, the data it contains is primarily static. When it comes to how the actual data is persisted in the RMS system there are two obvious alternatives. Each field in the Application Store can map to a record in the RMS system, this will be referred to as "Multiple records approach", the alternative is to serialize the entire object and store it in a single record in the RMS system. This will be referred to as the "Single record approach".

Server and JAD authentication Before discussing different ways of implementing the actual storage, we will look at the design of another important part of the `ApplicationStore`, the JAD hashing used as part of server authentication. As discussed in the one of the papers [36], some of the attributes in the JAD should be dynamically generated in order to customize the settings for each application instance/installation. This poses as a problem because the JAR file contains a manifest that is added at compile time where you can add attributes. This JAR can then be signed, and even if it is not, the JVM will check the values that are in the JAD file against the ones inside the JAR manifest. Meaning that we would have to recompile every time the attributes are changed/generated.

In the original protocol [34] a digest of the public key was calculated and used as a part of the server authentication. However, since the public key is inside the JAD file we can expand on this idea and calculate the digest of the whole JAD file, containing the public key and thus verify not only the key, but an arbitrary set of JAD properties that we want to make sure are not tampered with. The digest is computed on the server and sent as part of the server authentication response. [36] The client can then calculate the digest for the JAD present on the device, and compare this with the one from the server.

The hashing is done by concatenating a string consisting of the values we want to verify in a predefined order. The order is the same on client and server. This alone however leaves the system vulnerable to spoofing attacks, if we have the properties “a=1”,”b=2”, and “c=3” and we concatenate them in the same order, there is nothing stopping an attacker from leaving “b” and “c” blank, and setting “a=123”. To prevent this, we prefix the value with its length + ’;’. Any empty values would have the length of 0. The previous example would then yield the value “3;1230;0;” which would not be the same as “1;11;21;3”. The concatenated string is then hashed and the hash from the server compared to the hash from the client values.

Multiple Records This is the approach that was initially used as it is usually the easiest to implement. Each field in the class is stored in a separate record in the record store. This can easily be achieved by mapping the record id to a constant and use this to retrieve or set the value in the record store. This means that each field can be changed and stored independently in the RMS. The benefits of having a one to one relation between the stored data and the object is that it is very easy to persist the data, even if just one field is changed. On the other hand, storing the whole object or retrieving it means that a number of records need to be written or read.

Another issue with storing an object as multiple records has to do with data integrity. Since there is no support for transactions when working with the Java ME RMS system, you could end up with inconsistent data in the record store. This could happen if an application terminates while storing data to a `RecordStore`. For example, we are storing the state of an object that has 10 fields. The first 5 are written successfully, then the application terminates due to the battery being empty. When the application starts back up, there is no way for it to know that the stored object stat is not correct. To avoid this issue one would have to keep an update bit in the record store, set it to invalidate the data, and then unset it once done to signal the state being valid.

Single Record This approach is used in the current implementation of the secureXdata system. Instead of storing each field as a separate record, we serialize all the fields into a single byte array using a `DataOutputStream` and store this in a single record in the record store. When we read the data from the record store it is de-serialized by reversing the store operation and the fields are set to their respectable values. Unlike with the multiple record approach, this means that a single read or write operation needs to be performed when storing or retrieving the data. On the downside, persisting a single field change means that the whole object will be serialized and stored.

Whereas the multiple records approach might cause objects to be in an undefined or inconsistent state, this problem can not occur with this approach. Since we are only working with one record, once we begin updating it will either finish successfully or the record would be in a corrupt state, in which case an exception would be thrown.

Hybrid approach A third alternative is to combine the two into a hybrid design. If some of the fields are changed frequently, it can be beneficial to keep these as a separate record in the record store and use serialization to store the data that rarely changes in a single record. This way one can get the benefits of both approaches, though it adds to code complexity.

For the Application Store however this is not very relevant since its data seldom changes. It could however be beneficial for the `UserKeyStore` which we will discuss in the next section.

4.5.2 The `UserKeyStore` Class

As described above the Application store contains information used by the whole application, that is, non user specific information. The `UserKeyStore` class' main role in the secureXdata system is to protect and manage the credentials and information. By successfully unlocking a `UserKeyStore` a user is also authenticated since the store is encrypted using a user password based key.

4.5.2.1 Storage Key

As described in [37], it is beneficial to separate the password used on the server, the password used locally on the device and the actual key used to encrypt the data stored on the device (storage key).

The most obvious benefit from this is that if the device should become compromised, an attacker would gain nothing more than the information on the device. Given they manage to crack the local storage, the password they would gain is used only locally and thus the server is safe.

Secondly, by saving the storage key elsewhere, such as on a remote location during registration, we make it possible to recover any encrypted data on the device even if the local password should be lost.

A third effect of this separation is that it is possible to change the local password independently from any data that might be stored on the device. In other words, if a user changes his or her password, the data on the device does not have to be re-encrypted, we only need to encrypt the `UserKeystore` with the new password. See figure 4.2 below for a graphical representation.

4.5.2.2 Implementation of the `UserKeyStore`

As already mentioned, the `UserKeyStore` protects the user data, this protection however includes giving the rest of the system access to it in a secure manner. The `UserKeyStore` keeps the following data: storage key, user id, session key, seed, request number and salt. The user name can be derived from the store's name.

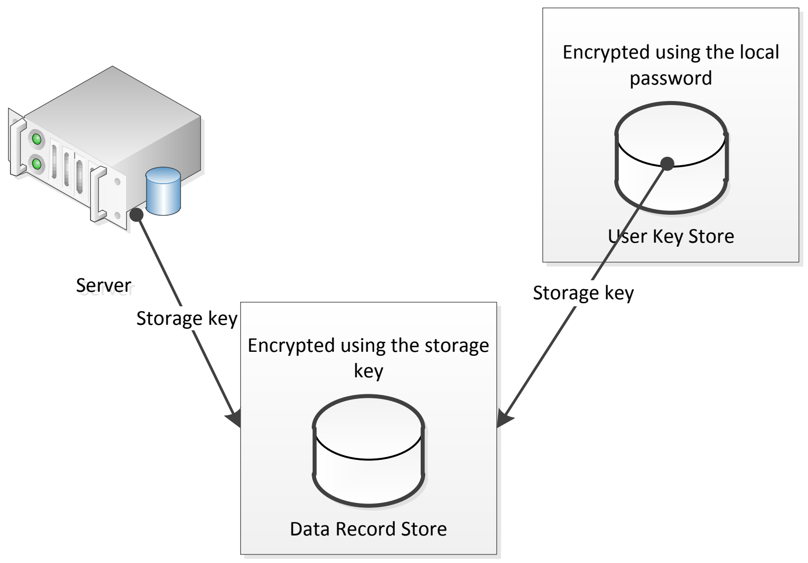

Encrypted using the local password
User Key Store
Server
Storage key
Storage key
Encrypted using the storage key
Data Record Store

Figure 4.2: Separation of data and storage key.

4.5.2.3 Usage Example

Before going into detail on the specific methods, it is advised the reader has a look at diagram 4.3 which illustrates a user logging in using the `UserKeyStore` and then writes some data to a `SecureUserKeyStore`. The user “Bill” is already registered in this example.

The `open()` Method The open method opens a underlying record store where the data will be stored. If the user does not exist and the create flag is set to false, an exception is thrown. Once the record store is opened, the `UserKeyStore` tries to read the encrypted data from it.

The `init()` Method Once a `UserKeyStore` has been opened for the first time, ie. no user data exists, it needs to be initialized. There are two methods for initializing a `UserKeyStore`, the first takes only a string, this string will be the password used to decrypt the key store. This method generates all the data that is needed (storage key, seed, number of iterations) encrypts the data using the provided password and stores it. Once done it cleans the object and returns the storage key.

The second method lets the programmer generate or otherwise decide on the storage key and seed used by the `UserKeyStore`. This can be useful for password resetting for instance, it is primarily provided as a part of the `SecureClient` user registration system.

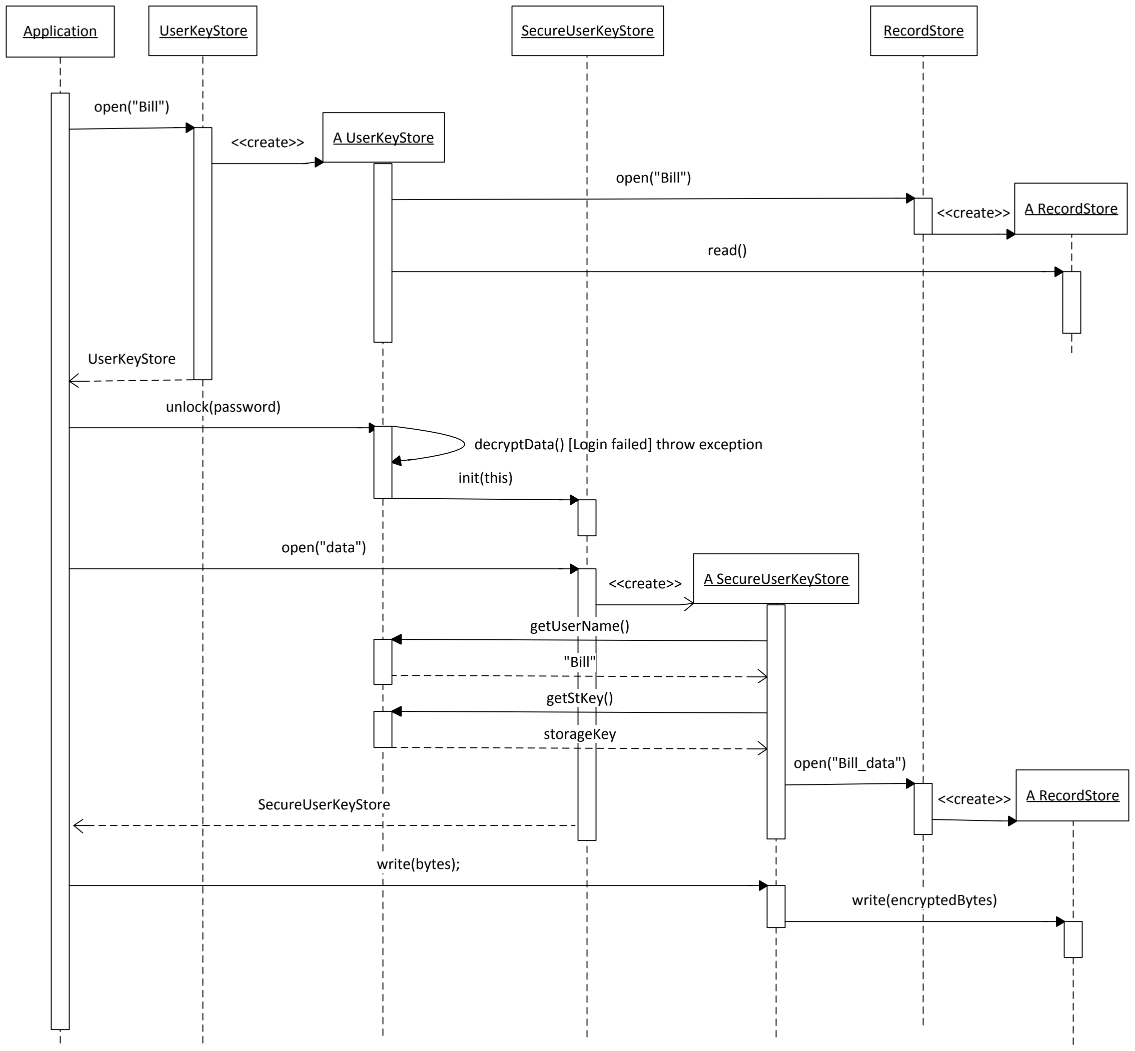


Figure 4.3: Sequence diagram showing how the `UserKeyStore` and `SecureUserRecordStore` is used to store some bytes for user Bill

The `decrypt()` Method When the `UserKeyStore` is saved, it persists encrypted data only, in other words, no plain text data is stored. When being opened, the stored data is read from the RMS and into memory. The `decrypt()` method decrypts the data and puts the plain text values into memory. Once there, they are accessible by the application through get methods. The process of decrypting the store unlocks the data but also acts as authentication since only the account owner should know the local password on the device.

If the decryption fails a number of times (by default 3), the `UserKeyStore` data is deleted permanently. In order to recover the account, the storage key must be set and secured with a password. This is essentially the `init()` being re-done. If the storage key used has not been backed up however, there is no way to recover the data encrypted in the `SecureUserRecordStore`. Initialization for recovery would use the second `init()` method that takes the storage key as a parameter.

The `clean()` Method To prevent other people from accessing the plain text values in memory, it is crucial that this data is removed once a user is logged out. This method sets all values in memory to null. This means that `open()` needs to be called followed by a successful decrypt in order to access the data again.

4.5.3 The `SecureRecordStore` Class

Whereas the `ApplicationStore` and the `UserKeyStore` are primarily used by the secureXdata system itself, containing information such as keys and security settings, the `SecureRecordStore` class provides the developer using the system with the means to secure the data used in the application in question.

4.5.3.1 Implementation of the `SecureRecordStore`

Extending the existing Java ME `RecordStore` class and adding security features to it would be the most obvious way to go about making this class. However, this is not possible due to the fact that the `RecordStore` class has no visible constructors, it is private.

Because of this we instead wrap the `SecureRecordStore` around the insecure `RecordStore`.

To be able to easily provide a number of alternatives when it comes to how the client application initializes the secure storage, most of the functionality is in an abstract class (`AbsSecureRecordStore`). The abstract class has one unimplemented method: `byte[] getKey()`. This method defines how the class gets the cryptographic key to use for encrypting and decrypting data going in and out.

4.5.3.2 Transparent Design and Easy Integration

As discussed in 4.1.1, one of the criteria for the design of the API was that it should be transparent and easy to integrate into existing systems. To accommodate this the

`SecureRecordStore` has been designed around the idea that it should be interchangeable with the normal `RecordStore`.

Mimics the RecordStore Even though it is not possible to make a class inherit from `RecordStore`, it is possible to mimic it by having the same method definitions in the `SecureRecordStore` as the normal one. By doing this we enable a programmer to interchange the API and Java ME version of the record store on the code level.

In chapter 7 we will discuss how we can make the relationship between the `RecordStore` and `SecureRecordStore` stronger.

Transparent Design Since the `RecordStore` class uses the singleton design pattern [31], the `SecureRecordStore` does the same. This opens a underlying `RecordStore` object that the secure object will write to and read data from. Code listing 4.3 shows the constructor. To the programmer the `SecureRecordStore` behaves the same as the normal `RecordStore`, and this conforms well with the first criteria 4.1.1. Internally however the data is encrypted before it is stored and decrypted before being returned to the application using the object. See code listing 4.4.

Apart from the read and write methods, nothing more than returning the corresponding value of the underlying `RecordStore` is done. See code listing 4.7 for clarification.

Code listing 4.5 and 4.6 shows a comparison of a code snippet using the normal `RecordStore` and the `SecureRecordStore`.

There is an additional method in the `SecureRecordStores` compared to the Java ME `RecordStore`: the `getRecordSizeQuick()` which returns the number of bytes stored in a record, that is, the size of the encrypted data. Since the data we are storing is encrypted, it might be in blocks. This depends on the `CryptoTools` implementation used, the block size could vary as well.

The default implementation 4.4.1.1 that comes with secureXdata uses a blockcipher and encrypts in blocks of 16 bytes [48]. This means that the cipher will only encrypt or decrypt 16 bytes at the time, if 6 bytes are given as input to the cipher and no more data exists, the remaining 10 bytes will be padded. If 16 bytes total are encrypted, a 16 byte padding block will be appended at the end.

This means that the data stored in the record store, might be of a different size than the number of bytes being written, but never less than the plain text data. So there would never arise a situation where the programmer expects to get more data than what would actually be returned when reading a record.

Since we are talking about, in most cases, a very small difference (maximum 16 bytes with the default implementation of `CryptoTools`) and since it can never be less than the actual data, it might make more sense to use the size of the encrypted data. This is because to get the actual byte size of the plain text data, it has to be decrypted, something that might be computational expensive.

The getRecordSize() method returns the correct number of bytes as expected, but as mentioned, this comes at the cost of having to decrypt the data.

