Master Thesis

Timed data deletion on mobile communication devices

Author(s):
Marforio, Claudio

Publication Date:
2010

Permanent Link:
https://doi.org/10.3929/ethz-a-006154349

Rights / License:
In Copyright - Non-Commercial Use Permitted
ETH ZÜRICH

MASTER THESIS

Timed Data Deletion on Mobile Communication Devices

by

Claudio Marforio

supervised by
Prof. Dr. Srdjan Capkun
Christina Pöpper

Department of Computer Science
System Security Group

8 February - 8 August 2010
“Against stupidity the gods themselves contend in vain.”

Isaac Asimov

“Mit der Dummheit kämpfen Götter selbst vergebens.”

Friedrich Schiller
Abstract

Master of Science

by Claudio Marforio

Forward secrecy under full compromise addresses one aspect of Information Security. Forward secrecy relates to protocol security and means that secret messages are not revealed to an attacker if the long-term keys of the communication partners fall into the attacker’s hands. Forward secrecy under full compromise tries to achieve the same property if all data on the devices of the communication partners is fully disclosed to a third party, either to an attacker or a lawful entity. In this case, a third party gets access to one or both of the communication devices at some time after the communication has taken place.

In this thesis, we analyze a protocol proposal that tries to achieve forward secrecy under full compromise. The main idea is to store the keys which are used to encrypt the message before its transmission on a separate device. By timely deleting these keys on the device the original exchanged information becomes unrecoverable after the specified time. We give a working implementation of the protocol and show its practical feasibility. Starting from this implementation and its analysis we identify the building blocks of the protocol and further discuss problems arising from their realization. We give possible solutions for each problem and conclude our work by pointing out in which research area more work is required to achieve the goals of the protocol.
Acknowledgements

First of all I would like to thank Prof. Dr. Srdjan Čapkun for presenting me with the opportunity to work on such an interesting topic.

I also wish to express my gratitude towards Christina Pöpper for introducing me to the problem and helping me throughout the development of this master thesis project.

I would like to thank Meaghan Jones for proof-reading this work.
# Contents

## Abstract

## Acknowledgements

### 1 Introduction

1.1 Motivation ..................................................... 1
1.2 Background .................................................... 2
1.3 Structure of this Document .................................... 3

### 2 Preliminaries

2.1 Forward Secrecy Under Full Compromise .......................... 4
2.2 Protocol Description ........................................... 5

### 3 Building Blocks

3.1 Secure Deletion ................................................ 7
3.2 Timely Deletion ................................................ 10
3.3 Protocol Extensions .......................................... 14

### 4 Design and Implementation

4.1 Setting ......................................................... 15
4.2 Android OS Overview .......................................... 16
4.2.1 Android OS Storage ......................................... 16
4.2.2 Android OS Alarms System .................................. 17
4.3 Design and UML ................................................. 17
4.3.1 UML Overview ............................................... 17
4.3.2 UML for R: PC Client ....................................... 18
4.3.3 UML for P: Android Device .................................. 18
4.4 Implementation Details ....................................... 19
4.4.1 Sender-receiver Communication Details ..................... 20
4.4.2 Receiver-Porter Device Communication Details ............ 23
4.4.3 Protocol Communication Details ............................ 23
4.4.4 Storage of Secrets ......................................... 24
4.4.5 Timely Deletion of Stored Secrets ......................... 25
4.4.6 Secure Delete on the Storage Device ...................... 26
4.4.7 Secure Delete on the Communication Devices ............ 27
4.5 Analysis of Implementation .................................. 28
## Contents

4.5.1 Storing and Retrieving ................................................. 28
4.5.2 Deletion ........................................................................ 28
4.5.3 Alarm Reset ..................................................................... 29
4.6 Implementation Facts .......................................................... 29
4.7 Known Problems .................................................................. 30

5 Conclusion ............................................................................. 32
5.1 Future Work ......................................................................... 33
5.2 Lessons Learned .................................................................... 33

List of Figures ........................................................................... 34

Bibliography ................................................................................ 35

A .................................................................................................. 37
A.1 Protocol Code ......................................................................... 37
A.2 Javascript Helper Functions .................................................... 43
A.3 Java Helper Functions Found in the CryptoHelper Class ....... 44
Chapter 1

Introduction

1.1 Motivation

Communication is found around us starting from animal life to human beings and resolves to the need of transmitting bits of information from one entity to another entity. Humanity has a peculiar need for some of this communication to be secure, as shown throughout history. Starting around 4500 years ago in Egypt and Mesopotamia, continuing with the Romans and the Greeks of Classical times, the need for secure communication has reached modern times and has seen an increased use. Starting from a written form, shown by Mesopotamian clay tablets to encrypt recipes, or most notably by Spartan and Roman soldiers to communicate privately, it has evolved with technology and spread in today’s heavily used data communication systems, Internet above all.

To enable communication in a secure way standards have been established, broken, and new ones proposed. Security has become more and more a topic of discussion with more and more services, needing and providing private information, becoming available to the user. An obvious example is money/information transactions where one user wants to transfer money or information to another user or service—such as a bank, an auction house, or a company.