Listing 4.3: Singleton constructor method for SecureRecordStore.

```java
public abstract class AbsSecureRecordStore {
  // ...

  /**
   * Superclass constructor for the abstract SecureRecordStore.
   *
   * @param store the underlying store that the data will be
   *        written to.
   */
  protected AbsSecureRecordStore(RecordStore store) {
    this.store = store;
  }

  // ...
}

public class SecureRecordStore extends AbsSecureRecordStore {
  // ...
  /**
   * Open a secure record store, internally uses the key that i
   * has been initialized with.
   * @param recordStoreName the name of the record store
   * @param createIfNecessary if true, creates the record store
   *        if needed.
   * @return an instance of SecureRecordStore
   */
  public static SecureRecordStore openRecordStore(String
    recordStoreName, boolean createIfNecessary) throws
      RecordStoreFullException,
      RecordStoreNotFoundException,
      RecordStoreException {
    return new SecureRecordStore(
        RecordStore.openRecordStore(
          recordStoreName, createIfNecessary
        )
      );
  }

  /**
   * Open a secure record store, internally uses the key that i
   * has been initialized with.
   * @param recordStoreName the name of the record store
   * @param createIfNecessary if true, creates the record store
   *        if needed.
   * @param authmode the mode under which to check or create access.
   *        (See RecordStore API).
   * @param writable true if the RecordStore is to be writable by
   *        other MIDlet suites that are granted access.
   * @return an instance of SecureRecordStore
   */
  public static SecureRecordStore openRecordStore(String
    recordStoreName, boolean createIfNecessary, int
    authmode, boolean writable) throws
      RecordStoreFullException,
      RecordStoreNotFoundException,
      RecordStoreException {
    return new SecureRecordStore(
        RecordStore.openRecordStore(
          recordStoreName, createIfNecessary,
          authmode, writable
        )
      );
  }

  /**
   * Open a secure record store, internally uses the key that i
   * has been initialized with.
   * @param recordStoreName the name of the record store.
   * @param vendorName the vendor of the owning MIDlet suite.
   * @param suiteName the name of the MIDlet suite.
   * @return an instance of SecureRecordStore
   */
  public static SecureRecordStore openRecordStore(String
    recordStoreName, String vendorName, String suiteName)
      throws RecordStoreNotFoundException,
      RecordStoreException {
    return new SecureRecordStore(
        RecordStore.openRecordStore(
          recordStoreName, vendorName, suiteName
        )
      );
  }
}
```

Listing 4.4: getRecord() and setRecord() internal workings

```
  /**
   * Encrypts the data and adds it to the underlying record store.
   *
   * @param data the data to be added
   * @param offset the index into the data of the first relevant
   *        byte to store
   * @param numBytes the number of bytes from the offset to
   *        include in the record
   * @return the id of the newly created record
   */
  public int addRecord(byte[] data, int offset, int numBytes)
      throws RecordStoreNotOpenException, RecordStoreFullException,
      RecordStoreException, ProtocolException {
    byte[] lBuffer = new byte[numBytes];
    System.arraycopy(data, 0, lBuffer, offset, numBytes);
    byte[] d = CryptoToolsFactory.getCt().symmetricEncryption(
      lBuffer, getKey(), new byte[16]);
    return store.addRecord(d, 0, d.length);
  }

  /**
   * Sets the content of a record to the passed in data.
   *
   * @param recordId the id of the record in the store that is
   *        to be updated
   * @param newData the array containing the new data to store
   * @param offset the index into the data of the first relevant
   *        byte to store
   * @param numBytes the number of bytes from the offset to
   *        include in the rec
   */
  public void setRecord(int recordId, byte[] newData,
              int offset, int numBytes)
      throws RecordStoreNotOpenException,
      InvalidRecordIDException, RecordStoreFullException,
      RecordStoreException, ProtocolException {
    byte[] lBuffer = new byte[numBytes];
    System.arraycopy(newData, 0, lBuffer, offset, numBytes);
    byte[] d = CryptoToolsFactory.getCt().symmetricEncryption(
      lBuffer, getKey(), new byte[16]);
    Logger.write(Arrays.toString(d),doLog );
    store.setRecord(recordId, d, 0, d.length);
  }
```

Listing 4.5: Example: SecureRecordStore (Transparent design)

```
// Uses the secure class
SecureRecordStore rs = SecureRecordStore.openRecordStore("test_store",
    true);
if (rs.getNumRecords() == 0) {
  byte[] b = "Data to store recordstore".getBytes();
  rs.addRecord(b, 0, b.length);
  print("Stored data..");
}
print("Reading data:");
print(new String(rs.getRecord(1)));
rs.closeRecordStore();
```

Listing 4.6: Example: Normal Java ME `RecordStore` (not secure)

```
// Uses the Java ME class
RecordStore rs = RecordStore.openRecordStore("test_store", true);
if (rs.getNumRecords() == 0) {
  byte[] b = "Data to store recordstore".getBytes();
  rs.addRecord(b, 0, b.length);
  print("Stored data..");
}
print("Reading data:");
print(new String(rs.getRecord(1)));
rs.closeRecordStore();
```

Listing 4.7: Methods return the corresponding underlying values.

```
    // Store is the underlying RecordStore.
  public long getLastModified() throws RecordStoreNotOpenException {
    return store.getLastModified();
  }

  public String getName() throws RecordStoreNotOpenException {
    return store.getName();
  }

  public int getNextRecordID() throws
    RecordStoreNotOpenException, RecordStoreException {
    return store.getNextRecordID();
  }
```

Future Work: Descriptions for SecureRecordStore entries When data is stored using the`SecureRecordStore`, the data needs to be decrypted in order to give get any meaningful information. Cryptographic operations are generally costly (see chapter 5 for benchmarks). It is not unrealistic to assume that an application using the API would want to display a list of records for the users to pick from, this means that all the records need to be decrypted just to inform the user of what they contain.

In order to make it less costly for the application to display such a list, some form of description should be added for the records. Structurally it would make the most sense to store all of these in a single (the first) record in a given record store in a serialized format. Since it should be fair to assume that if such descriptions would be shown, it would most likely not be shown by itself, rather the descriptions of all records in a record store would most likely be shown together in a list. To make it secure the description record could be encrypted in the same way as the rest of the record store.

This would mean that one or more additional methods would be added to the secure record store classes. These could be for instance `public String[] getDescriotions()` which would return all the descriptions, or `public String getDescription(int id)` which would return the description for a single record.

A way of setting the description can be more difficult to add. Since most of the current API is designed around mimicking the Java ME classes we don't want to break this by changing the existing methods. Making an overloaded write/update method for a record store which takes an additional description parameter should not be a problem.

However, this would mean that changes would have to be made in any existing MDCS. To make such an integration easier, we propose that an interface can be added and integrated by the programmer. The record store could be initialized with an instance of this interface and through it extrapolate a description from any written using the existing `addRecord()` and `setRecord()` methods.

The interface could look something like the following listing:

Listing 4.8: Description extractor interface

```java
public interface DescriptionExtractor {
  /**
   * Extracts a description based on the given byte array.
   *
   * @return the description.
   */
   public String makeDescription(byte[] data);
}
```

This way any current applications don't need to change in order to get a description for their data. An implementation could be as easy as taking the first 20 bytes of the data and storing it as a string.

4.5.3.3 The `SecureUserRecordStore` Class

As described above, the `SecureRecordStore` encrypts the data before storing it on the device, this provides the programmer the means to secure the data from being read or tampered with by anyone knowing the key used to store it.

Initially, the `UserKeyStore` initialized the `SecureRecordStore` with a storage key upon user log in, this however obviously caused issues when more than one user comes into the picture. Since user A does not have the same storage key as user B, reading any data written while a different user was logged in, would result in a cipher exception. This situation occurs because the intended method of integration was to replace the existing `RecordStore` with the `SecureRecordStore` when securing existing MDCS. The database name is therefore the same for both users. The solution to this as far as the `SecureRecordStore` is concerned is to not initialize it when a user logs in. This however means the programmer would have to keep track of the different users' stores which in itself is an issue, since security related responsibilities would be put on the programmer. Furthermore, this would mean that to integrate security into an existing application a lot of changes would have to be made.

Both of these issues go against the design criteria for the API. To try and remedy this, we introduce the `SecureUserRecordStore`. The SecureUserRecordStore works in the same way as the `SecureRecordStore`. The methods and method signatures are the same and with the exception to the open method, they do the same. The open method however adds a prefix to the record store name provided. This prefix is the user name of the currently logged in user (this means that `SecureUserRecordStore` won't work without a user being logged in.) This means that in a way, each user has their own RMS system or storage environment.

If the programmer calls `SecureUserRecordStore.openRecordStore("data",true);` while user B is logged in, the actual record store name would be "B_data". In normal database terms, this could be seen as each user having their own database. Even though all the queries are done on tables with the same names, the data is different. This gives us three things: first, a users data can be encrypted with said users storage key and not cause any trouble. Secondly, because this happens behind the scenes, integrating with an existing application would only be a problem if you want to be able to share data among users. Lastly, this means the `SecureUserRecordStore` performs authorization on the data, only the correct user has access to his or her data.

4.5.3.4 The `SecureRecordEnumeration` Class

A `RecordStore` can be iterated with a `RecordEnumeration`, this would however return the data as it is stored, in other words the encrypted data. To remedy this the `SecureRecordEnumeration` was created, it works the same way as the normal enumerator, however any calls to the `nextRecord()` method will return the plain-text stored in the record as expected.

Since decrypting the stored data can be processor intensive, it is generally a good idea to refrain from enumerating the `SecureRecordStore` unnecessarily. See the Implementation Evaluation chapter 5 and Future Work section 7.1 for details on speed benchmarks and alternative solutions.

4.6 The `communication` Package

We've looked at how we can secure the data stored on the device, and how the programmer can leverage the `SecureClient` to register and log users in. In this section we will discuss how to provide secure communication in a easy and flexible way.

4.6.1 The **`SecureOutStream`** Class

This class encrypts anything written to it prior to writing the data into a `OutputStream` given in the constructor. It also takes care of appending the correct IV and a HMAC to the output so that the data can be successfully decrypted and its integrity verified once received by the server.

4.6.1.1 The **`write()`** Method

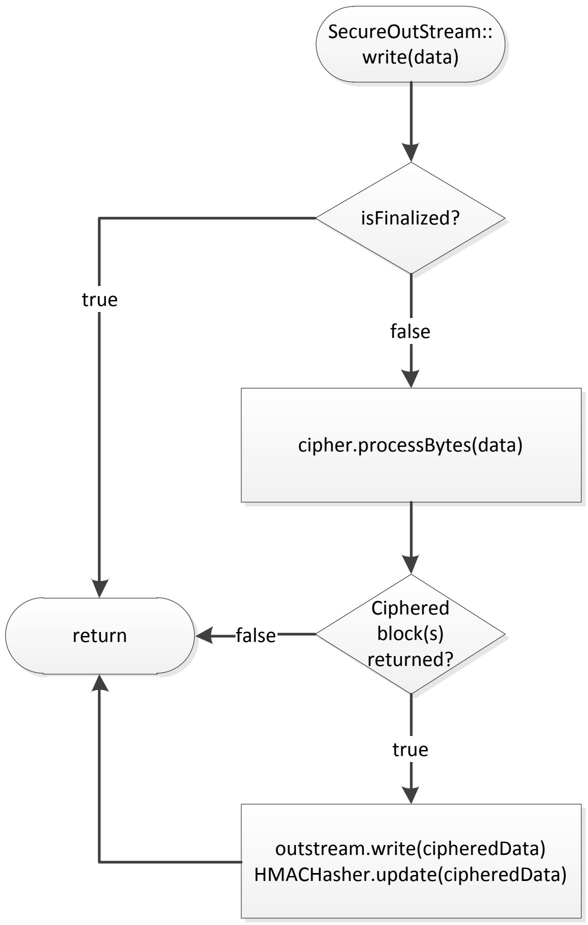


Figure 4.4: Inner workings of `SecureOutStream.write()`.

As we can see in 4.4 a few steps are taken before any data is actually written to the underlying stream. First we need to check if the stream is finalized, more on this in the next paragraph. If it isn't finalized, we pass the data to a cipher object. Depending on what type of cipher this is, it may or may not return any ciphered bytes. As previously mentioned, in our default implementation, a `PaddedBufferedBlockCipher` [48] is used. Whenever any data is added to the cipher, it will check if it has enough bytes to create a complete block (16 bytes). If this is the case, it will return as many ciphered blocks as possible. If not, it will add the new data to the buffer and wait for more until it can complete at least one block. When a block is returned, this is written to the out stream and the HMAC [22] is updated with the additional blocks.

4.6.1.2 The `flush()` and `finish()` Methods

Once we are done writing data to the `SecureOutStream`, it needs to be flushed correctly. The cipher and HMAC makes this more complicated than just flushing the buffer.

The first step is making sure any data written to the cipher is flushed out and that the ciphered data is complete so it is possible to decrypt it at some later point. This is done using the `doFinal(..)` method. What this does is to pad any data in the cipher-buffer so that it can create a full block. This block is then returned, and as with `write(..)`, written to the underlying stream and used to update the HMAC.

The next step is to append the HMAC hash at the end of the stream, this is done by calling `doFinal(..)` on the HMAC object. This returns a hash of all the ciphered bytes that have been written into the stream. This hash is written to the end of the `OutputStream` and it is flushed. The `SecureOutStream` object is now considered finalized. This is because we have sent the final cipher block and the HMAC of the data and as a result no further writing is possible.

As just described, the flushing of this stream is not without consequence. Once it has been flushed, no more data can be written. Because of this, and to avoid any inadvertent flushing, the `finish()` method takes care of flushing.

On some devices, flushing the stream will close it. This is at least the case on the Nokia 2330c [47], and as such, the `flush()` method has been changed to do nothing. Instead, if one wishes to flush the underlying buffer there is a `forceFlush()` method available to explicitly do so without finalizing the stream.

4.6.2 The `SecureHttpConnection` Class

The `SecureOutStream` provides the means to stream encrypted data to some destination, in this section we will discuss how we can, among other things, use this to create a class for securing HTTP communication between the client and server.

4.6.2.1 Common Issues

Before discussing different ways of designing a secure HTTP class based on the protocol, we will look at some common issues which we will encounter regardless of what design we choose.

As discussed in [38], the integrity of the data sent using the protocol is maintained by computing the HMAC [22] of the data received and making sure this is identical to the HMAC calculated by the server prior to sending. This means that we need to read all the data in the response before we can verify its integrity. In other words, even though the cipher used might support decrypting the data in chunks, we have no way of verifying it until it has all been read.

This can be problematic while working on phones with restricted memory and communicating large amounts of data. In order to address this, it would be possible to split the data into multiple requests and verify one piece at a time. This will be discussed in chapter 7 Conclusion and Future Work.

4.6.2.2 Different Approaches

The papers (section 1.3) on which this work is based, gives the grounds for the actual protocol used to secure the communication between client and server. However, how to implement this functionality and make it available to the programmer is yet to be discussed. In this section we will look at different approaches to making the protocol available to a programmer and look at how the current API implementation does this.

As a Static library One approach to this problem is to provide the bits and pieces needed to use the protocol. This means that the programmer using the API would use a normal HTTP connection, and through the API create the correct headers or chunks of data to send to the server in order to secure the communication. This is best illustrated by example.

Listing 4.9: Example: SecureHttp as a static library

```
// Get the HttpConnection object and set it to a post request.
c = (HttpConnection)Connector.open(url);
c.setRequestMethod(HttpConnection.POST);
// Get the output stream
os = c.openOutputStream();
// Write the protocol header.
os.write(SecureHTTP.getHeader());
// Write the desired data.
os.write(SecureHTTP.encapsulateData(data));
// Write the HMAC to the stream, after this point no further
// Calls to SecureHTTP.encapsulateData() would be possible.
os.write(SecureHTTP.getHMAC());
// Normal HttpConnection checking of response code and getting
// the response data, the read data will be in the array respData.
rc = c.getResponseCode();
if (rc != HttpConnection.HTTP_OK) {
  throw new IOException("HTTP response code: " + rc);
}
is = c.openInputStream();
(...)
if (SecureHTTP.validHMAC(respData)) {
  // Extract the actual response data and process it
  byte[] respDataPT = SecureHTTP.decaspulate(respData);
  process(respDataPT);
}
```

Pros

The programmer has full control of the code and when what is done.
The API enforces no restrictions, there are no limitations to what other items might be put in the stream.
The programmer could design his or her own systems for uploading and downloading data to the server.

Cons

It is up to the programmer to make sure the protocol specifications are followed. (Header needs to be written first, followed by the data, and the HMAC appended at the end.)
Programmer needs to understand security concerns.
Easy to make mistakes.

API Controlled At the other end of the spectrum we can let the API handle the entire process of securing and performing the communication. The API would more or less integrate the above code listing 4.9 and make it available as a method. The programmer would then supply a URL, and the request to send. The API would then do everything and return the response data when the transaction is complete. See the below code listing.

Listing 4.10: Example: API controlled `SecureHttp`

```
SecureHTTPRequest req = new SecureHTTPRequest(url);
req.setProperty("Content-Language", "en-US");
SecureHTTPResponse resp = SecureHTTP.fetch(url,data);
if (resp.isValid()) {
  process(resp);
}
```

Pros

The programmer needs no knowledge about the underlying security concerns.
Very easy to use, retrieving the content of a URL takes only four lines of code.

Cons

The programmer has no control of what goes on while the data is being fetched, this means that giving feedback to the end user would be difficult.
Integration into existing systems could prove difficult or require a lot of changes.

Transparent Design A third approach to the problem is to make a compromise between the two previous solutions. As we did with the `SecureRecordStore` 4.5.3, we can try and mimic the way the Java ME `HttpConnection` works, only our implementation secures the data streams using the `SecureOutStream` described in 4.6.1. This means that the programmer would use the `SecureHttpConnection` in more or less the same way as the default Java ME `HttpConnection`. The only difference would be, as discussed in the `SecureOutStream` 4.6.1, to ensure no issues with flushing the `SecureOutStream` the `finish()` method needs to be used to finalize and flush the stream.

This would only be a problem if the programmer wishes to explicitly flush the stream, in which case the `forceFlush()` would have to be used. However, since one would not normally explicitly flush, but rather use a method to proceed to the next "stage" of the communication this, should not be an issue. Such methods are `getResponseCode()` and `openInputStream()` (see [49] for details).

The example code listings below shows the difference between standard Java ME code and the secure code using `SecureHttpConnection`, line 3 in 4.11 is the integration point.

Listing 4.11: Example: `SecureHttpConnection` (Transparent design)

```
 1 HttpConnection hc = (HttpConnection)Connector.open(url);
 2 // Make the connection secure.
 3 SecureHttpConnection c = new SecureHttpConnection(hc);
 4 c.setRequestMethod(HttpConnection.POST);
 5 os = c.openOutputStream();
 6 os.write(someData);
 7 rc = c.getResponseCode();
 8 if (rc != HttpConnection.HTTP_OK) {
 9   throw new IOException("HTTP response code: " + rc);
10 }
11 is = c.openInputStream();
12 byte[] data;
13 // Reads the available data into the array.
14 readData(is,data);
15 process(data):
```

Listing 4.12: Example: Normal Java ME `HttpConnection` (not secure)

```
HttpConnection c = (HttpConnection)Connector.open(url);
c.setRequestMethod(HttpConnection.POST);
os = c.openOutputStream();
os.write(someData);
rc = c.getResponseCode();
```

```
if (rc != HttpConnection.HTTP_OK) {
  throw new IOException("HTTP response code: " + rc);
}
is = c.openInputStream();
byte[] data;
// Reads the available data into the array.
readData(is,data);
process(data):
```

Pros

Very easy to integrate into existing systems.
Programmer needs no knowledge about underlying security concerns.
The API puts no restrictions on the programmer compared to normal Java ME.

Cons

Flushing the `OutputStream` requires special attention if done directly.

4.6.2.3 Selected Approach

Based on the criteria discussed in 4.1, we decided that the best approach to this problem is the transparent design solution. In the following section we will look closer at some key aspects of the implemented SecureHttpConnection class.

Sessions As described in one of the the protocol papers [38], both symmetric and asymmetric encryption is used to secure the communication with the server. The asymmetric part is needed to initially verify the server and to securely transmit the session key used in the current communication. Once this communication is established we no longer need to include the asymmetrically encrypted header information and instead we use a normal HTTP session [42] and encrypt using a symmetric cipher and the session key. By doing so we save data overhead from the secure header, but more importantly, we save computational time since we no longer need to do costly asymmetric encryption.

Figures 4.5 and 4.6 show the differences between the HTTP request sent to the server with and without an active session.

The `openOutputStream()` Method This method sets up and opens the `SecureOutputStream` used to write the request data to the server. See figure 4.7 for an overview of the inner workings.

When the method is called, if an output stream has been opened already, the same object is returned. If not, the request method of the underlying HttpConnection is set to post and an IV is generated. If there exists a valid session, this is set to the cookie of the underlying `HttpConnection`. Else a new session key is generated and the API header is

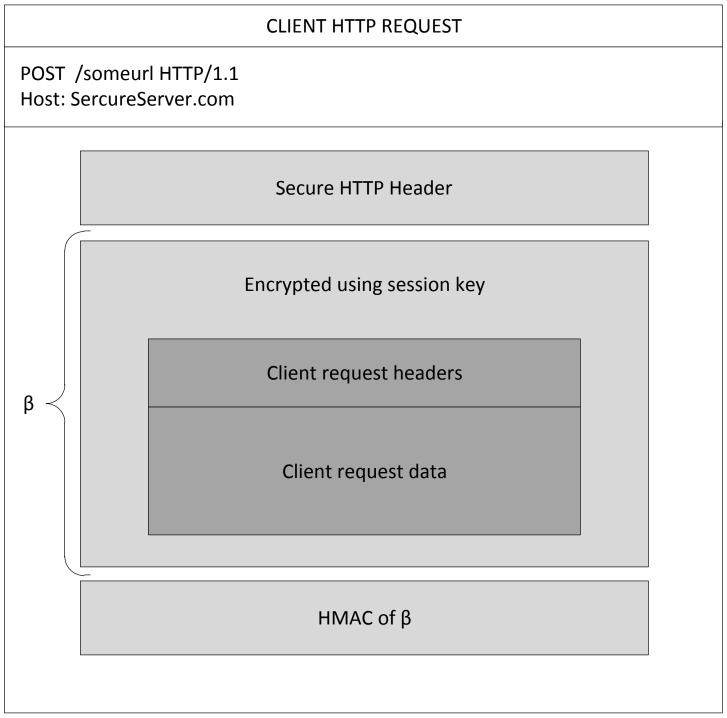


Figure 4.5: API secured HTTP request without session

added instead, see figures 4.5 and 4.6 for details. This means that it is the API meaning the client, that establishes the session. Even if a session already exists, it is entirely possible for the client to establish a new session. Figure 4.8 shows the actual byte stream structure in either case. The IV is written to the underlying `HttpConnection` output stream (in plain text) and a `SecureOutputStream` is created wrapping the said stream. Using a `DataOutputStream` any properties set in the `SecureHttpConnection` is written to the secure stream and the secure stream is returned.

The `openInputStream()` Method This method returns a `ByteArrayInputStream` containing all the response data sent from the server. As mentioned in 4.6.2.1 to be able to check the HMAC of the read data and thereby verify its integrity, we need to read all the data from the input stream and this is why we return a `ByteArrayInputStream`. Figure 4.10 shows the inner workings of this method. When the method is called, we check if the input stream has already been opened, in which case we return the existing stream. If not we must make sure the ouput stream has been properly flushed to the server. If no output stream has been opened we open one and finalize it. Otherwise we make sure it has been finalized. This conforms to the state transitions as described in the `HttpConnection` documentation [49]. We can now open the `HttpConnections` input stream and read the data. If the status code is not 200 an exception is thrown.

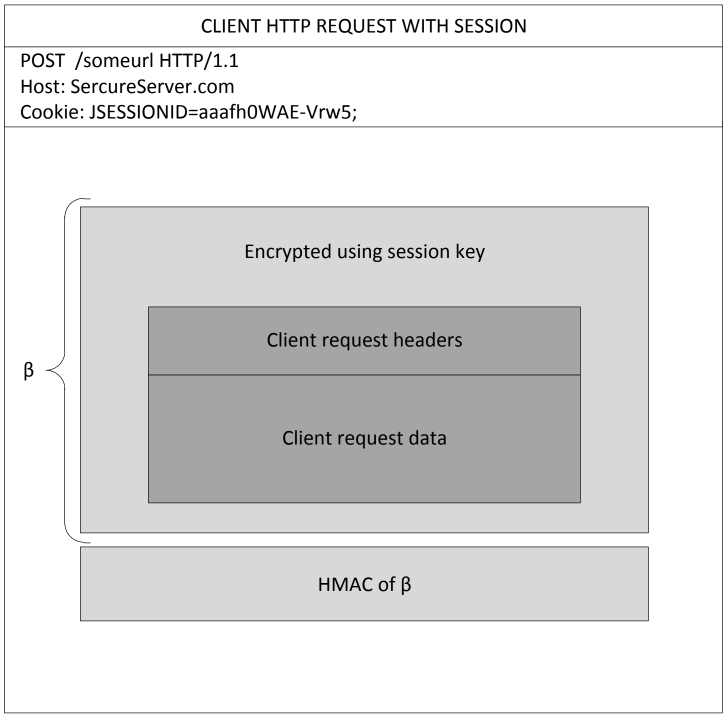


Figure 4.6: API secured HTTP request with session

Once all the data has been read we split it according to the protocol specifications as seen in figure 4.8. We can then verify the integrity of the data by calculating the HMAC of what we received and compare it to the HMAC provided by the server. If they do not match the data has been corrupted and an exception is thrown.

If no exception occurred, the The data is decrypted. Depending on whether or not a session exists for the current communication, the decrypted response may contain some session details from the server. This needs to be extracted before we can return the actual response, figure 4.9 shows the details. Once any session information is taken out, the remaining bytes comprise the plaintext response from the server. Since all the data is already read we return a `ByteArrayInputStream`.

4.7 The `display` Package

The `Display` package contains the different screens and displays used by the `SecureClient`. Getting user information and displaying the different steps. Not much interesting goes on in this package and as such it will not be discussed to any greater extent than some screenshots of the different steps when logging in to the application.

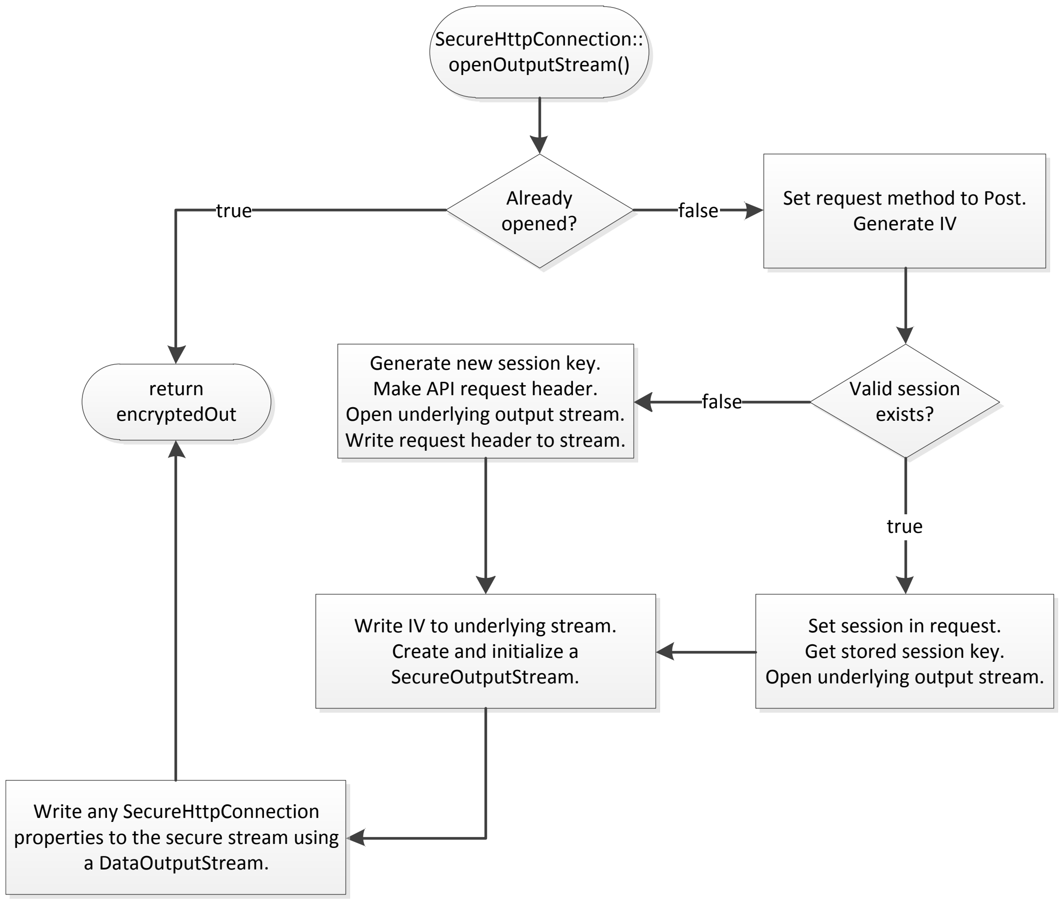

SecureHttpConnection::
openOutputStream()
Already opened?
true
false
Set request method to Post.
Generate IV
Valid session exists?
false
Generate new session key.
Make API request header.
Open underlying output stream.
Write request header to stream.
return encryptedOut
true
Write IV to underlying stream.
Create and initialize a SecureOutputStream.
Set session in request.
Get stored session key.
Open underlying output stream.
Write any SecureHttpConnection properties to the secure stream using a DataOutputStream.

Figure 4.7: Inner workings of the `openOutputStream()` method.

4.7.1 Screen Flow

The screen and program flow between the different screens is managed by the already discussed `SecureController` 4.3.2.

Figure 4.11 shows the flow between the different screens depending on user actions. The actual screens corresponding to the different stages can be seen in the screenshots below the figure.

We do note however that there is a “WaitingScreen” as part of the screens in this package, and in order to get this to work properly special care needs to be taken with the system thread. This is discussed in chapter 6.

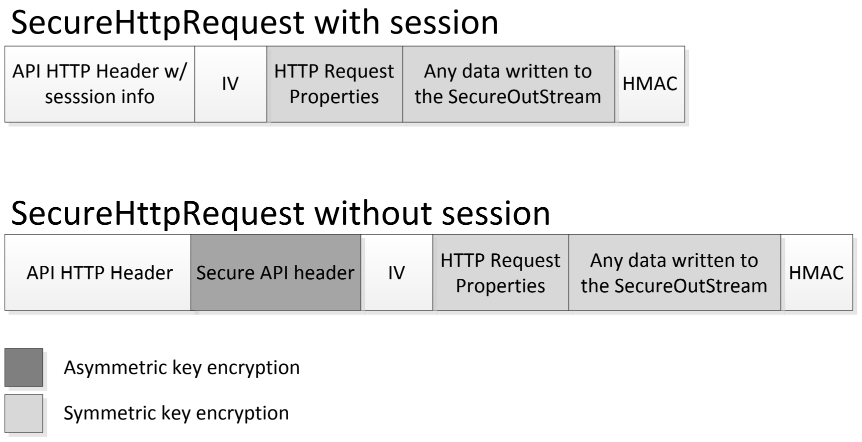

SecureHttpRequest with session
API HTTP Header w/ sesssion info
IV
HTTP Request Properties
Any data written to the SecureOutStream
HMAC
SecureHttpRequest without session
API HTTP Header
Secure API header
IV
HTTP Request Properties
Any data written to the SecureOutStream
HMAC
Asymmetric key encryption
Symmetric key encryption

Figure 4.8: Content/structure of the byte `SecureHttp` output stream.

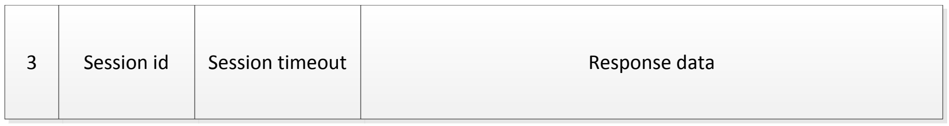


Figure 4.9: Decrypted response data with session data.

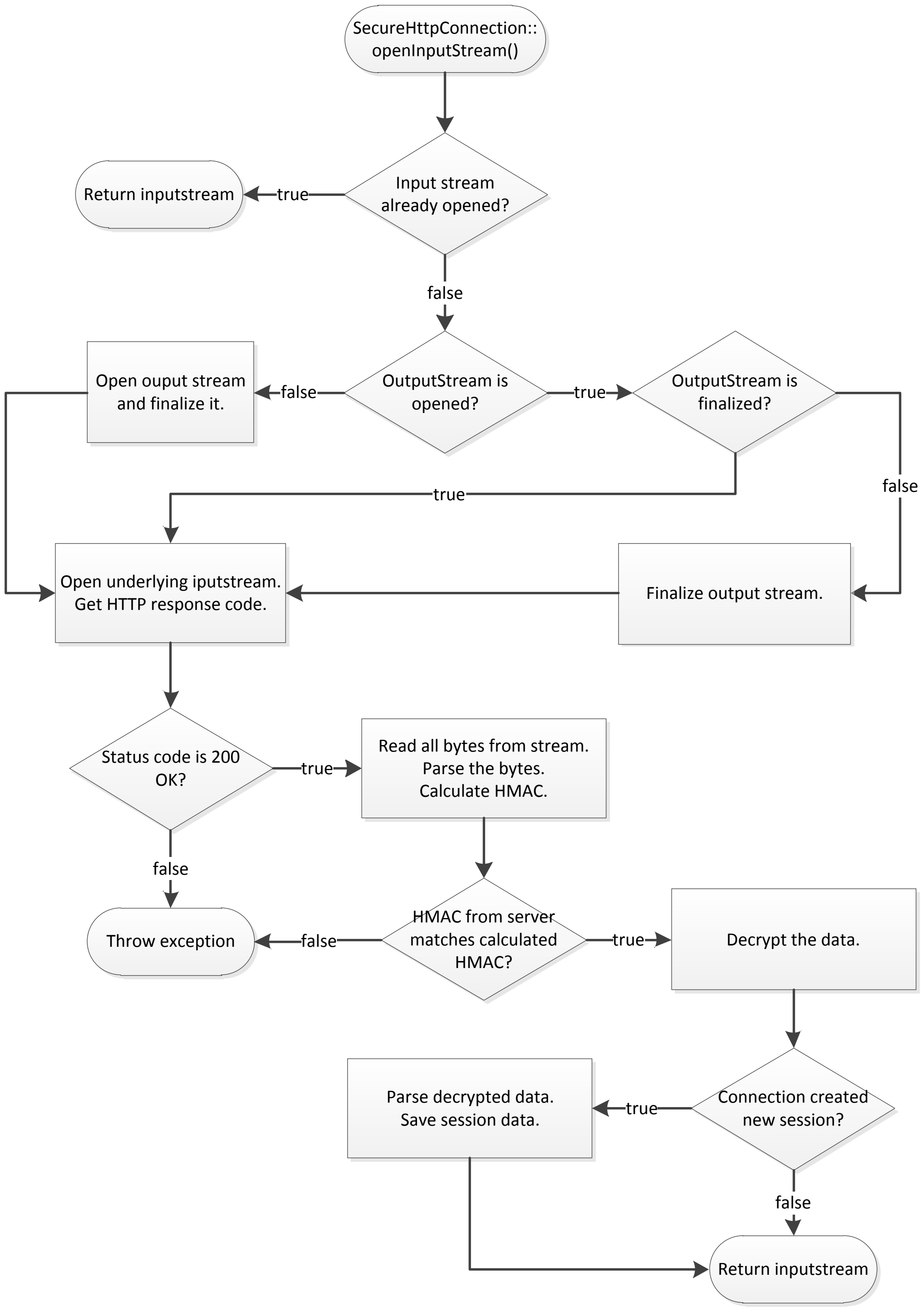

SecureHttpConnection::
openInputStream()
Input stream already opened?
true
Return inputstream
false
OutputStream is opened?
false
Open ouput stream and finalize it.
true
OutputStream is finalized?
true
false
Open underlying iputstream. Get HTTP response code.
Finalize output stream.
Status code is 200 OK?
true
Read all bytes from stream. Parse the bytes. Calculate HMAC.
false
Throw exception
HMAC from server matches calculated HMAC?
false
true
Decrypt the data.
Connection created new session?
true
Parse decrypted data. Save session data.
false
Return inputstream

Figure 4.10: Inner workings of the openInputStream() method.