Typical solutions for problems concerning secure communication take into account the possibility for an attacker to observe the communication between the two parties and prevent the attacker from gaining any possibility to discover the secret exchanged. This work focuses on ensuring the secrecy of the exchanged data even in the case that one or both ends of the communication are compromised after the communication has taken place.
Compromising a communication device could be seen either as an attack from a malicious third party or a lawful acquisition of the device by some officer. In either case, users may want that data remains private. If such confidentiality of the data is preserved, the protocol is said to enable *forward secrecy under full compromise*.

### 1.2 Background

*Forward secrecy under full compromise* has gained new interest in the information security field. The first proposal to solve this problem was given in 2005 [1]. The proposed solution uses a centralized trusted machine to generate and store keys used to encrypt data. Another proposal to solve this problem was presented in 2009 [2]. This work utilizes a decentralized key storage with a probabilistic deletion method. It has been proven to be vulnerable to attacks [3].

This thesis is based on the protocol proposed in [4] (Chapter 2). The main purpose of this thesis is to show the practical feasibility by a protocol implementation. Furthermore, this work wants to analyze into more detail the two prominent problems arising with *forward secrecy under full compromise*: **timely** and **secure deletion** of data.

The solution to the *forward secrecy under full compromise* analyzed throughout this work stores the keys used for encryption of data on a separate device, owned by one of the two parties of the communication, and timely deletes them when not needed anymore. This new approach works to overcome the two “drawbacks” present in [1, 2]. It does not use a separate trusted machine nor a probabilistic deletion system.

Given the protocol [4], our approach has been to start with a concrete instantiation through a test implementation. While working on the implementation, we discovered different implementation-specific problems and we tried to solve them. Investigating the solution for a specific problem in a given actual system, we also tried to abstract it to gain a better understanding of the key components of the protocol.

In some cases, the solution to the problem is available for existing systems, in others a solution could not be found and we could only propose a possible path to follow to reach a convincing solution. This proves that both **timely deletion** and **secure deletion** of data, two *building blocks* of the protocol, are not easy tasks to accomplish.
1.3 Structure of this Document

This document is organized as follows. Chapter 2 describes and explains the protocol proposal and enables a better understanding of the whole work by introducing the notation and general concepts used throughout the document. Chapter 3 goes into details about the secure and timely deletion problems. Implementation details are given in Chapter 4 where implementation related analysis of the protocol is also shown. Chapter 5 concludes the work and gives an overview for future work.
Chapter 2

Preliminaries

In this Chapter we will introduce the protocol that we analyzed and implemented. The protocol wants to achieve forward secrecy under full compromise by using a porter device which stores the mid-term key used for encryption of data. The protocol uses multiple DH key exchange protocol and relies on secure deletion and timely deletion of keys. First we introduce the concept of forward secrecy under full compromise, the description of the protocol follows. Analysis of the protocol building blocks is detailed in Chapter 3.

2.1 Forward Secrecy Under Full Compromise

Forward secrecy relates to protocol security and means that an attacker cannot decrypt previously encrypted messages exchanged by two communication parties even if their long-term keys are disclosed. Forward secrecy under full compromise extends the concept of forward secrecy to obtain the same goal even in the case that one or both of the communication devices fall into possession of an attacker or a lawful authority after encrypted communication has taken place.

Throughout our work we utilize a two-phased attacker model. In particular to guarantee forward secrecy under full compromise we set an expiration time $t_e$ after which data will be unrecoverable by any person, being it an attacker, a lawful party or the two communication parties themselves. The attacker model is a two-phased one:

- $t \leq t_e + \Delta_{\text{deletion}}$: Active external attacker. Modeled under the Dolev-Yao threat model, the attacker can overhear, intercept and synthesize any message.

- $t > t_e + \Delta_{\text{deletion}}$: Full compromise. The attacker has full access to the devices used by the two communication parties.
We define $\Delta_{\text{deletion}}$ as a given amount of time which is needed by any device to execute the deletion operation. Such time can be calculated preemptively by observing a given implementation. We give an estimate when analyzing our implementation of the protocol in Chapter 4.

It is important to notice that before the expiration time the attacker cannot alter in any way the communication parties’ devices or the data that is stored on them.

We stress that an attacker can also be a lawful entity because our attacker model, being stronger than the Dolev-Yao model, might look improbable to someone. Nonetheless in the case of a lawful entity we can imagine the possibility that they obtain, through a lawful verdict, both the control over the network, through the ISPs, and of the devices used by the communication parties. \textit{Forward secrecy under full compromise} wants to guarantee the confidentiality of data exchanged between two parties even after such an event.