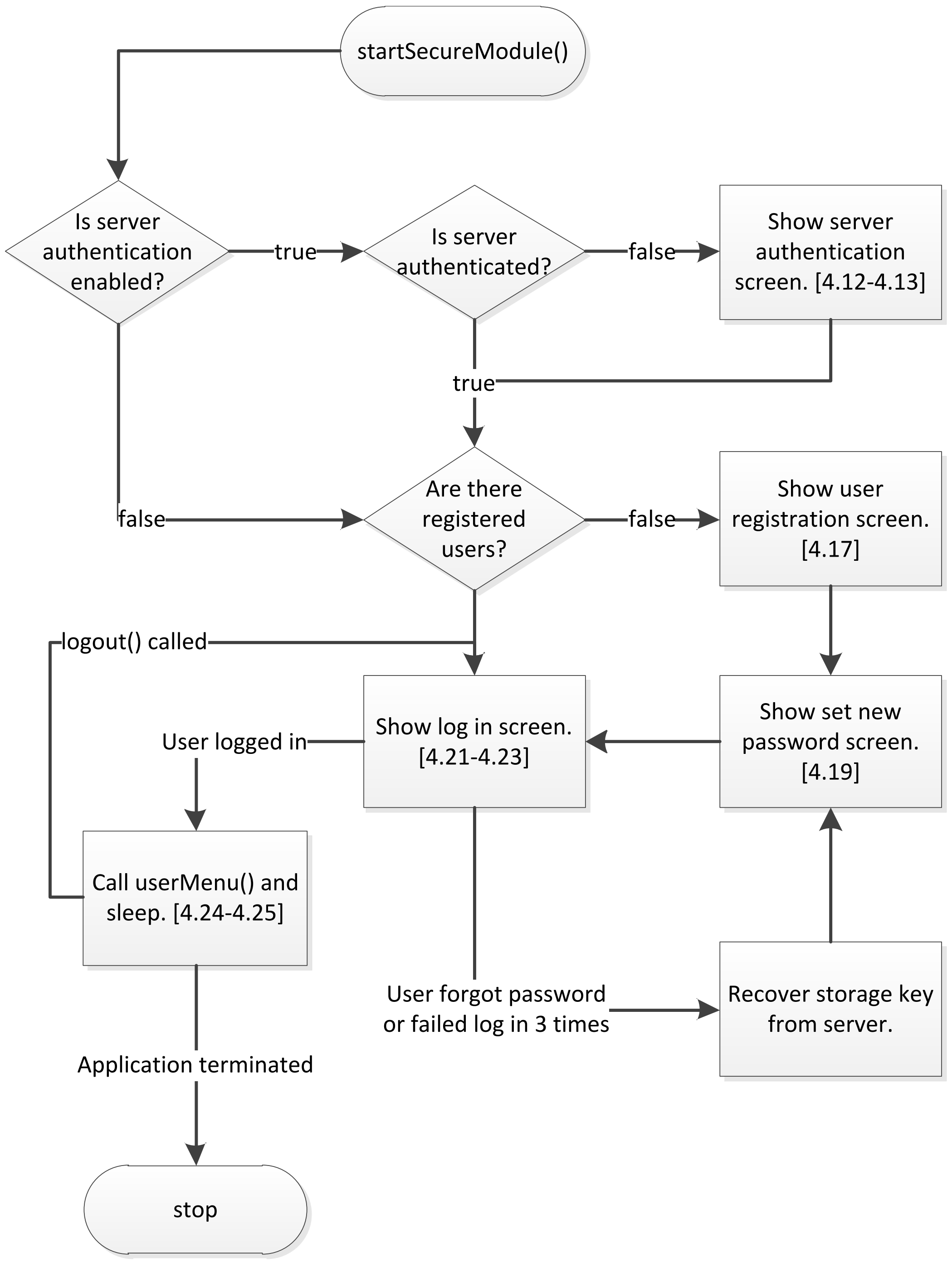

startSecureModule()
Is server authentication enabled?
true
Is server authenticated?
false
Show server authentication screen. [4.12-4.13]
true
false
Are there registered users?
false
Show user registration screen. [4.17]
logout() called
User logged in
Show log in screen. [4.21-4.23]
Show set new password screen. [4.19]
Call userMenu() and sleep. [4.24-4.25]
User forgot password or failed log in 3 times
Recover storage key from server.
Application terminated
stop

Figure 4.11: Screen flow in the application. The screenshots below show the different screens, the numbers in the diagram are the figure numbers for the corresponding screen.

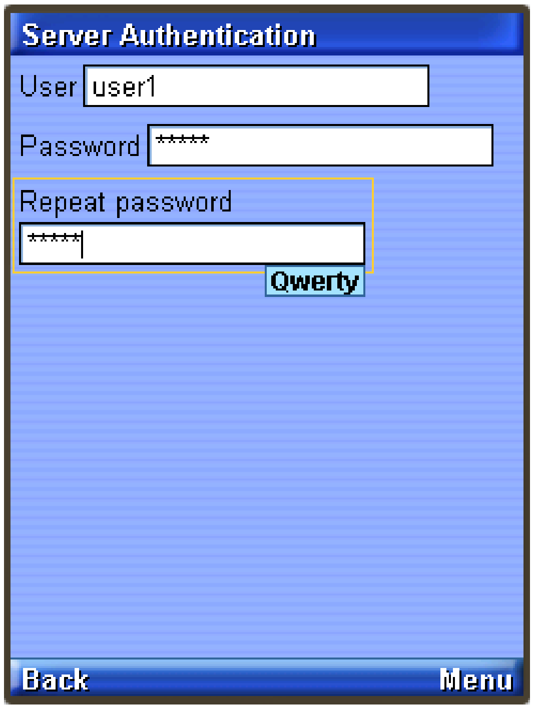


Figure 4.12: Server authentication: before the first user can be registered, the server must be authenticated.

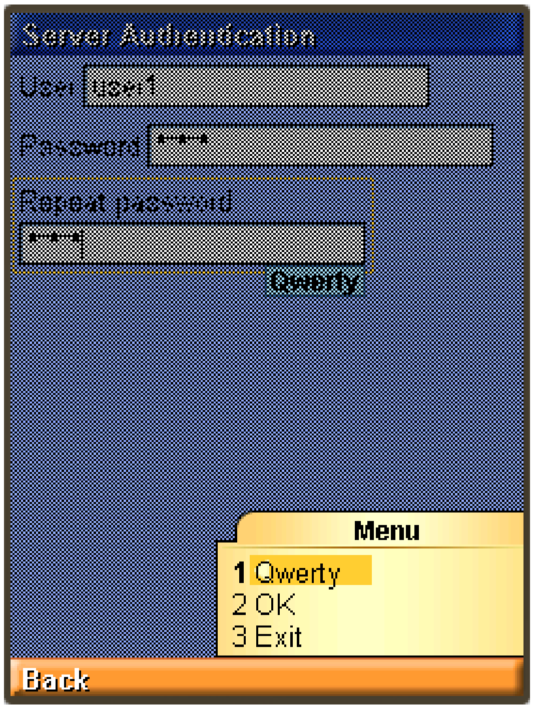


Figure 4.13: Server authentication: all the screens have a menu, here we find options like OK and Exit, the mapping of these commands can vary from device to device.

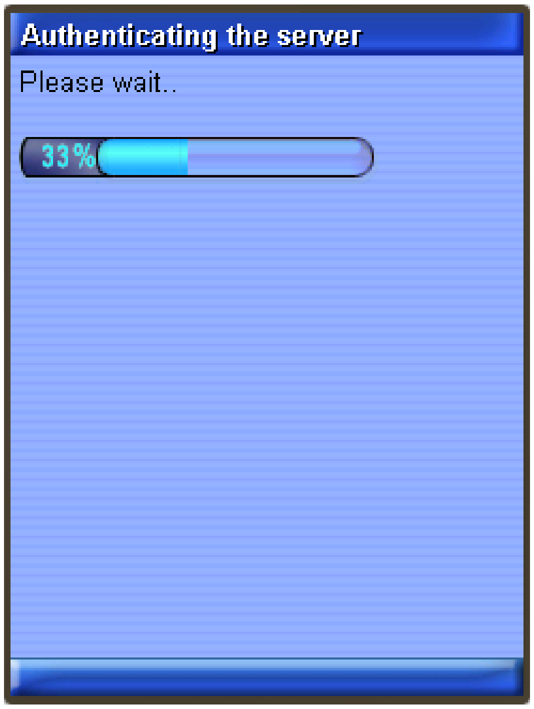


Figure 4.14: Waiting screen: this is the waiting screen, it gives the user feedback on progress so they can know the application is working.

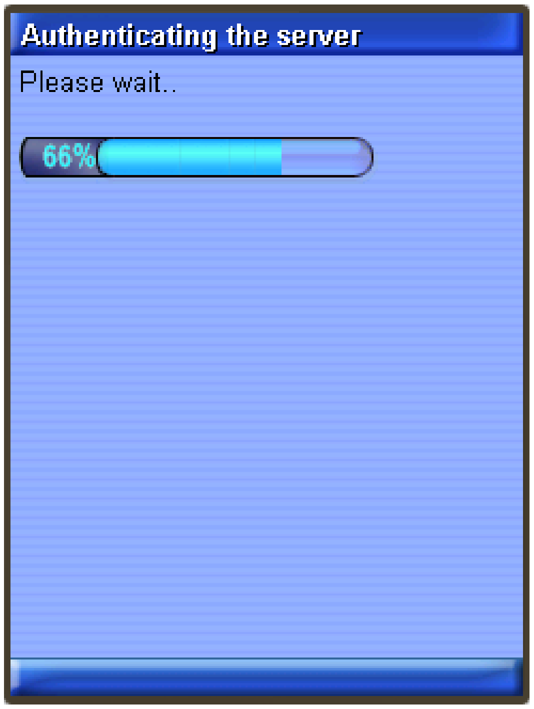
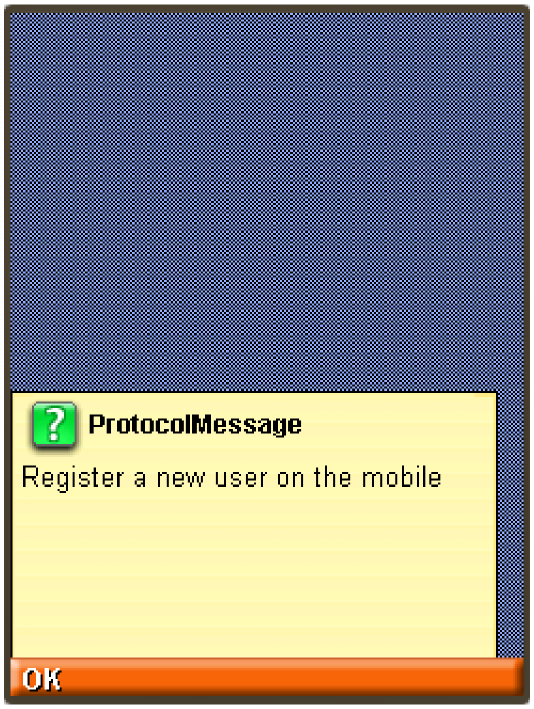
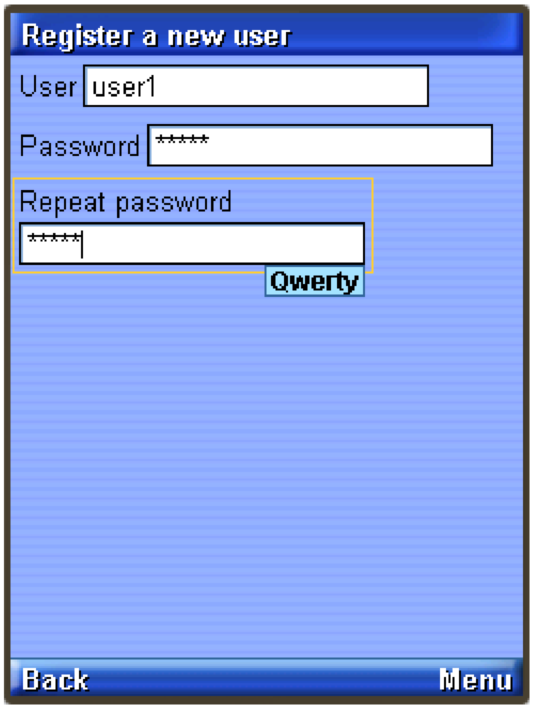


Figure 4.15: Waiting screen: getting this to work can be a challenge, see section 6.2 and the article [45] for details.


ProtocolMessage

Register a new user on the mobile

Figure 4.16: Server authentication: the server has been authenticated and a user can now be registerd.

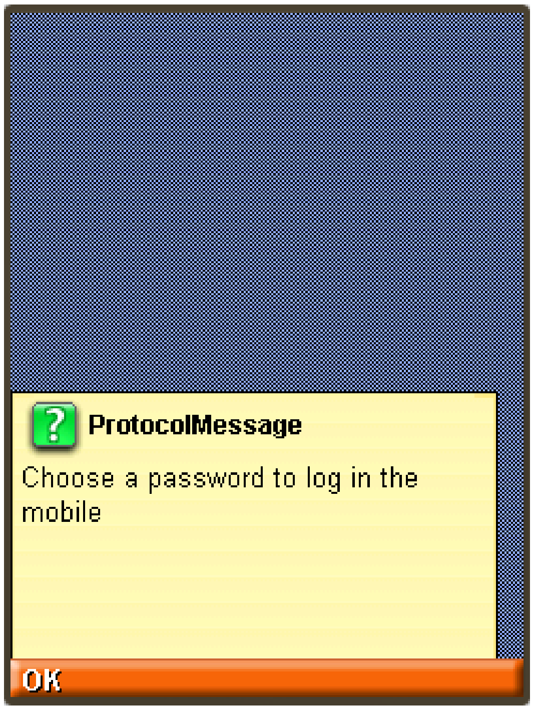

Register a new user

User user1

Password *****

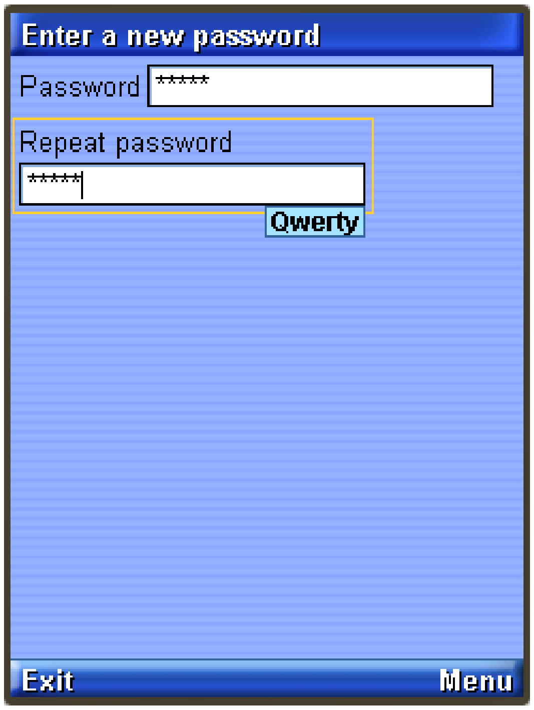
Repeat password *****

Figure 4.17: User registration: registers a new user on the device, this can be considered logging on to the server, only users on the server should be able to register on the device.


ProtocolMessage

Choose a password to log in the mobile

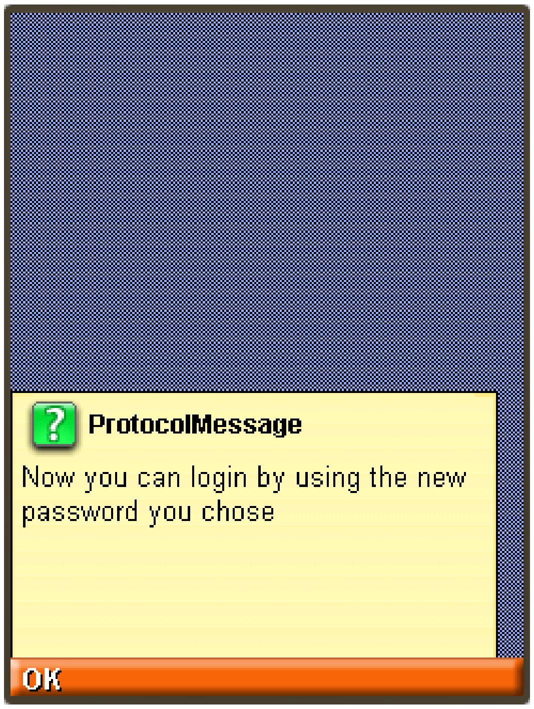
Figure 4.18: User registration: the user has been registered on the device, and now needs to select a password for loggin in locally.


Enter a new password

Password *****

Repeat password *****

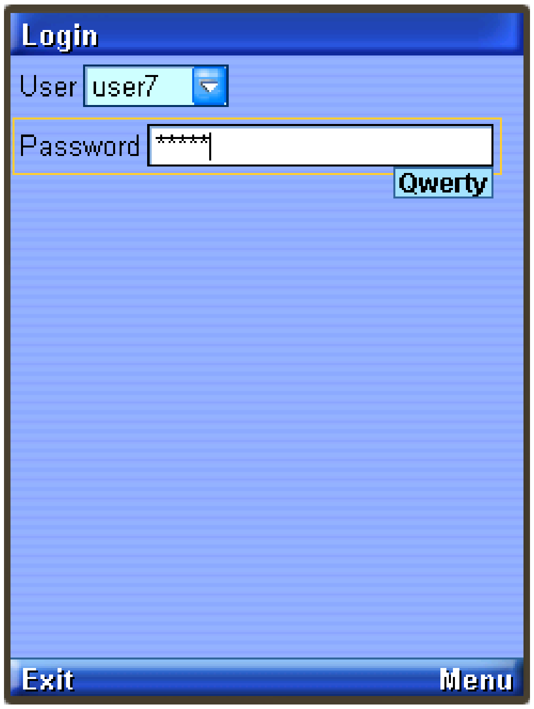
Figure 4.19: Password registration: the password is entered twice to avoid any typos.



ProtocolMessage

Now you can login by using the new password you chose

Figure 4.20: Password registration: the password is set and the user credentials are encrypted in the `UserKeyStore` using a password derived key.



Login

User user7

Password *****

Figure 4.21: User selection: once one or more users are registered on the device, this is the default screen. Entering the local password would log the user in to the application.

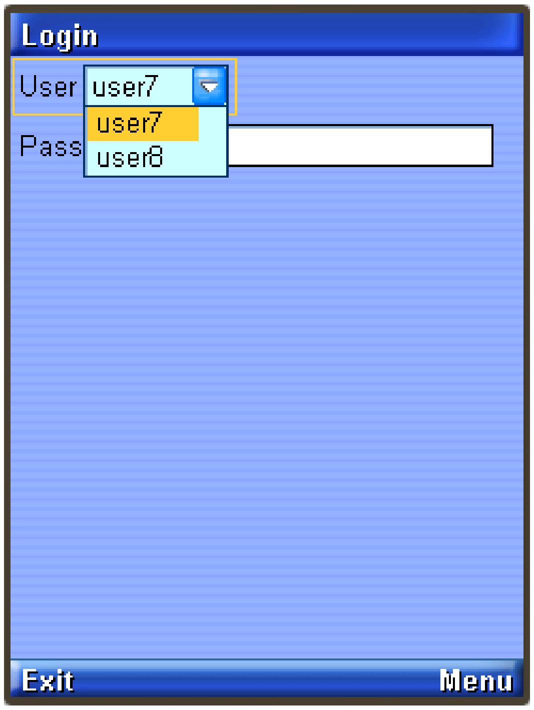


Figure 4.22: User selection: the drop down menu gives a list of the registered users on the device.

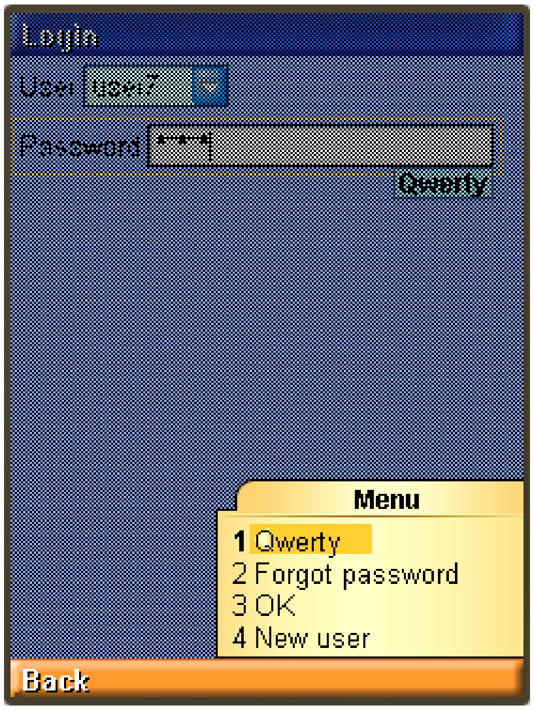


Figure 4.23: User selection: by opening the menu, a user can recover his or her account or a new user can be registered on the device.

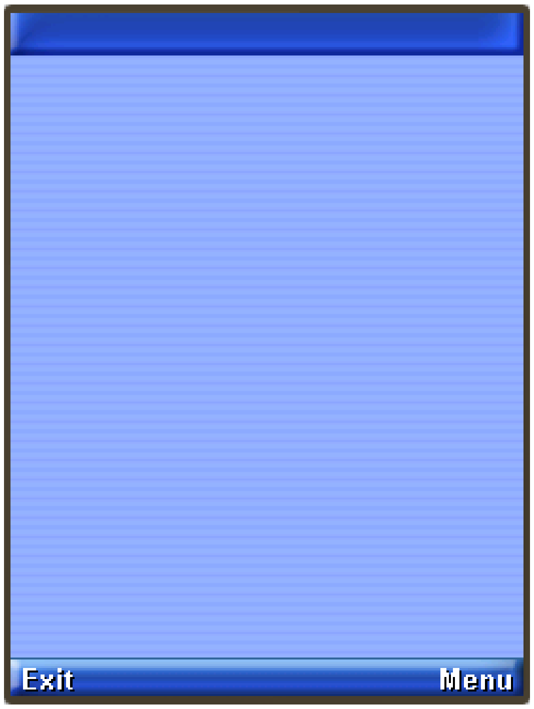


Figure 4.24: OpenXdata: Once the user has been logged in the `userMenu()` method is called, and control is given to the application. This is a screen in the openXdata application. See the next chapter for more information on the prototype integration with openXdata.

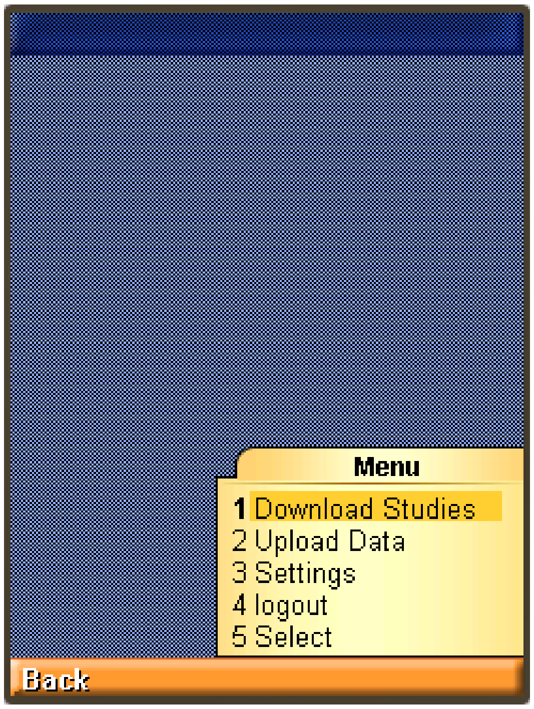


Figure 4.25: OpenXdata: as we can see these are the applications features available. Logging out would lead back to the screen in figure 4.21

5

Implementation Evaluation

5.1 Prototype Integration With OXD

As discussed in chapter 2, we are using the openXdata MDCS as a reference system for testing. As such, a prototype client which integrates the secureXdata API into an existing openXdata client has been developed. In this section we will look at how this is done and how smoothly the integration went.

5.1.1 Securing the OpenXdata Client

Since this is a prototype or proof of concept integration and since both the API and the openXdata client has evolved since the original integration started, we will not go into great detail regarding all the code changes needed. We will focus on the parts that related to the API and how it is integrated.

For the prototype we wanted to test all the features, and as such we did a complete integration using the `SecureClient`. The original openXdata client has its own log in system. This is both beneficial and troublesome. On one hand this needs to be bypassed since the `SecureClient` would take care of user authentication and so on. On the other hand, since there was already a log in system in place, there would also have to be a point after which a user is logged in. This would be where we want to give the client back control once the `SecureClient` successfully authenticates a user.

5.1.1.1 Prototype Implementation

Once the application starts, the `SecureClient` takes control and either authenticates the server or new registers users if the application is started for the first time, or asks a user to authenticate. Once a user is authenticated, as discussed in chapter 4, the `userMenu()` method is called by the controller and the client gets back control. Since we were interested in developing the API and not the openXdata client, the prototype crudely bypasses the log in part of the openXdata system by setting all the openXdata specific data and programmatically setting a default user that exists in the openXdata system as logged in. This is not ideal, but for our purposes (prototyping/testing of the API) it is sufficient.

Once authentication was in place, securing the communication and device storage was the next phase. However, since the API had been designed to be easily integrated into existing applications and because the openXdata client has a loosely coupled transport layer and RMS storage system, it was rather straightforward to secure the storage and the connection.

5.1.1.2 Ease of Integration

The integration with the existing openXdata client consisted of three parts, bypassing the existing authentication system, securing storage and securing communication.

Bypassing the OpenXdata Authentication As discussed in chapter 4, a way to add the security layer in an existing application through the API, is to extend the `SecureClient` class and once the application starts, we start the `SecureController` which performs server authentication or user registration as needed and ultimately authenticates a user followed by a call to `userMenu()` method. The user is at this point authenticated based on the data in the secureXdata server. The application then programmatically sets the default user as logged in (no actual authentication is done on the openXdata system) so that the openXdata server will accept requests passed through the secureXdata communication tunnel.

This part of the integration was the most time consuming and difficult, however it is also the most trivial. If we had been doing a proper integration and not just prototyping, the entire openXdata log in system would have been replaced by the `SecureClient`, and the secure server would have shared the user credentials with the openXdata server.

Securing storage Once user authentication was in place, the next step was to secure the storage. There are two things that we wish to accomplish: first and foremost we want to keep the data safe on the device, and secondly we wish to separate data from different users. OpenXdata has an issue with their storage system, if a user logs out, the next user logging in has access to the other users data. By using the `SecureUserKeyStore` both these issues can be solved. Since openXdata uses a centralized class for all storage and stores no data that are shared between users, very few changes were needed to secure storage and provide users with their separate storage areas.

Code listings 5.1 and 5.2 show the integration of the secure record store. Lines 4 and 9 are of interests as this is where the changes are made. Because the method signatures are the same, no other changes are needed for the record store to be secure.

Listing 5.1: The original openXdata RMS system.

```java
 1 public class RMSStorage implements Storage{
 2   // ...
 3   // Normal record store
 4   private RecordStore recStore;
 5   // ...
 6   private boolean open(){
 7     try{
 8       // Opening normal record store
 9       recStore = RecordStore.openRecordStore(name, true);
10       return true;
11     }
12   // ...
13   }
14 }
```

Listing 5.2: The secureXdata secured RMS system.

```java
 1 public class RMSStorage implements Storage{
 2   // ...
 3   // Secure record store
 4   private SecureUserRecordStore recStore;
 5   // ...
 6   private boolean open(){
 7     try{
 8       // Opening secure record store
 9       recStore = SecureUserRecordStore.openRecordStore(name, true);
10       return true;
11     }
12   // ...
13   }
14 }
```

Securing Communication The last part of the integration was securing communication. As with securing the record store, openXdata has a centralized transport layer class, so changes were only needed in one place. The below code listings (5.3 and 5.4) describe the changes needed to secure the communication. Lines 4 and 12 are of interest. Again, as with the record store, the method signatures are the same and as such no other changes are needed.

Listing 5.3: The original openXdata transport layer.

```java
public class TransportLayer implements Runnable, AlertMessageListener
    {
  // ...
  // Normal connection
  private HttpConnection con;
  // ...
  protected void connectHttp() throws IOException {
  // ...
    try{
      String HTTP_URL = (String)conParams.get(
              TransportLayer.KEY_HTTP_URL);
      // Opening the normal connection
      con = (HttpConnection)Connector.open(HTTP_URL);
  // ...
  }
}
```

Listing 5.4: The secureXdata secured transport layer.

```java
public class TransportLayer implements Runnable, AlertMessageListener
    {
  // ...
  // Secure connection
  private SecureHttpConnection con;
  // ...
  protected void connectHttp() throws IOException {
  // ...
    try{
      String HTTP_URL = (String)conParams.get(
              TransportLayer.KEY_HTTP_URL);

      // Wrapping the normal connection
      con = new SecureHttpConnection(
        (HttpConnection)Connector.open(HTTP_URL));
  // ...
  }
}
```

Other Changes In addition to the above described changes, for a proper integration with the openXdata client, somewhere the exceptions that the secure classes throw (`ProtocolExceptions`) should be caught and handled to make users aware if something should go wrong. For our prototype however we have not done this.

5.1.1.3 Actual Integration with OpenXdata

Since we used the openXdata client as our reference system, we have been in collaboration with the openXdata group throughout the development of the API. At the current time it is their wish to try and actually incorporate the API into their system. Together we decided to give priority to the integration of the secure storage part, as this requires only local changes to the client (and possibly some small changes to the server side to allow a recovery procedure for users' passwords).

In the current version of openXdata, as mentioned in chapter 2 they no longer store the entire user database on the device, only the registered users. Leveraging on this, we can use the `UserKeyStore` for storing the user credentials instead of storing them in a normal record store. When a user tries to authenticate, if a user's `UserKeyStore` exists, we try to unlock it using the supplied password and stored salt. If the password is entered three times incorrectly, the user is locked out and needs to recover the account by contacting the server. If a user's `UserKeyStore` does not exist, the user name and password are sent to the server for authentication, and, if successful, the salt is returned and the `UserKeyStore` is created.

The integration of the `SecureUserRecordStore` would be done in the same way as described in the in the prototype integration section (listings 5.1 and 5.2). Server communication would still be insecure, however providing confidentiality locally on the device is a step in the right direction.

5.2 Evaluation of performance

While designing the API, the main focus has been to make it easy to integrate with existing systems and easy to use in general. Furthermore, decoupling between the storage and communication parts have also been important. In this section we will evaluate the cryptographic performance of the current design. That means using the default `CryptoTools` implementation that is implemented using BouncyCastle.

5.2.1 Cryptographic Benchmarking

The benchmark of the cryptography will be done in the same way as the record store benchmarks. We will be benchmarking the encryption and decryption speed of the same setup as used in the default implementation of `CryptoTools`. Once we have the times, we can calculate the throughput speed of both operations, and the combined speed for decrypting followed by encrypting. Decryption followed by encryption is what happens if data is read from a secure record store and then sent to the server, this is a result of the strong decoupling between storage and communication.

The reason it is of interest to view the results as throughput speed and not just as times is that this relates to the upload / download connectivity speed on the device. If the speeds for the cryptographic operations are much lower than the speed at which data can be transferred to or from the server, then this means that the cryptography is a bottleneck in the system, which we do not want.

As with the previous benchmarks, we include the code listings of the actual benchmarked code.

The Cipher Method Unlike the record store and serialization benchmarks which were all on Java ME API methods, the cipher method is made using a third party library, and as such it is relevant to see how the method itself is designed.

Listing 5.5: Cipher benchmarks: Common cipher code used in both benchmarks.

```java
private PaddedBufferedBlockCipher symEnc = new
  PaddedBufferedBlockCipher(new CBCBlockCipher(new AESLightEngine()));
private PaddedBufferedBlockCipher symDec = new
  PaddedBufferedBlockCipher(new CBCBlockCipher(new AESLightEngine()));

/**
 * symEnc is initialized to encrypt
 * symDec is initialized to decrypt
 */

private void init() {
  // ...
  symEnc.init(true, new ParametersWithIV(new KeyParameter(key), iv));
  symDec.init(false, new ParametersWithIV(new KeyParameter(key), iv));
  // ...
}

/**
 * Performs the cipher operation on data using the provided
 * cipher object. Returns the resulting byte array.
 */
private byte[] cipherData(byte[] data,
                           PaddedBufferedBlockCipher cipher)  {
  // get the minimum output size.
  int minSize = cipher.getOutputSize(data.length);
  byte[] outBuf = new byte[minSize];

  // process all blocks of 16 bytes
  int length1 = cipher.processBytes(data, 0, data.length, outBuf, 0);
  int length2=0;

  try {
    // process the rest and add padding
    length2 = cipher.doFinal(outBuf, length1);
  } catch (Exception e) {
    e.printStackTrace();
  }

  // Return the result.
  if (cipher == symEnc)
    return outBuf;

  int actualLength = length1 + length2;
  byte[] result = new byte[actualLength];
  System.arraycopy(outBuf, 0, result, 0, result.length);
  return result;
}
```

Benchmark K - BouncyCastle `PaddedBufferedBlockCipher` encryption. This benchmark will determine how long it takes to encrypt a byte array of varying size and content.

Listing 5.6: Benchmark for encryption

```
long total = 0, result;
for (int i = 0; i < itt;i++) {
  // generate test data
  startClock();
  encryptedData = cipherData(dataToEncrypt, symEnc);
  result = stopClock();
  total += result;
}
return total;
```

Benchmark L - BouncyCastle `PaddedBufferedBlockCipher` decryption. This benchmark will determine how long it takes to decrypt a byte array of varying size and content.

Listing 5.7: Benchmark for decryption

```
long total = 0, result;
for (int i = 0; i < itt;i++) {
  // generate test data
  startClock();
  decryptedData = cipherData(dataToDecrypt, symDec);
  result = stopClock();
  total += result;
}
return total;
```

5.2.1.1 Results

Data Size (bytes)	**Encrypt**	**Decrypt**
32	3,0	3,8
64	4,5	6,4
128	8,4	10,8
256	15,3	19,5
512	29,0	37,8
1024	56,1	74,9
5120	277,6	365,9
10240	556,2	720,9
20480	1103,7	1454,1
25600	1374,8	1824,4

Table 5.1: Benchmark results for encryption and decryption using the default `CryptoTools` implementation. All times are in ms.

Data Size (bytes)	**Encryption**	**Decryption**	**Combined**
32	11,1	8,7	4,9
64	14,5	10,2	6,0
128	15,7	12,2	6,9
256	17,2	13,5	7,5
512	18,1	13,9	7,8
1024	18,7	14,0	8,0
5120	18,9	14,3	8,1
10240	18,9	14,5	8,2
20480	19,0	14,4	8,2
25600	19,1	14,4	8,2

Table 5.2: Speeds for encryption and decryption. The combined speed is the throughput speed for decrypting and then encrypting some data. Speeds are in kb/s

As we can see from the tables 5.1 and 5.2 and the chart 5.1, cryptographic operations are very costly on the slow processor. If these results are acceptable or not depends on the context in which the API would be used. How large are the data sets stored? What connectivity conditions will the application operate under? In the end it comes down the the individual project to determine if these results are acceptable. One thing we can however conclude is that the throughput speeds on the used device (Nokia 2330c) are not very high. Under optimal conditions a GPRS/EDGE [21] as of 2007 speeds up to 22 kb/s can be reached for upload [23]. If 3G [19] then higher speeds could be expected. It is however likely that the areas where the API and device would be used (low-income countries) would not have optimal conditions.