### 2.2 Protocol Description

<table>
<thead>
<tr>
<th>$S$ (sender)</th>
<th>$DY$ channel 1</th>
<th>$R$ (receiver)</th>
<th>$DY$ channel 2</th>
<th>$P$ (porter)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pick $r_S$</td>
<td>compute $g^{r_S}$</td>
<td>pick $r'_R$, compute $g^{r'_R}$</td>
<td>$A_{\mathcal{S}}(m_1,g^{r'_S},t_L)$</td>
<td>pick $r_P$, compute $g^{r_P}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$A_{\mathcal{R}}(m_2,g^{r'_R})$</td>
<td></td>
<td>$L = g^{r_P t_L}$, delete $r_P$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pick $r'_R$, compute $g^{r'_R}$</td>
<td></td>
<td>$K = [K]_{L^{-1}}$, delete $L$</td>
</tr>
<tr>
<td></td>
<td>$K = g^{r_S r_S}$</td>
<td>$K = g^{r'_S r_S}$, $L = g^{r'_R r_R}$</td>
<td>$A_{\mathcal{R}}(m_5,g^{r'R_R},t_L)$</td>
<td></td>
</tr>
<tr>
<td>delete $r_S$</td>
<td>delete $K$, $L$, $r_R$, $r'_R$</td>
<td>delete $r'_R$, compute $g^{r'_R}$</td>
<td>$A_{\mathcal{R}}(m_4,S,t_L,[K,4])$</td>
<td></td>
</tr>
<tr>
<td>$m$, $[m]_K$</td>
<td>$A_{\mathcal{S}}(m_6,[m,6,1],t_L)$</td>
<td>$A_{\mathcal{S}}(m_7,g^{r'<em>{R'} R'},[K]</em>{L'})$</td>
<td>$A_{\mathcal{P}}(m_8,g^{r'<em>R R'},[K]</em>{L'})$</td>
<td>$L' = g^{r'_R R'}$, delete $r'_R$</td>
</tr>
<tr>
<td>delete $m$, $K$</td>
<td>pick $r'_R$, compute $g^{r'_R}$</td>
<td>pick $r'_P$, compute $g^{r'_P}$</td>
<td>delete $L'$</td>
<td>At time $t_L$, delete $(S_{L,4},K)$</td>
</tr>
</tbody>
</table>

**Figure 2.1:** Implemented Protocol. The established symmetric key $K$ is used to encrypt the time-limited data $m$. All messages are authenticated, denoted by the authentication function $A_X(\cdot)$ (representing the digital signature of principal $X$ over the function input). The \textit{square brackets} notation represents symmetric encryption of data, $[K]_{L}$ means data $K$ encrypted with key $L$. The inverse function $[K]_{L^{-1}}$ represents the decryption function using key $L$ to decrypt $K$. Where the delete keyword is used the protocol expects a \textit{secure deletion} operation to be carried out.

Figure 2.1 (p. 5) shows the protocol proposed in \cite{4} to achieve \textit{Forward secrecy under full compromise}. 

The main idea of the protocol is to have three communicating parties, or devices. The sender $S$ has a device (i.e. a PC) and wants to transmit some data $m$ to a receiver $R$. $R$, itself, has a receiver device, (i.e. a PC) and owns a separate device $P$ which could be a mobile phone or a specifically tailored device. The communication channels ($DY\text{channel1}$ and $DY\text{channel2}$) are subject to attacks under the Dolev-Yao threat model, where an attacker can overhear, intercept and synthesize any message.

The protocol is started by $S$ which begins a first key establishment session and defines the expiration time ($t_e$) of the generated key and therefore of the encrypted data. $R$ during the key establishment session communicates with $P$ to store the generated key $K$.

Once $K$ has been generated (and stored on $P$) $R$ can conclude the key establishment session with $S$ which is now able to send an encrypted, time-limited message $m$ to $R$. $R$ is going to be able to decrypt $m$ only until the previously defined time $t_e$ when its porter device $P$ will delete the stored key $K$. To decrypt $m$, $R$ will fetch $K$ from porter device $P$. Both storage and retrieval operations, which result in transmitting key $K$ from the two devices, establish fresh session keys to ensure the secrecy of the exchanged information.

All the messages exchanged by the 3 parties are authenticated by a function $A_X(\cdot)$ that uses a pre-shared key between the devices.

The keys used throughout the whole protocol are securely deleted immediately after their use and they are not needed anymore. $K$, on the other hand, is only stored on the porter device $P$ and is ensured to be securely deleted at time $t_e$.

The proposed protocol uses the DH key exchange protocol in multiple instances to have short-term session keys to send encrypted data between two parties. It is important, after a successful transmission of the encrypted data, that the data is securely deleted from the devices. Furthermore the mid-term session key $K$ needs to be deleted from $P$ in a timely fashion. These are the two challenging problems of the protocol and are further analyzed in the following chapter.
Chapter 3

Building Blocks

As we have described, the proposed protocol needs, at its core, two building blocks: secure deletion and timely deletion. In this chapter we analyze both of them and we also present an extension to the protocol to improve its usability.

3.1 Secure Deletion

With secure deletion we indicate the complete removal of data, rendering it un retrievable, from a storage system. For example if someone writes his name on a file we say that the file is securely deleted if, after a removal operation, it is impossible for anyone, with any tool, to retrieve the contents of that file, notably the previously written name.

Secure deletion, both in the scientific community and in governmental institutions [7–10], is ensured in two ways:

- **Overwriting** data before deletion. With a variable number of overwrites depending on the attacker model one wants to protect against.

- **Encrypting** data. And the issue of secure deletion would boil down to securely deleting the keys used to encrypt the data. In the case where the keys are stored on the same storage system of the encrypted data this solution would end up requiring some form of overwriting of the keys before deletion.

At the moment of writing the government’s own NIST and NISPOM specification for secure deletion of data requires that data is to be overwritten at least 3 times [7, 10]. Different opinions are available in the scientific community with respect to how many times data should be overwritten in order to prevent any recovery. It is accepted by a
widespread number of researchers that even a single overwrite of data makes the data unrecoverable by softwares methods \[7\]. At the same time data could still be recovered by using advanced techniques even after several overwrites with a decreasing probability the higher number of overwrites before deletion has been made \[8\]. Such proposals, being made over magnetic-media storage systems (i.e. magnetic hard drives), still hold true also for flash based storage systems (i.e. Random-Access Memory) \[8\].

On top of a direct write or delete operation to a storage device there are, usually, operating systems interfacing to the storage device through a filesystem. Latest filesystems (i.e. Ext3, Ext4, HFS+, YAFFS and YAFFS2, . . . ) moved to a different number of journaled solutions, mainly to speed up recovery time after a system crash. There are three journaling types:

- **Journal** all data and metadata logged to the journal
- **Ordered** data gets written first to the disk and only the changes to metadata are logged to the journal. This usually is also the default type used in journaled filesystems
- **Writeback** only the changes to metadata are logged to the journal

The main idea behind a journaled filesystem is to store some information, either metadata (file name, ownership, flags, . . . ), data, or both, to a journal. The journal is usually kept on the same disk (but can also be stored on a different one, for instance). While every write operation will be slower (two writes would be needed rather than only one: one actual data write and one journal write), in the case of a system crash, to recover the filesystem to a working state, it will be enough to go through the journal rather than the whole filesystem.

Obviously such widespread use of journaled filesystems renders secure deletion harder to achieve because data may be stored in multiple places throughout the disk, especially when overwriting the file multiple times as a standard way to securely delete files. Patches to the popular ext2, ext3 and ext4 filesystems \[7, 11\] have been provided and analyzed in the past although they have never been merged into the code of the filesystems.

This is even worse, on mobile devices that use the YAFFS or YAFFS2 \[12\] filesystem. YAFFS2 is a true log structured filesystem meaning that each write operation is stored on the storage medium subsequently without any writes on a previous block. As free

---

\[^{1}\] names might differ from one implementation to the other, the ones highlighted here are used in the Ext3 and Ext4 implementations
blocks are consumed, the whole storage medium is used in a circular buffer manner with subsequent writes starting at the first blocks taking up the space freed.

It is obvious that overwriting before deleting would not provide any secure deletion because the original data would be kept on the storage medium until the capacity would be filled and the data would be overwritten by some other data. An advisable way to tackle the secure deletion problem for true log structured filesystems is that to erase a block once it is marked as deleted instead of waiting for that block to be needed again to first erase it and then rewrite it with some new content. With this solution we would not violate the rationale behind the filesystem, in particular, that data should never be written to a previous block unless no available space is found.

We try to visualize the different operations that are happening when a deletion instruction is issued for a true log structured filesystem. Please note that this is just a high-level view of the deletion operation and does not take into consideration all the implementation details. On the left the normal operation, on the right the devised solution for secure deletion.

1. delete block $x$
2. a new block is written to mark block $x$ as deleted
3. subsequent write/delete operations fill the capacity of the disk
   → space needs to be freed
   → blocks marked as deleted get erased
4. block $x$ gets erased and subsequently overwritten
5. . . .

1. delete block $x$
2. a new block is written to mark block $x$ as deleted
3. immediately erase block $x$
4. subsequent write/delete operations fill the capacity of the disk
   → space needs to be freed
   → blocks are already erased
5. block $x$ gets overwritten
6. . . .

This change would ensure that erasure happens immediately after a deletion operation. It would render the deletion operation slower, but it will ensure secure deletion of data after a deletion operation. This is the case if the erasure operation also overwrites the data accordingly to the attacker model taken into consideration. Furthermore we observe that the proposed deletion model overhead can be lowered if the filesystem allows for differentiation of blocks. In particular only blocks marked for immediate erasure, such as those containing confidential data, would need to use our proposed method of immediate erasure, while the majority of blocks would follow the normal deletion method.
Regarding **secure deletion** both general filesystem and specialized filesystem considerations are relevant to our protocol because we rely on secure deletion both on the PCs of S and R and on the porter device P, in the case that a general mobile phone is used.

### 3.2 Timely Deletion

With **timely deletion** we indicate the time precision with which a given secret is deleted from a system independent of the state of the system. In particular we focus on understanding what are the different causes for a deletion event not to happen at the desired point in time and propose different solutions to the problems. We immediately state that we have to accept as **timely deletion** any deletion that occurs within a delta \( \Delta_{\text{deletion}} \) after \( t_e \), the ideal exact time when data should not be available anymore. In particular we compute this \( \Delta_{\text{deletion}} \) as the time it takes to securely delete a key after receiving the command to do so. Refer to Chapter 4 for the analysis of the implemented protocol that shows the magnitude of \( \Delta_{\text{deletion}} \) for our current implementation.

Throughout this chapter we refer to “deletion timer” as a timer that would trigger a deletion of a given file from a device when it expires. A “deletion timer” can be visualized as an actual timer, a *NIX cron job or as an Alarm specifically for the Android OS.