Since reliable data regarding what connectivity conditions to expect are scarce, we try

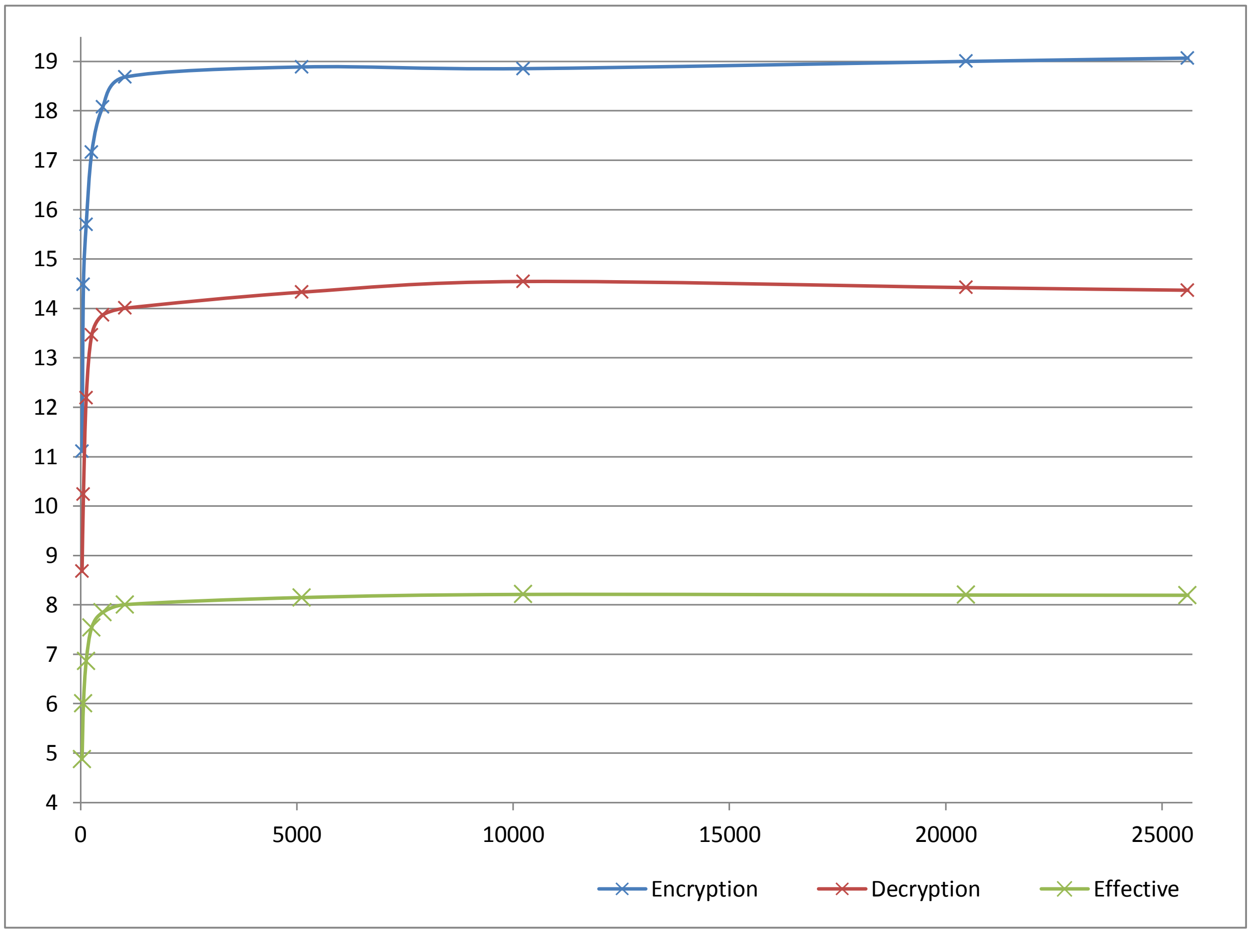


Figure 5.1: A chart displaying the speed results found in table 5.2. The X axis is bytes and Y axis is kb/s. Each node is a data pair from table table 5.2.

to put the speeds into perspective using a different factor: the maximum size of a given record store on the device. This also provides some insight in how the throughput is when looking at the other specifications of the phone. The size can be found easily by using the `getSizeAvailable()` method on an empty record store. Doing so shows that the record store can contain a maximum of 131072 bytes, or 128 kilobytes. These bytes would include any bookkeeping or structure overhead from the RMS system itself. Comparing this to the speeds of the larges benchmarked data set, we get the following times: 6,7s encryption, 8,9s decryption, or 15,6s for both. Waiting 6,7s to store some data is a fairly long time to wait, but then we are talking about a full record store. The average English word length is somewhere around 8 [43] characters. Lets assume that due to spaces or other formatting characters the average length is 10. This means that a full record store corresponds to roughly 1300 words of text, that is a lot of text for a low-end mobile device. OpenXdata estimates a form being between 1 and 50 kb in size, usually a couple of kilobytes.

5.2.1.2 Making it Faster

In this section we will look at what can be done to improve the cryptographic performance on the device. One obvious way is to upgrade the hardware, but this is not a viable option.

Save for Upload The main issue with the current API design is that when something is being transferred from the secure store and to the server, it needs to first be decrypted and then re-encrypted before sending. Though this makes for highly decoupled and flexible implementations, it has a severe impact on the performance as seen in section 5.2. In an earlier version of the protocol [35], the API is designed in such a way that data can be uploaded as it is in the record store. This means that the only encryption needed is when data is stored and data is only decrypted if it is edited.

In most cases, data is not stored all at once. Take forms, for example. A form would have to be filled out before being stored, so the actual time spent storing the data would be some seconds over the course of a session making the actual wait time negligible or at least appear so. For upload on the other hand, all the forms being uploaded would have to be processed at once.

The reason this was not done during the current work was that the main priority was to make the API easy to integrate into existing systems. After looking closer at how the different cryptographic operations perform on the device however, the candidate suggests that this should be changed.

A solution to this issue would need to solve three problems. The first is how the data should be stored, so that it can be uploaded without being decrypted and at the same time being accessible on the client. A possible solution is to let each separate record be encrypted using a randomly generated key. The key with the corresponding IV is then encrypted using the storage key and stored together with the encrypted data in the record. See figure 5.2 below for an illustration of what this might look like. Since each record would have it's own key, the different records could be uploaded, changed, or uploaded independently from each other.

The second problem is how to write the data from the record store into the output stream without exposing encrypted data to the programmer, and without allowing the programmer to write plain text data to the stream. This is a problem because the output stream and the record store are in different packages, making any direct communication between them public and also available to the programmer. The existing `SecureHttpConnection's` `SecureOutStream` encrypts anything written to it so this cannot be used.

A number of different solutions have been considered, the easiest being having a public write method for writing plain text data, though this would make the API too easy to misuse. Other alternatives are moving everything into one package, which is not acceptable since

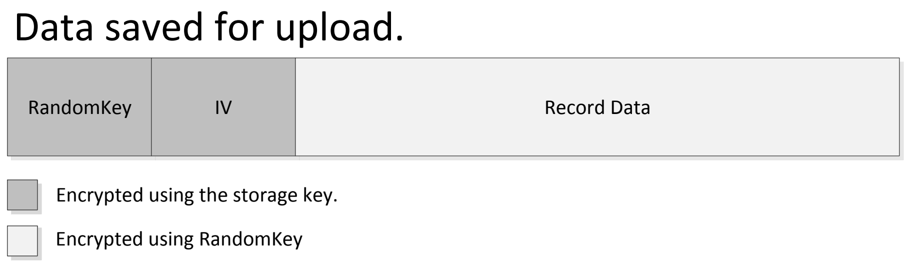


Figure 5.2: Record format when saving for upload.

storage and communication should be separate. The best solution the candidate could come up with is to make a separate write method which takes some interface as input. This interface would contain the bytes from a record. As we will discuss in the next paragraph, some object would then read data from the record store, wrap it in the interface, and pass it to some output stream or object. Since the write method and the interface would both be public this could still be misused, but it appears to be the best solution.

The third and final problem stems from the transparent design used in the rest of the API. As with the `SecureHttpConnection`, we want to encapsulate the request in such a way that it appears to the programmer they are writing into an output stream. This can be achieved by making an additional iterator, such as `SecureUploadIterator`. This iterator would be constructed using the factory pattern on a opened secure record store object containing the records that should be uploaded. The constructor would take some specialized `SecureHttpConnection` or `SecureOutStream` as input parameter.

The iterator object would have methods for iterating through record descriptions 4.5.3.2 or uploading given record ids. Whenever something is written by the programmer to the output stream, it is cached somewhere. When the iterator is told to upload some record, that encrypted record would be written into the stream as plain text, together with the session key encrypted key and IV for the record. Furthermore, some placeholder token with an id would be added to the cached output data so that the secure server would be able to reconstruct the correct request. In the end, the cached data would resemble the formatting string of a call to `printf()`. As the connection is finalized, the cached data would be finally written to the server. Figure 5.3 shows what the HTTP request could look like. If a session exists, the secure HTTP header would be omitted as shown in figures 4.9 and 4.9.

Code listing 5.8 shows a code example using the proposed iterator. In a real example, a `DataOutputStream` would be used to structure the data instead of a simple ; separator. In figure 5.4 we depict what the "Data written by the programmer and `SecureUploadIterator`" portion of the request (see figure 5.3) would look like. "REC1" and "REC2" are placeholders for the record data in the request. The server would first read the encrypted records and decrypt them using the supplied keys. The "Data written by the programmer and `SecureUploadIterator`" part of the request could then be parsed by the server, and the decrypted record store data be put where the placeholders are. Meaning, the plain text

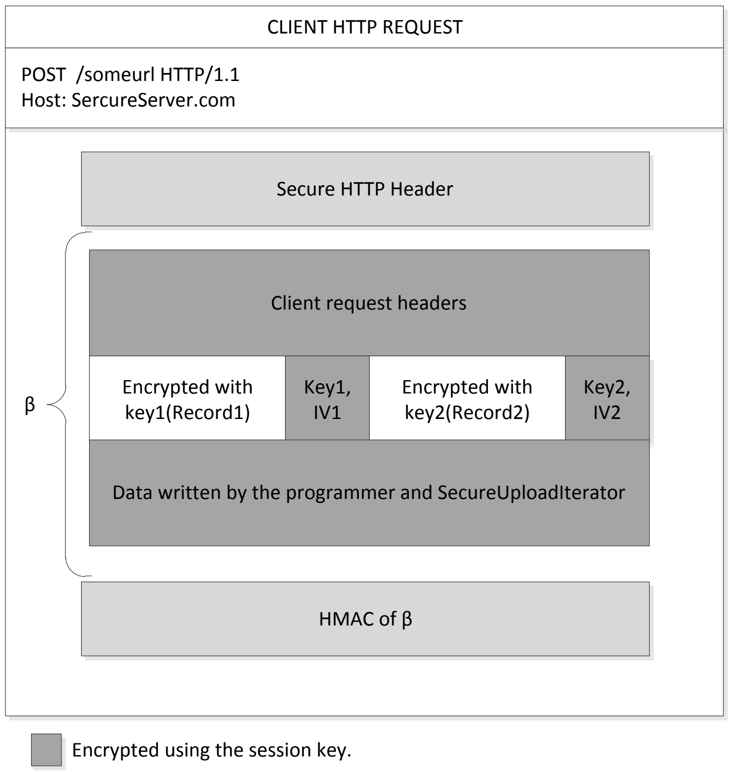

CLIENT HTTP REQUEST
POST /someurl HTTP/1.1
Host: SercureServer.com
Secure HTTP Header
β
Client request headers
Encrypted with key1(Record1)
Key1, IV1
Encrypted with key2(Record2)
Key2, IV2
Data written by the programmer and SecureUploadIterator
HMAC of β
Encrypted using the session key.

Figure 5.3: The structure of the client request data when uploading directly from store.

record data for record 1 would be where the place holder "REC1" is in figure 5.4. The reconstructed request can then be sent to the server on the other side of the secure connection.

Listing 5.8: Example: Possible usage of the store for upload code.

```java
protected void startApp() {
// The data sink we want to send the records to.
SecureUploadOutStream out;

// The store containing the data.
SecureUserRecordStore store;

/* The store and connection has been opened */

// Open the upload iterator
SecureUploadIterator uploadIt = store.secureUploadIterator(out);
// write some server command in the request
out.write("UPLOAD;2;");
// write record 1 into the request
uploadIt.uploadRecord(1);
```

```
// separator for server
out.write(";");
// write record 2 into the request
uploadIt.uploadRecord(2);
// finish the request and flush.
out.write(";");
out.flush();
out.close();
```

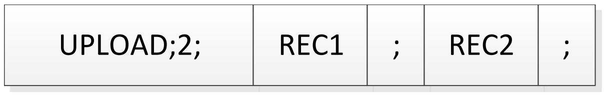


Figure 5.4: Example: "Data written by the programmer and `SecureUploadIterator`" portion of figure 5.3 after running the code in listing 5.8.

Instead of caching the programmer made request with placeholder tokens in it, we could write this into the stream immediately and then append the record data sets at the end. However, this would mean that the programmer cannot delete the record after it is written to the stream, since it is not actually read until we flush.

There are still some issues to be resolved, such as how to make sure the placeholder tokens are unique, that is, the data written by the programmer can not be mistaken for a placeholder token. If the programmer uses a `DataOutputStream` for writing his or her data, then the `SecureUploadIterator` needs to do the same. Since this has not been tested in the API, and since there are probably some other quirks that need be resolved for it to work, we propose this as future work.

Alternative Cryptographic Implementations The use of other cryptographic implementations or algorithms might give better performance. For instance, stream ciphers could yield better speeds [59]. As mentioned in previous chapters, the JSR177 standard [56] could be better in multiple ways (memory footprint, memory usage, speed). These things would have to be tested though, and are outside the scope of this thesis.

Using Compression If speeding up the cryptographic operations is not an option, then reducing the amount of data to process might be an alternative. This can be achieved by compressing the data prior to encryption. Since the compression ratio is highly dependent on the input data, and the data can be more or less anything, we can not give any concrete answer as to whether or not compression should be used. Assuming the speed of the operations are more or less regardless of the data, we can however get an indication to whether or not compression can be beneficial at all.

The tests are run using a old version of JZlib [44] using the default configuration, we are compression randomly generated data. For details on the benchmark setup can be found in section A.2 in the appendices.

Data Size (bytes)	**Compress (kb/s)**	**Decompress (kb/s)**	**Even Split Ratio**
32	0,3	7,4	N/A
64	0,6	13,5	N/A
128	1,1	23,4	N/A
256	2,0	15,8	N/A
512	3,5	24,2	N/A
1024	6,0	36,2	0,07
5120	13,3	58,6	0,58
10240	15,6	63,6	0,63
20480	16,0	64,6	0,64
25600	16,1	65,9	0,64

Table 5.3: Benchmark results for compression and decompression using jZlib. The even split ratio is the compression ratio needed in order to get the same total time using compression and not using compression in the following scenario: data is compressed, then encrypted and stored in a record store, data is read and decrypted, written to a `SecureHttpConnection` and thereby encrypted. In other words it's the minimum ratio for compression to give any positive effect.

As already discussed, the total time spent encrypting data when storing is distributed across a number of operations in a session. Uploading data however does not benefit from this and as a result long consecutive wait times can occur for large amounts of data. Regardless of the compression time, any compression ratio below 1.0 would be beneficial with regards to upload. The compression occurs before encrypting the data for storage, so the time spent compressing would also be distributed. Since the data needs not be decompressed before being uploaded, this would reduce the overall size of the data and therefore also the time spent handling it during upload. In other words we would move some of the waiting time from upload to when data is being stored. This could benefit the user experience.

The even split ratio is the compression ratio that would result in the same total wait time for the user. Total means the time it takes to store (herein also the compression) in addition to the it takes to decrypt and encrypt during upload. The time distribution would be shifted so that more time is spent storing data and less time spent uploading, but the total time would be the same as without compression. Any compression rate smaller than this would result in less total time than without compression.

Looking at the results in 5.3 we can see that for data smaller than 5kb compression is rather pointless, the overhead is so large that it cripples the throughput speed, and as a result it would be better to not use compression at all. For larger data sizes compression might be a viable option, according to zlib typical compression rates range from 0.5 to 0.2 which is better than all the even split ratios for data above 5kb.

5.2.1.3 Conclusion

Cryptographic algorithms are computationally intensive, and running them on a device with limited processing power such as the target device 2330c gives rather low throughput speeds. Whether or not these speeds are acceptable depends on the circumstance under which the API is being used. If data transfer rates are low as well, there might not be any noticeable difference. However, if connectivity conditions are very good, then the cryptographic processing will definitely be a bottleneck.

Changing the API design to allow uploading of records as they are stored on the client, the cryptographic implementation used or applying compression to lower the amount of data to transfer are some ways to improve performance. Compression can also be seen as the means to move some of the time spent uploading to a earlier stage when the data is saved. This can be beneficial since the time spent saving data is distributed, ie. 20 records can be saved across the duration of 2 days. Whereas with uploading it will most likely have to process multiple records at once.