We start by defining two general states in which any porter device, needed for the operation of the described protocol, could be:

- **Operational** meaning that the device is turned on and initialized. In particular a device in this state is capable of interacting with other devices through some form of communication (GSM, WiFi, Bluetooth, . . . ). We consider that a device in this state can also execute instructions and has a notion of time.

- **Non-operational** meaning that the device is turned off or otherwise not able to communicate and to execute instructions.

We also take for granted that a device has a clock. Furthermore that this clock gives the device a notion of time either by running while the device is in a Non-operational state or through synchronizing with an external service when the device enters an Operational state. Any device that does not have a clock is not suitable to use in the proposed protocol. A different protocol would need to be devised such that the porter device could trigger the deletion of the stored keys through an external input.

Clearly, to accept an incoming communication, for example to store a secret, the porter device needs to be in the Operational state. We will analyze, as we did throughout the
whole work, the case such that the porter device is not attached directly to the receiver R. This is the case in which the communication channels are defined by a DY model.

When $S$ sends a timestamp $t_e$, marking the time at which data should be deleted, it could do so in two different ways:

- an offset computed as $t_e - t_{\text{now}}$
- an absolute timestamp as the one defined by unixtime\footnote{the number of seconds elapsed since midnight proleptic Coordinated Universal Time (UTC) of January 1, 1970, not counting leap seconds}

In both cases the two devices, the sender and the porter device, would need to have a common notion of time. In particular three scenarios could happen:

1. $S$ and $P$ reside in different timezones
2. $S$ and $P$ have a different time
3. a combination of both

In case (1) the easiest solution would be that of sending a timezone offset with regard to UTC along with each message that carries the $t_e$ information. The receiver of the message would then compare his timezone with UTC and apply the correct changes to the received $t_e$ to set up proper timely deletion. Similarly another option is to have $t_e$ always expressed in UTC time and set the deletion timers accordingly. Case (2) is more complex, and we conclude that the best practice would be that of running a time synchronization protocol \cite{13} between $S$, $R$ and $P$ before running our protocol to ensure that all clocks are synchronized with a given margin of error below which we still consider deletion of secrets to happen in a timely fashion. Case (3) is solved by combining the solutions to case (1) and (2).

Abstracting the timely deletion problem away from the protocol in which it is used, and considering only one device in which we declare a given point in time in the future $t_e$ at which to delete a secret still requires some thoughts. In particular we still need to consider the problems that could arise after having set $t_e$.

Four possible problems could arise:

1. between $t_{\text{now}}$ and $t_e$ the device goes from an operational state to a non-operational state
2. between $t_{\text{now}}$ and $t_e$ the device goes from an operational state to a non-operational state and back to an operational state.

3. between $t_{\text{now}}$ and $t_e$ the device changes time because of a change in the timezone. This could happen in the case that the device is time-synchronized across the internet or through the mobile network (operator changes).

4. between $t_{\text{now}}$ and $t_e$ the device time is changed manually, potentially by an attacker.

We now analyze these problems and suggest solutions to solve them.

1. What happens is that since $t_e$ is already passed when the device goes back to an operational state the secret is still stored at a point in time when it should not be stored. The best solution is to be able to perform a clean up operation at boot time so that all secrets that should have been deleted while the device was in a non-operational state are promptly deleted. Furthermore this operation should be handled in a blocking way so that other operations cannot read the secrets while they are deleted.

2. The solution to this case is fairly straightforward. Whenever a device goes back to an operational state a reset timers operation should be carried out to set up new timers to properly delete the secrets at $t_e$.

3. The solution to this case is handled in the same way as (4) where a change would be of either 30 or 60 minutes.

4. The solution to this case is straightforward but not always applicable (See Chapter 4 for implementation details). In particular we notice three possibilities:

   (a) the time change operation carries the offset of how much the time has changed

   (b) the time change operation does not carry the offset of how much the time has changed

   (c) the time change operation is not noticed by the system

In case (a) it is possible to adjust the deletion timers accordingly. In case (b), on the other hand, the best that one could do is to keep the timers unchanged and notify the user that the time has changed and enable him to operate manually on the deletion operation for each secret affected by the change. Case (c) is the one that poses the most threats. It is possible that an attacker finds a way such that changing the time is not “noticed” by the system. We cannot propose a solution to this latest case but we also notice that any device that is required to carry out a timely deletion operation and in any way is not able to “notice” a time change can be considered either malfunctioning or not suited to accomplish this task.
We note that the problems enumerated above are true for both solutions, whether setting the future time at which deletion should occur as an offset or as an absolute timestamp. Anyway we note that storing the absolute timestamp along with the secret is the best practice especially for the cases when the device will enter a non-operational state, in order to be able to reset the deletion timers accordingly when the device will go back to an operational state.

We now analyze different kind of attacks possible with respect to the depicted timely deletion scenarios. We distinguish two different kind of attacks:

1. **remote** where the attacker does not have the device in his possession
2. **local** where the attacker has the device in his possession or has an installment on the device, such as an application running on the device

In (1) we take into consideration the possibility that the attacker can change the time of the mobile device by impersonating a time synchronization server or by impersonating a cellular service provider and sending a time update to the device. In both cases the attacker can only change the time on the mobile device. Time changes have already been covered for possible solutions to them, so we consider this case to fall into the same category as the ones described previously.

Case (2) is not a possible attack given the attacker model that we defined. Just for completeness and to discuss timely deletion for a general case rather than the protocol instantiation we take into consideration the possibility that the attacker either has the device in his possession or that he has a malicious software installed on the device. In the first case the attacker can simply dump the secret data to another storage medium before timely deletion occurs, making the deletion useless. In the second case the malicious application could interact with the system in different harmful ways in relation to the timely deletion problem. Apart from the obvious time changing possibilities, which we already covered both regarding the problems they arise and the different solutions, a malicious application running on a mobile device, could pose different threats depending on the security enforced or not by the device operating system:

- direct access to protected data before deletion
- suspension, reset or deletion of deletion timers
- interception of a deletion request

As a conclusion: in our analyzed implementation the underlaying operating system prevents most of the listed malicious activities, different implementations for different
devices should take into account such problems when designing an implementation of the protocol to make sure that attacks targeting the timely deletion of keys would not succeed. In particular we described how to handle the time changing problem while the device is in an operational state.

3.3 Protocol Extensions

We found two limitations of the protocol for a real life scenario. In particular we analyze the case where $S$ and $R$ want to encrypt messages sent over e-mail, as also is the case for our implementation (see Chapter 4).

The first limitation comes from the immediate deletion of $K$ by $S$ after encrypting the message. With this enforcement in place, in case that $S$ wants to send something to $R$, in a given time $t_i$ which is before the expiration time $t_e$, $S$ would need to start the protocol all over again.

The second limitation comes from the fact that the protocol is one way only. In the e-mail case it is not improbable that $R$ would send back a message to $S$. If $R$ would want the reply to have the same time limitation (i.e. the data should be accessible only before $t_e$), it would still need to start the same protocol reversed where $R$ would be $S$ and vice-versa.

A suggested solution to both limitations is to ensure that $S$ has a similar porter device, $P_S$, to the one of $R$, $P$. The mid-term key, $K$, could then be stored on $P_S$ following the same protocol used to store it on $P$, and re-used until the expiration of time $t_e$ when the key would be deleted both from $P_S$ and $P$. In such a way data exchanged back and forth from the two communication parties will remain confidential and not accessible after $t_e$. 
Chapter 4

Design and Implementation

4.1 Setting

The implementation of the protocol described in [4] and shown in Figure 2.1 (p. 5) has been devised with the following abstract setting in mind:

- $S$ (sender): a Personal Computer
- $R$ (receiver): a Personal Computer
- $P$ (porter): an Android [14] bluetooth-enabled phone

In particular the testbed settings for each component of the protocol

$S$ an Apple MacBook Pro running the latest patched operating system: Mac OS X 10.6.4 with Thunderbird 3.0.4 and the Java plugin or a PC running the latest patched Ubuntu installation 10.04 or Fedora 10 with Thunderbird 3.0.4 and the SUN Java plugin

$R$ an Apple MacBook Pro running the latest patched operating system: Mac OS X 10.6.3 with Thunderbird and the SUN Java plugin

$P$ a NexusOne [16] running the latest patched Android OS: firmware 2.1-update1, Kernel: 2.6.29-01117-g4bc62c2 android-build@apa26 #1

An overview of the communication happening in the protocol between the three parties can be seen in Figure 4.1 (p. 16).
4.2 Android OS Overview

We decided to provide our implementation using as a porter device a smartphone. We used, in particular, a NexusOne running the Android OS.

Android is an operating system especially written for mobile devices such as mobile phones, tablet computers or netbooks. It is developed by Google and is based on the Linux kernel. The operating system software stack consists of Java applications running using the Java core libraries on top of the Dalvik virtual machine.

Relevant for this work are the storage capability of the Android OS and its Alarms system. We will introduce them and later on describe them in more details, respectively in Section 4.4.4 (p. 24) and Section 4.4.5 (p. 25).

4.2.1 Android OS Storage

An Android phone can have two different storage media:

- Internal flash memory, usually hardwired on the phone itself. This medium is shared also to store the applications themselves. On the NexusOne this amounts to 512MB.
- External memory, usually in the format of a removable flash card.
The internal flash memory is the default storage used to store applications code and data, but each application can also decide to store data on the external memory. This is the case, for example, with the camera application that stores its pictures on the external memory to have more free space on the internal flash memory. The Android operating system can enforce the ownership of each saved file to the application writing it.

### 4.2.2 Android OS Alarms System

To be able to delete a given secret at a future time $t_e$ we exploit the fact that Android enables multitasking and in particular the Alarms mechanism offered by the Android OS. This mechanism allows one application installed on the device to register for a particular Alarm ID (a String) that will be fired by the OS at a given time. It is then possible to execute any part of code upon receiving the Alarm from the OS.

Alarms are broadcasted by the OS to all running applications, only the ones that are listening for them will act upon receiving one.

### 4.3 Design and UML

#### 4.3.1 UML Overview

In this section we show the UML diagram for the code implementing the protocol described throughout. Figure 4.2 (p. 17) shows the overview of the implementation system. In particular it highlights the two classes shared between the code running on $P$ (the Android phone) and on $R$ (the PC client).