6 Experiences

In this chapter the candidate will reflect on some of the experiences he is left with after working on the project, both in terms of the tools used and general experiences.

6.1 Programming in the Large

The candidate finds that the most valuable experience gained through the work described in this document has to be from working with the mHealth Security Group and the collaboration with openXdata. While the candidate has worked on a number of projects in the past, these are more or less exclusively part of some course in school working with friends, classmates or acquaintances.

As part of the secureXdata project the candidate was part of writing a number of papers, one of which was accepted at ESSoS12 [5] and one that is still pending acceptance. This part of the work has given the candidate a greater insight into how academic research is done and works.

Through the collaboration with openXdata the candidate got exposed to programming in the large. Having discussions with people working in a system that they will continue to work on and care about for many years to come, is very different from discussing this weeks programming assignment with your fellow classmates. Stakeholders have much strong opinions and have to conform to micropolitics and the overall goal of the project, which they may or may not agree with.

Seeing how openXdata is structured in order for it to be able to be a collaboration between different groups and people from across the world, all with different personal goals, has given the candidate an insight into some of the non-programming related challenges behind larger projects. A simple discussion becomes much more complicated when the participants are from five different timezones and nationalities.

6.2 Java ME

Before starting this project the candidate had no previous experience with Java ME, or developing applications for hand held devices in general. Java ME is as discussed in chapter 2 a platform for running applications on mobile devices. As a result of previous experience with with Java SE [54] and Java EE [51] starting to work with Java ME [52], it was not difficult to pick up. The remainder of this section will be divided into three categories.

The Good Since Java ME and Java SE are both still java at the end of the day, having experience with one will make using the other much easier, at least as far as basic syntax and concepts are concerned. So picking up Java ME and writing simple applications is pretty straight forward with some previous java experience. The application flow is pretty straight forward and there are many great resources online to get you started [13].

The Bad Since Java Me is more or less a stripped down version earlier of Java SE, there are limitations compared to Java SE. Most of these are annoyances such as the lack of for-each loops, or more frequently encountered, lacking commonly used and familiar packages such as `java.util.*`. A lot of the time the candidate would find himself trying to use classes from the `java.util` library. In some cases, such as with lists and collections, Java ME provides some lesser alternatives such as the `Vector` class. As its features are not as rich and there are no generics in Java ME (see next paragraph) this can be tedious compared to Java SE. Objects retrieved from a vector need to be cast to the correct class for instance, which requires extra work and can easily cause mistakes. In other cases, re-implementing commonly used methods such as `Arrays.toString()` or `Arrays.equals()` was needed as there is no alternative available.

The Ugly The only lack in functionality that couldn't be circumvented was the absence of generics. Generics are a powerful tool for reusing or making dynamic code and the candidate found himself trying to use generics a number of times before getting used to the Java ME restrictions.

Apart from functionality, there is one aspect of the Java ME platform the candidate found to be difficult at first. The system thread or the main thread of the application is responsible for screen updates and other system related tasks. When the candidate tried to make a waiting screen for user feedback while the application was communicating with a server, the application would freeze for the duration of the communication and then for a fraction of a second display the waiting screen before setting the next screen.

This happened because the lengthy process of downloading data was being run in the system thread, so while the system had been instructed to set the new screen, it had to wait for the download process to complete in order to be freed and thus able to order to do so, at which point the download would be done.

When deploying the openXdata prototype on the actual device, this became more than just a nuisance, the application manager on the device would prompt the user to allow the client to connect to the Internet, but since the system thread was busy waiting to connect, it would not properly handle the user response and instead the client would deadlock.

Understanding how the system thread works is critical in order to be able to make responsive applications that won't deadlock. For a more in depth explanation see the article "Networking, User Experience, and Threads" [45]

Because the applications are run on emulators it can at times, be very tedious to test your code. This was very frustrating when being used to working with Java SE programming.

6.3 Eclipse

Due to previous experience and personal preference the candidate used Eclipse [32] during the project. Both for writing the applications and for writing this report. The candidate has had good experience with Eclipse throughout the work, in the rest of this section we will look closer at some of the plug-ins and features used.

6.3.1 EclipseME

EclipseME [33] is a Eclipse plug-in for developing Java ME applications in Eclipse. It allows you to create a Java ME suite project in Eclipse in which you can add Java ME MIDlets. The candidate found the plug-in very useful when initially starting to make applications. EclipseME will take care of creating and configuring the JAD file and setting up and running the emulator when you run your code.

Installing and configuring the plug-in was pretty straight forward, install the desired Java ME SDK and then install EclipseME using the Eclipse plug-in manager. Details can be found at [14].

The candidate found that once the projects grow in complexity, meaning additional libraries and things like code obfuscation and so on comes into the picture using EclipseME can become tedious. All settings are set in menus and unless you have used the plug-in for a while it is not always that easy to find which menu does what.

If you add a code obfuscation step to your build process, your build might fail during the preverification/obfuscation step, even though the un-obfuscated code will run fine. The error messages return from the preverifier or obfuscator give little indication to what's wrong. In some cases adding the libraries to the preverifiers command line parameters remedied this problem.

All in all, the candidate found the EclipseME plug-in to be very useful as an aid when initially starting to develop Java ME applications. However, once you have gained a fair understanding of how the different pieces fit together the candidate found that using build scripts were a better solution. Moving from EclipseME to using a build script is easy as EclipseME can generate one for the existing project. See the next section.

6.3.2 Apache Ant

Ant comes as a part of Eclipse, and so no installation is required.

Ant primarily has two benefits compared to using EclipseME. Firstly, anyone wanting to work on or compile the project needs to be using both Eclipse and EclipseME in order to properly import the project. Using ant scripts, anyone can build and run the solution regardless of what IDE and configuration they run, in fact, a normal text editor and ant is sufficient.

The second benefit, which some might see as a downside, is that ant builds based on a build script. The candidate finds that build scripts give much more control over the build process and what goes into it and comes out. Any error messages that might be produced during the build process were also easier to resolve than when using EclipseME.

The openXdata client is distributed with a ant build script as opposed to being a EclipseME project.

The downside of using ant is that while EclipseME takes care of mostly everything, especially for simple projects, with ant one needs to create the build scripts as well as the code. Once learned however this becomes a very powerful tool.

6.3.3 Subclipse [16], LaTeX [7] and Texlipse [18]

Subclipse is a Eclipse plug-in that provides subversion from within Eclipse. Installing it can be done through the Eclipse plug-in manager. Subclipse is easy to use and provides all the features the candidate was looking for in a version control program.

This document is written in LaTeX, the candidate had no previous experience with latex prior to writing this report. Knowing HTML and and programming, LaTeX was a very nice way of writing documents. Unlike Microsoft office and other WYSIWYG (what you see is what you get) editors, working in LaTeX a template controls most of the layout so you can focus on writing the content.

For writing the LaTeX documents, the Texlipse plug-in for Eclipse was used, this provides a view, project type and configuration for working with LaTeX in eclipse. The plug-in can be installed from the Eclipse plug-in manager, this does however not include the programs used to compile the LaTeX documents. The candidate has been using MiKTeX [1] which is a package that contains all the needed tools for making a LaTeX document.

6.4 Emulators

During the development, we used emulators for testing the application and API prior to running it on the device itself. This makes things much easier since not only is it much harder to test/debug the application on the device, but to the candidates knowledge it is not possible to deploy a JAD and JAR pair to the device through Bluetooth or cable. This means that in order to run a proper test of the application, it has to be downloaded from an external server, meaning paying for bandwidth costs. At 5 kr per megabyte and a 200kb application, not many deployments are needed before this starts to be come costly. This might not be the case with all devices, but is the case with the Nokia 2330c.

One way to work around this is to include all the JAD attributes in the JAR manifest when running tests. As we discussed in chapter 4 we want to be able to generate some JAD attributes after the code has been compiled, these attributes can be inside the JAR when testing, making it possible to deploy over Bluetooth. Once we have a working version we take them out of the JAR manifest and deploy by downloading from a server.

Originally the Java ME SDK 3 [53] emulator was used when doing initial testing and development of the API itself. However, once integration started with the openXdata client, the Sun Java Wireless Toolkit 2.5.2 (WTK) [55] emulator was introduced as this is used by openXdata for client development.

One interesting observation is that out of the box, the emulators behaved differently. The Java ME SDK 3 emulator put very few restrictions on the emulated application, for example running benchmarks that took up to a minute on the device were instant on this emulator. The WTK 2.5 emulator on the other hand seems to emulate the device specifications, or at least the specifications of a low-end device, and the same benchmarks took if not a minute, at least some notable time.

There were some other difference as well. Unlike the SDK emulator, the WTK one prompted the user for permissions when the unsigned application tried to access the Internet in the same way as the device would, this revealed some issues with the system thread. On the SDK emulator it would run smoothly and everything would work, but on the WTK emulator and on the device the application would prompt for user permission and thus deadlock. A third and very important difference between the two is that the WTK emulator persists any record stores between runs. So if the emulator is shut down and then started with the same application at a later point, any data saved in the record store on the first run will still be present. The SDK emulator on the other hand, does not persist anything and would start the second time with empty records.

It is likely that either of the emulators can be configured to behave the way the other does, however the candidate found it useful to have them behave differently. The SDK emulator made early stage testing easier since it would always start with empty record stores and not bring up permission screens. However, once the application worked as intended, the WTK emulator would be used for testing if the application behaved correctly when the record stores were not empty, or if there were any parts that could deadlock due to improper threading. In fact the WTK emulator would deadlock in cases where the Nokia 2330c would not, most likely other devices would not recover from these deadlocks however.

6.4.1 Emulator vs Deployment

Deploying the actual application to the device would be done by either installing it from a online source (have to pay for bandwidth) or by sending the JAR file to the device using Bluetooth. However, even with testing on both emulators prior to deployment, the application would not always run as expected on the device. One such issue is with flushing. On either of the emulators, flushing a `HttpConnection` `OutputStream` would do just that, flush the data. On the device however, flushing would also close the connection which caused the application running on the device to throw seemingly unexplained `IOExceptions` until this difference was discovered. In other cases, minor errors might run fine on the emulators but not on the device.

In general, as long as the application ran on the WTK 2.5 emulator, it would behave in the same way on the actual device.

6.5 ProGuard and Code Obfuscation

ProGuard [57] is a tool for shrinking, optimizing, obfuscating and pre-verifying java class files. There are two reasons why we use this tool.

The Java ME (also known as lightweight) version of BouncyCastle contains some Java SE classes in the java/* namespace. According to their FAQ [9] his is to provide better consistency between any server application and Java ME applications. However, the JVM will not allow you to create classes in the java namespace and as such, code obfuscation is required to make this work. In short, obfuscation means that methods, classes and packages have their name changed to something that would be hard to read for humans [61]. This changes the namespace and therefore the JVM will accept the code.

Since mobile devices have size limitations for binary files run on them, including large libraries such as the BouncyCastle API, it will make the resulting binary too large to run on the phone. ProGuard however shrinks the bytecode by removing any unused parts and as a result we get a much smaller compiled binary that will fit on the device. The secureXdata integration JAR is 1,33 megabytes prior to obfuscation/reduction, the Nokia 2330c maximum JAR size is 512 kb, so the application is way too big. After ProGuard has been run however the JAR is 213 kb, and fits nicely on the device.

Obfuscation, as the name suggests, also makes the code hard to read and as a result reading a stack trace becomes considerably harder since none of the methods or classes have meaningful names anymore. (See listings 6.1 and 6.2) ProGuard does however come with a tool (ReTrace) used to de-obfuscate stack traces. This did however not work out of the box. The output from the emulators are different from the standard/expected stack trace structure, (see listings 6.1 and 6.2 for comparison) which means that ReTrace is unable to parse the supplied stack trace.

In order to get a readable stack trace from de-obfuscating, either post-processing of the output to make it conform to the expected format is needed. Or alternatively we can supply ReTrace with a custom regex that will match the output from the emulators. The candidate had trouble getting the latter to work, and in the end had to debug the ReTrace source code to find the problem: for ReTrace to be able to parse the given stack trace correctly, the regex used contains some custom wildcards which are used to signal that here comes a class name (%c) or a method definition (%m). See [58] for all wildcards and more details. The reason the candidates regex didn't work was related with the Windows operating system, the % character has special meaning when read used in the windows command line terminal or scripts for it. This caused the % characters in the regex to be evaluated and thus the command passed to ReTrace was not what was written in the script file. The solution is to escape the % by using %%. In this way the correct regex is passed to the program, as seen in listing 6.3.

Listing 6.1: Expected stack trace format.

```
java.lang.Exception:
  at r.a()
  at r.run()
```

Listing 6.2: Emulator stack trace format.

```
[exec] java.lang.Exception:
[exec]  - r.a(), bci=438
[exec]  - r.run(), bci=19
```

Listing 6.3: Batch script for de-obfuscation.

```
@echo off
REM set the regex to the variable r and reference it in the call.
set r="(?:.*\s%%c:.*)|(?:.*-\s%%c.%%m\s*\(.*\).*)"
java -jar retrace.jar -regex %r% map.txt trace.txt
```

Working with obfuscated code makes things more complicated. Not only does it require some extra effort to be able to read stack traces, you add another step to the build process which introduces new tools and processes that can break and ultimately the build time is extended.

6.6 Character Encoding

As discussed in 4.5.1.3, we hash the JAD attributes to verify the server and the integrity of the JAD file. While working on this part of the API, the candidate encountered some problems. Even though the values were all the same both on the server and the client, the generated hashes were different.

It was pretty obvious that this had to do with character encoding, what was not so clear at first was the number of places that this could be an issue.

The first thing we made sure was that the file was saved in UTF-8, then we had the server read the file as UTF-8, this gave a different hash, but still not the correct one. Turning the String object containing the data we want to hash into bytes, we used the `getBytes()` method. This however will return the bytes depending on the server settings, so we need to explicitly specify the encoding `getBytes("UTF-8")`. Now we have the correct bytes to hash. The client returned UTF-8 by default, but this will probably differ between devices so using `getBytes("UTF-8")` here as well would be good measure.

6.7 Wireshark and WebScarab

When working on anything related to communication, it is very useful to be able to inspect the communication going between the emulator and the server. Wireshark [24] and WebScarab [15] are two tools used by the candidate to monitor the traffic going back and forth, without these, finding bugs and verifying that things are as they should in the communication, would be much more difficult. Wireshark is a packet sniffer while WebScarab is a HTTP proxy which lets you intercept and read/edit any HTTP(S) communication going trough it.

6.8 Best Practices and pitfalls

This section will in short summarize some best practices and pitfalls that the candidate find would be useful to anyone who would work on Java ME or similar projects.

6.8.1 Java ME

General

System thread and threading In order for the application to be responsive, never run any code that will not return at once in the system thread, make your own.

Character encoding Always specify the encoding when using `String.getBytes()` and be consistent with the encoding across the entire system.

Use build scripts Use build scripts for building your applications instead of IDE plug-ins. You'll spend more time working and less time looking for the correct menu to set some options. Some plug-ins such as the EclipseME can generate a build script based on your current project as well.

Record Store

Serialize when possible It is beneficial to store sets of data (for instance an object) as a single serialized byte array in a single record instead of storing multiple smaller records.

Record id's are not reused When deleting a record, the record id will not be reused.

Communication

`flush()` might close the stream On the Nokia 2330c, flushing the output stream of an `HttpConnection` will also close it. This might be the case on other devices too.

Decompression on the Device is Cheap It is relatively cheap (time wise) to decompress data on the device, this means that having the server compress data before upload can be beneficial.

7 Future Work and Conclusion

7.1 Future Work

Since the API is at an early stage of development, there is still a lot of room for improvements. In this section we will discuss some of them and how they might be implemented.

7.1.1 Descriptions for SecureRecordStore entries

In section 4.5.3.2 we suggested that the encrypted data in the secure record stores could be given a description. This is to make them easier to identify for the end user without having to decrypt all the encrypted records. A very costly operation, as seen in chapter 5.

7.1.2 HTTPS support for SecureHttpConnection

The current `SecureHttpConnection` requires a `HttpConnection` as a parameter for its constructor. This means that any application that uses both the `SecureHttpConnection` and HTTPS for data transmission, would have to implement a special case for each of these. This can quite easily be remedied by supporting `HttpsConnection` to be passed in the `SecureHttpConnection` constructor as well. The type of connection can be detected inside the `SecureHttpConnection` object and any data written to it would just be sent as plain text if the connection is HTTPS. This way enabling or disabling HTTPS would be as easy as changing the URL.

7.1.3 Stronger Relationship Between Record Stores

As discussed in 4.5.3 it is not possible to extend the Java ME `RecordStore` class as this is a singleton created using the factory pattern. Instead we mimic the behavior of this class. The downside of this is that there is no relationship between the `SecureRecordStore` and `RecordStore` classes. If one wishes to swap one for the other this has to be done on the code level.

One way around this is to create a interface based on the `RecordStore` class and then have all the record stores implement this interface. The only problem then is that they are all created using the factory pattern, and as such another record store factory responsible for creating both the Java ME `RecordStore` as well as the secure versions is needed.

This would make swapping between different kinds of record stores without having to re-compile possible. The downside is increased code complexity.

7.1.4 Chunk Reading from `SecureHttpConnection`

As discussed in 4.6.2.1, because we calculate the HMAC of the entire encrypted message during transmission, we also have to read the entire message into memory on the device before we can verify that it has not been tampered with. For very large messages this could prove troublesome if the device does not have enough memory to keep the whole response in memory at once.

By sending the message in chunks at the time, and having a HMAC calculated for each chunk, this problem could be avoided. The client would only need to load a single chunk into memory at any given time to verify it's integrity and then it could be stored or read and discarded by the application. This would however add some overhead to the communication.

7.1.5 Dynamic JAD Hashing

As a part of the server authentication process, the JAD file is hashed and compared to the one located on the server. If the hashes do not match, it is assumed that it has been tampered with and it is therefore discarded. Since this functionality is already in place, it would be convenient to provide the programmer with the means to verify his or her own JAD attributes that are outside the JAR package. The best approach to this would be to sort any attributes names in the same way on the server and the client, otherwise the programmer would have to be very careful that the values are added in the same order on both the server and the client. Otherwise they will not produce the same hash. Sorting the list first would give the order list regardless or which order they are added in.

7.1.6 Store for Upload

In section the candidate proposes a way to store records so that they can be uploaded with minimal overhead for decryption and encryption. Since this has not been tested and is not a complete system it is proposed as future work.

7.1.7 Improve or Remake of the `SecureClient`

The back end controllers for the `SecureClient` are more or less the same as they were in the prototype. Though it works as intended, the system is overly complicated and could be made to better utilize new features such as the support classes for the `SecureHttpConnection`. This could also in a nice way expose some of the steps that make up the `SecureClient` to give better flexibility compared to the current all or nothing design.

7.1.8 Roles and Shared Storage

By not using the `UserKeyStore` and instead managing keys and stores manually, a programmer could introduce roles and shared record stores into their application. It would be possible to extend the `UserKeyStore` and introduce such features into the API.

7.2 Conclusion

The main objective for the work described in this thesis was to evolve the prototype implementation of the protocol into an easy to use API for making secure Java ME applications. Based on some working assumptions regarding what context the API would most likely be used in, we determined some criteria for the implementation.

We have discussed how the API works and why it is designed the way it based on benchmarks and the intended usage.

We will evaluate to what degree the criteria which the API should satisfy has been fulfilled.

The first criterion is that the API should be easy to use. Most of the classes that are exposed to the programmer are very similar to the Java ME counterparts in terms of methods and behavior. This means that any programmer who knows how to use the normal Java ME classes will also know how to use the secure ones. Furthermore, we have seen from the integration with openXdata that by simply changing some of the classes used in the existing MDCS code to the secure version, existing code can be secured given the secure classes have been initialized. By using the `UserKeyStore` or `SecureClient` this initialization should not be more than a few lines of code. So yes, the candidate would say that this criterion has been satisfied.

The second criterion is that the different functionalities should be decoupled and be flexible. Looking at the package overview diagram 4.1 we can see that the different packages are loosely coupled. In fact, disregarding the core packages which are used by most of the other classes in the system, the only inter-package dependency is from the `SecureClient` to the `communication` package. As far as flexibility is concerned, we have seen that the API supports multiple ways of using the different aspects of the system. Ranging from simply leveraging on the complete API through the `SecureClient` (meaning less work but also less control for the programmer), to the programmer having full control of every aspect of the application, including keys and data storage. The candidate finds it fair to say that this criterion is satisfied as well.

The third criterion is that the API should work on low-end devices. The API is implemented using Java ME and a configuration is supported by most java enabled devices. The user interface is easy to use even on small screens and areas such as the record store has been tested to find the best way of storing data through benchmarks. However, as discussed in 7.1.6, it would be possible to get noticeable gains as far as time spent ciphering data being uploaded is concerned by storing data in a format that allows for direct upload. Regarding cryptographic throughput while storing data on the device, little can be done in terms of the API design in order to improve this.

One aspect of performance which has not been discussed in depth in this thesis is alternative implementations of `CrytpoTools`. This might give performance benefits in multiple areas, but is as previously mentioned outside the scope of this document. Does the API, to a sufficient degree, take the low-end device requirements into consideration? The candidate finds it fair to say yes it does. However, for the `SecureHttpConnection` to be a more viable option, especially in areas where decent network connectivity is available (if connection is fast the cryptographic processes will be the bottleneck), a faster solution for uploading stored data is needed.

OpenXdata has shown interest in integrating the API with their existing client to, as a first step, secure data stored on the device. The candidate finds this to be an indication that the API holds potential and may be adopted by other systems in the future. Some work, especially actual field testing, remains though.

The candidate has gained a much better understanding of how mobile technologies work in general, but even more so how Java ME is designed and works. Based on the work presented the candidate feels that making APIs for the Java ME is not that different from making an API for any other Java platform. The only two real differences are that Java ME is more restricted in what one can and cannot do and that the system thread needs to be given special attention if a method does not return instantly.

All in all the project has given the candidate an insight into how working on larger projects with many different people differs from working on a small project in ones spare time, or on a school project with fellow students. Being part of writing and publicating research papers the candidate has a better understanding of how this type of research and publication is done.

A
Benchmark Extras

A.1 Additional `RecordStore` Benchmark Results

As stated in section 3.1 benchmarks on different update scenarios has been done. It can not be assumed that the data being updated always is the same size, and so below we will look at the results of a number of different update configurations.

Results Based on the results in tables A.1 and A.2 we can see that regardless of the update configuration, it will at worst be marginally slower than doing a write operation. Considering one would have to perform a delete prior to writing the data to avoid duplicates, in the end updating will always be faster than writing.

It is interesting to observe that updating with a marginally smaller data set is faster than updating with one of the same size as exists, and that updating with an additional 50% sized array is slower than twice as much data when the data sizes become big.

A.2 Compression Benchmark

We use JZlib [44] to do the compression and decompression, this is a java implementation of zlib [27]. Because of the way the DEFLATE [39] algorithm used by zlib works, the compression ratio is completely dependant on the input data. In our tests we will use

Data Size (bytes)	Update	Update-1%	Update-50%
32	0,6	0,3	0,5
64	0,7	0,5	0,6
128	0,7	0,7	0,6
256	0,8	0,6	0,6
512	0,8	0,7	0,6
1024	0,9	0,7	0,7
5120	1,2	1,0	1,0
10240	1,5	1,4	1,6
20480	2,3	2,4	2,8
25600	2,8	2,8	2,8

Table A.1: Benchmark results for different update scenarios. The percentage refers to the difference between the stored data and the new data that it is being updated with. -50% would mean that a record of 100 bytes is being updated with 50 bytes of data.

Data Size (bytes)	Write	Update+10%	Update+50%	Update+100%
32	1,5	0,7	0,8	1,5
64	1,4	0,6	1,4	1,5
128	1,5	0,5	1,5	1,5
256	1,6	0,8	1,4	1,4
512	1,8	2,0	1,5	1,8
1024	1,9	1,8	1,9	1,9
5120	2,6	2,4	2,4	2,5
10240	3,0	3,0	3,4	2,9
20480	4,4	4,2	4,7	4,2
25600	4,8	4,9	5,2	4,7

Table A.2: Benchmark results for different update scenarios. The percentage refers to the difference between the stored data and the new data that it is being updated with. +10% would mean that a record of 100 bytes is being updated with 110 bytes of data.

random data and as such the achieved compression ratio is not of any interest.

The compression benchmark is performed in the same way as the other benchmarks done. 10 data sets are generated, the min and max of each is discarded and the remaining is averaged.

Compression and Decompression Methods As with the encryption benchmarks, we supply the used methods since these are note a part of Java Me.

Listing A.1: Compression method.

```
/**
```

```
 * Compresses the input data and returns the compressed
 * byte array. Uses the default JZlib settings.
 */
public static byte[] compress(byte[] data) {
  baos = new ByteArrayOutputStream();
  try {
    zDefalte = new ZOutputStream(baos,JZlib.Z_DEFAULT_COMPRESSION);
    int i = 0;
    while (data.length-i > 2048) {
      zDefalte.write(data,i,2048);
      i+=2048;
    }
    zDefalte.write(data,i,data.length-i);
    zDefalte.finish();
    zDefalte.close();
    return baos.toByteArray();
  } catch (IOException e) {
    e.printStackTrace();
  }
  return null;
}
```

Listing A.2: Decompression method.

```
/**
 * Decompresses the input data and returns the original
 * byte array. Uses the default JZlib settings.
 */
public static byte[] decompress(byte[] data) {
  bais = new ByteArrayInputStream(data);
  try {
    zInflate = new ZInputStream(bais);
    baos = new ByteArrayOutputStream();
    int i;
    while ((i = zInflate.read(buff ,0,buff.length)) != -1) {
      baos.write(buff,0,i);
    }
    zInflate.close();
    return baos.toByteArray();
  } catch (IOException e) {
    e.printStackTrace();
  }
  return null;
}
```

Benchmark M - Compression This benchmark will determine how long it takes to compress a byte array of varying size and content.

Listing A.3: Benchmark for compression.

```
long result, total = 0;
for (int i = 0; i < itt;i++) {
  // generate test data of some size
  startClock();
  deflatedData = compress(dataToDeflate);
  result = stopClock();
  total += result;
}
return total;
```

Benchmark N - Decompression This benchmark will determine how long it takes to decompress a compressed byte array of varying size and content. The input will be the output from a compression process.

Listing A.4: Benchmark for decompression.

```
long result, total = 0;
for (int i = 0; i < itt;i++) {
  // generate test data of some size and compress this
  // dataToInflate will be this compressed data.
  startClock();
  inflatedData = decompress(dataToInflate);
  result = stopClock();
  total += result;
}
return total;
```

Bibliography

[1] About miktex. http://miktex.org/about. Accessed Aug 2011.

[2] Android. http://www.android.com/. Accessed Aug 2011.

[3] Apple - ios 5. http://www.apple.com/ios/. Accessed Feb 2012.

[4] Blackberry - official blackberry. http://us.blackberry.com/. Accessed Feb 2012.

[5] Essos | international symposium on engineering secure software and systems. http://distrinet.cs.kuleuven.be/events/essos/2012/. Accessed Aug 2011.

[6] J2me test suite homepage. http://j2metest.sourceforge.net/. Accessed Jun 2011.

[7] Latex - a document preparation system. http://www.latex-project.org/. Accessed Aug 2011.

[8] Legion of the bouncy castle. http://www.bouncycastle.org/. Accessed June 2011.

[9] Legion of the bouncy castle - faq. http://www.bouncycastle.org/wiki/display/JA1/Frequently+Asked+Questions. Accessed August 2011.

[10] The mhealth summit: Local and global converge. http://www.caroltorgan.com/mhealth-summit/. Accessed Feb 2012.

[11] Mobile data collection for the real world [mobenzi researcher]. http://www.mobenzi.com/researcher/. Accessed Nov 2011.

[12] openxdata. http://www.openxdata.org/. Accessed Jul 2011.

[13] Oracle java wireless client. http://www.oracle.com/technetwork/java/javame/javamobile/overview/getstarted/index.html. Accessed Jun 2011.

[14] Oracle java wireless client. http://eclipseme.org/docs/installation.html. Accessed Jun 2011.

[15] Owasp webscarab project. https://www.owasp.org/index.php/Category:OWASP_WebScarab_Project. Accessed Jun 2011.

[16] Subversion team provider for eclipse. http://subclipse.tigris.org/. Accessed Jun 2011.

[17] Tastephone - java.net. http://java.net/projects/tastephone. Accessed Jun 2011.

[18] Texlipse homepage - latex for eclipse. http://texlipse.sourceforge.net/. Accessed Aug 2011.

[19] Wikipedia: 3g. http://en.wikipedia.org/wiki/3G. Accessed Dec 2011.

[20] Wikipedia: Boilerplate code. http://en.wikipedia.org/wiki/Boilerplate_code. Accessed Aug 2011.

[21] Wikipedia: Enhanced data rates for gsm evolution. http://en.wikipedia.org/wiki/Enhanced_Data_Rates_for_GSM_Evolution. Accessed Dec 2011.

[22] Wikipedia: Hmac. http://en.wikipedia.org/wiki/HMAC. Accessed August 2011.

[23] Wirelessmoves: Gprs and edge multislot classes. http://mobilesociety.typepad.com/mobile_life/2007/04/gprs_and_edge_m.html. Accessed Dec 2011.

[24] Wireshark - go deep. http://www.wireshark.org/. Accessed Jun 2011.

[25] Youtube: openxdata launch - cu@school project, uganda. http://www.youtube.com/watch?v=ThOTDorm980. Accessed Jan 2012.

[26] Youtube: openxdata launch - mdr-tb treatment monitoring, pakistan. http://www.youtube.com/watch?v=1N8236ReWnM. Accessed Jan 2012.

[27] zlib home site. http://zlib.net/. Accessed Dec 2011.

[28] Joshua Bloch. How to design a good api and why it matters. Talk/video: http://www.infoq.com/presentations/effective-api-design. Accessed Jun 2011.

[29] Stefan Bodewig Conor MacNeill. Apache ant - welcome. http://ant.apache.org/. Accessed Sep 2011.

[30] Vital Wave Consulting. mhealth for development: The opportunity of mobile technology for healthcare in the developing world. http://www.vitalwaveconsulting.com/pdf/mHealth.pdf. Accessed Jul 2011.

[31] *Design Patterns, Elements of Reusable Object-Oriented Software*, pages 128–134. Addison-Wesley, second edition, 1995.

[32] Inc. Eclipse Foundation. Eclipse - the eclipse foundation open source community website. http://www.eclipse.org/. Accessed Jun 2012.

[33] EclipseME. Eclipseme home page. http://eclipseme.org/. Accessed June 2011.

[34] S. H. Gejibo J. Klungsøyr F. Mancini, K. A. Mughal. Adding security to mobile data collection. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6026793. Accessed Jun 2011.

[35] S. H. Gejibo J. Klungsøyr F. Mancini, K. A. Mughal. Securing mobile data collection. Unpublished paper.

[36] S. H. Gejibo R. A. B. Valvik J. Klungsøyr F. Mancini, K. A. Mughal. Challenges in implementing end-to-end secure protocol for java me-based mobile data collection in low-budget settings. Accepted for the International Symposium on Engineering Secure Software and Systems. ESSoS 12, http://distrinet.cs.kuleuven.be/events/essos/2012/.

[37] S. H. Gejibo R. A. B. Valvik J. Klungsøyr F. Mancini, K. A. Mughal. End-to-end secure protocol for mobile data collection systems in low-budget settings. Unpublished paper.

[38] S. H. Gejibo R. A. B. Valvik J. Klungsøyr F. Mancini, K. A. Mughal. On the security of mobile data collection systems in low-budget settings. Submitted for the International

Conference on Mobile Systems, Applications and Services. Mobisys 2012 http://www.sigmobile.org/mobisys/2012/cfp.html.

[39] Antaeus Feldspar. An explanation of the deflate algorithm. http://zlib.net/feldspar.html. Accessed Dec 2011.

[40] Eric Giguere. Databases and midp, part 1. http://developers.sun.com/mobility/midp/articles/databaserms/. Accessed August 2011.

[41] Network Working Group. Http over tls. http://www.ietf.org/rfc/rfc2818.txt. Accessed Jan 2012.

[42] Network Working Group. Hypertext transfer protocol – http/1.1. http://www.w3.org/Protocols/rfc2616/rfc2616.html. Accessed Jan 2012.

[43] Csaba Zainkó Géza Németh. Word unit based multilingual comparative analysis of text corpora. http://speechlab.tmit.bme.hu/publikaciok/page2035.pdf. Accessed Jan 2012.

[44] JCraft. Jzlib - zlib in pure java. http://www.jcraft.com/jzlib/. Accessed Dec 2011.

[45] Jonathan Knudsen. Networking, user experience, and threads. http://developers.sun.com/mobility/midp/articles/threading/. Accessed Jan 2012.

[46] Richard Marejka. Learning path: Midlet life cycle. http://developers.sun.com/mobility/learn/midp/lifecycle/#jadsnjars, 2005. December November 2011.

[47] Nokia. Nokia 2330 classic. http://www.developer.nokia.com/Devices/Device_specifications/2330_classic/. Accessed August 2011.

[48] Legion of the Bouncy Castle. Java docs: Class paddedbufferedblockcipher. http://www.bouncycastle.org/docs/docs1.4/org/bouncycastle/crypto/paddings/PaddedBufferedBlockCipher.html. Accessed August 2011.

[49] Oracle. Java doc: Interface httpconnection. http://docs.oracle.com/javame/config/cldc/ref-impl/midp2.0/jsr118/index.html. Accessed June 2011.

[50] Oracle. Java docs: Interface serializable. http://docs.oracle.com/javase/6/docs/api/java/io/Serializable.html. Accessed November 2011.

[51] Oracle. Java ee at a glance. http://www.oracle.com/technetwork/java/javaee/overview/index.html. Accessed Jan 2012.

[52] Oracle. Java me and java card technology. http://www.oracle.com/technetwork/java/javame/index.html. Accessed Jun 2012.

[53] Oracle. Java me sdk download. http://www.oracle.com/technetwork/java/javame/javamobile/download/sdk/index.html. Accessed Jun 2011.

[54] Oracle. Java se at a glance. http://www.oracle.com/technetwork/java/javase/overview/index.html. Accessed Jun 2012.

[55] Oracle. Sun java wireless toolkit 2.5.2_01 for cldc download. http://www.oracle.com/technetwork/java/download-135801.html. Accessed Aug 2011.

[56] Oracle. Security and trust services api for j2me (satsa). http://java.sun.com/

products/satsa/, 2006. Accessed November 2011.

[57] ProGuard. Proguard: Main. http://proguard.sourceforge.net/. Accessed November 2011.

[58] ProGuard. Proguard: Retrace usage. http://proguard.sourceforge.net/index.html#manual/retrace/usage.html. Accessed November 2011.

[59] Mansoor S.P. Sharif S.O. Performance analysis of stream and block cipher algorithms. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5578961. Accessed Jan 2012.

[60] Wikipedia. Feature phone. http://en.wikipedia.org/wiki/Feature_phone. Accessed Jan 2012.

[61] Wikipedia. Obfuscated code. http://en.wikipedia.org/wiki/Obfuscated_code. Accessed Jan 2012.