![Diagram](image)

**Figure 4.2:** Overview of the system, on the left the Android OS classes, on the right the PC client ones and in the middle the shared classes

Such classes (Figure 4.3 p. 18) implement common functionalities needed on both the device and the PC. In particular the class CryptoHelper provides methods to encrypt
or decrypt a given payload, package different messages, and extract specific fields from messages. Some interesting methods of the CryptoHelper class can be found in the Appendix, in particular Code A.6, A.7, A.8, A.9 and A.10 (p. 44 and subsequent). The class SharedConstants, as the name suggests, is used to share constants that are used throughout the whole system in order to have just a single place where to store and edit them.

<table>
<thead>
<tr>
<th>CryptoHelper</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte[] createMessageToSend(byte[] mID, byte[] key1, byte[] key2, long time, long S, byte[] IV, byte[] encryptedPayload)</td>
</tr>
<tr>
<td>byte[] checkMsgSignature(byte[] msg)</td>
</tr>
<tr>
<td>byte[] signMessage(byte[] msg)</td>
</tr>
<tr>
<td>byte[] long2byte(long l)</td>
</tr>
<tr>
<td>long byte2long(byte[] b)</td>
</tr>
<tr>
<td>byte[] generateRandomIV()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SharedConstants</th>
</tr>
</thead>
</table>

**Figure 4.3:** The two classes in ch.ethz.maclaudi.shared that are shared between the Android code and the PC code

### 4.3.2 UML for \( R \): PC Client

On the PC the code is structured into three packages:

- **ch.ethz.maclaudi** which provides the main classes with the functionality of the client \( R \) described in the protocol
- **ch.ethz.maclaudi.bt** which provides all classes needed to discover bluetooth devices and services
- **ch.ethz.maclaudi.ui** which provides all classes related to the GUI of the program

Figure 4.4 (p. 19) shows the overall structure of packages with a more detailed view on the main class ProtocolWorker. We show the relevant forwardBytesToProtocol method in the Appendix, Code A.2 (p. 38)

### 4.3.3 UML for \( P \): Android Device

On the Android device the code is structured following the general structure of typical Android applications.

- **SecureDeleteActivity** is the main class that displays messages to the user with some information on the protocol run.
• **BootupReceiver, DeleteAlarmReceiver** are the two classes that extend the Android OS `BroadcastReceiver`:
  
  – **BootupReceiver** registers itself to listen for boot completed events broadcasted by the Android OS and spawns a `ResetAlarmService` which checks all the stored keys for their expiration times
  
  – **DeleteAlarmReceiver** registers itself listening for any application-generated event signaling the expiration of a given key $K_i$

• **SecureDeleteService** is the main service which is spawned by the `SecureDeleteActivity` instance and handles bluetooth connections and communication.

• **SecureDelete** is an utility class that implements the method (currently overwriting 5 times each key before deletion) for secure deletion

Figure 4.4 (p. 20) shows the class diagram for the classes running on the pc client.

We show the relevant `forwardBytesToProtocol` method in the Appendix, Code A.3 (p. 40).

### 4.4 Implementation Details

The protocol implementation is achieved with a Thunderbird extension for the communication between $S$ and $R$. On $R$ we implemented a Java application that interfaces with Thunderbird and a running service on $P$ through bluetooth communication.

An overview of the protocol with particular focus on used channels between protocol parties is shown in Figure 4.6 (p. 20).
4.4.1 Sender-receiver Communication Details

Communication between the Sender $S$ and the Receiver $R$ is achieved through e-mail. The protocol is completely integrated in the Thunderbird e-mail client with an extension. Since the extension is using Java for its inner cryptography tasks it can work starting from Thunderbird 3.0 or higher that include Java integration to the e-mail client.

Thunderbird extensions are developed using the JavaScript language. Our extension, because of the needs of cryptography and bluetooth communication uses a Java implementation to achieve so. Figure 4.7 (p. 21) shows an overview of our Thunderbird extension ecosystem. In particular we see the interaction between the four parts of the extension.

At the core a JavaScript implementation that integrates the extension within the Thunderbird e-mail client. It provides buttons to start the protocol or check each SecureDelete
message. Also it automatically sends protocol messages that do not require further interaction by the user such as SDM1 and SDM5.

The JavaScript core extension interacts through bash scripts to two external programs. The first one is a Java implementation that uses the bluetooth bluecove library to interact with the bluetooth enabled porter device. The second one is the *NIX utility srm used to securely delete files on the sender $S$ and receiver $R$ machines.

Finally the JavaScript extensions makes use of the LiveConnect [19] utility to call directly the implemented Java crypto library that resembles the one used in the Java program and the Java implementation on the device.

![Diagram of SecureDelete Thunderbird extension overview](image)

**Figure 4.7:** SecureDelete Thunderbird extension overview, all elements are packed within the extension. In the dashed box is everything that works within Thunderbird, external tools are invoked but are actually other programs.

Generally the protocol begins with a user ($S$) starting the protocol specifying a $t_e$ date, using a calendar and time-picker, and a recipient ($R$) e-mail, as shown in Figure 4.8 (p. 22). A message window pops up with a SecureDelete block and the e-mail is sent automatically to the recipient. This message has as subject line SDM1.

A generated message will have a special block that is recognized by the extension and dealt with depending on the message type. An example message block is shown in Figure 4.9 (p. 22). Furthermore data contained between the starting tag (+++++ SECUREDELETE +++++) and the ending tag (===== SECUREDELETE =====) is Base64 encoded as with all the other data sent throughout the protocol.
Chapter 4. Design and Implementation

Upon receiving the starting message SDM1 for the protocol, $R$ is prompted to start communication to the Android device on which the SecureDelete service is running and upon successful storage of the key a newly created message is generated in Thunderbird and automatically sent back to the sender $S$. This message has as subject line SDM5.

Upon receiving SDM5, $S$ is prompted with a message compose window where $t_e$ is reminded again and the user can type a message. The message gets encrypted using the right key upon sending, the key is subsequently securely deleted from the hard-drive and the message is only stored in encrypted form. It has to be noted that only the body of the message is encrypted.

Upon receiving an encrypted message $R$'s Thunderbird client will start again automatically a Java client to retrieve the key for decryption from the mobile device. Once the key is retrieved it is used immediately to decrypt the message which is shown into a separate window. The decrypted message is therefore never stored on disk. $R$ will still be able to decrypt the encrypted message later on before the expiration of the key.
4.4.2 Receiver-Porter Device Communication Details

Communication between $R$ and $P$ is achieved via bluetooth using, on $R$, the Java bluecove library [20]. Encrypted data is also encoded using Base64 encoding [21] to prevent any problem during transmission, notably that some binary data representation when encrypted could cause an end of transmission for the bluetooth library while sending data. Furthermore all messages in the protocol are authenticated using a 32-byte HMACSHA with a 224-bit key that is pre shared between the laptop and the mobile phone.

Sending data over an RFCOMM channel over bluetooth requires that data should be sent in chunks of size smaller than 1024 bytes. The case that more than 1024 bytes needs to be sent over bluetooth may only occur in the $P \to R$ direction when sending message $M_3$ and $M_8$, the partial message is stored on $R$ and successfully received only if the next bluetooth communication carries the second part of that message.

Successful reception of a message is ensured by checking the message signature against the content of the message.

4.4.3 Protocol Communication Details

A packet sent throughout the protocol has the general structure depicted in Figure 4.10 (p. 23)

<table>
<thead>
<tr>
<th>MAC SIGNATURE</th>
<th>ID</th>
<th>CONTENTS</th>
</tr>
</thead>
</table>

**Figure 4.10:** A general packet structure

Detailed packet structures are found in Figure 4.11 (p. 24) where all messages as per design and implementation are shown.

The Android OS already features the bouncycastle crypto library [22] which is also used on the $R$ client implementation. The cryptographic details for the implementation are:

- **DH Key size:** 1024
- **Symmetric encryption of secret ($K$):** AES with CBC and PKCS5 Padding. The key is generated with the SHA-256 algorithm of the newly created DH-Key ($L$ or $L'$, depending on the protocol stage)
- **Message authentication:** HmacSHA256 with a pre-shared secret of 224-bit
Chapter 4. Design and Implementation

4.4.4 Storage of Secrets

We identify three alternatives for storing the keys on the device, in particular:

1. Storing keys in an SQLite database
2. Storing all keys in a file
3. Storing each key in a separate file

Each solution has advantages and disadvantages which are now analyzed. Solution (1) has the advantage of fast lookups of a given stored key at retrieval time given an indexed column is used for lookups. As a drawback, on the other hand, using SQLite would result in writing all the keys to a single file (the sqlite.db database file). Solution (2) has the advantage that only one file is created and keys are appended to it (when storing) and truncated from it (when retrieving). The biggest disadvantage of this approach resides in seeking in the file for key retrieval and storage, which could become an expensive operation. Solution (3) has the advantage of ease of implementation, fast storage and retrieval operations without dealing with appending or truncating a file. Furthermore the Android OS makes sure that files created within an application, with the appropriate flag, are accessible only by that application and not by others’ (an attacker application, for instance) applications.
We selected solution (3) and tested it in the current implementation and we are using the following call

```java
Context c = getApplicationContext();
c.openFileOutput(fileName, Context.MODE_PRIVATE);
```

to make sure that created files are accessible only by our application.

### 4.4.5 Timely Deletion of Stored Secrets

As the protocol requires that a stored secret $K_i$ is safely deleted at a given time $t_{ie}$ timely deletion scheduling is important. To achieve such a scheduling on the device the *Alarm* concept of the Android OS has been exploited. In particular when a secret $K_i$ is received at a time $t_i < t_{ie}$ the secret is stored on the internal memory of the device in a file named according to the sender $S_i$ and the time $t_{ie}$. The naming scheme follows the idea that it is unfeasible to receive two messages from the same user set to expire at exactly (millisecond precision) the same time. Therefore the files are named to: 

```
Hash(sender_email) time,
```

concretely

```
Hash(S_i) t_{ie}.
```

In particular the `Hash()` function takes a `String` and returns a `long`. This is done to be able to know a priori the size in bytes of $S$ transmitted in the packets. At the same time that the secret is stored on the internal storage an *Alarm* is set for time $t_{ie}$. At time $t_{ie}$ the alarm sets off, the application running on the device receives it and promptly deletes the file containing secret $K_i$.

If at time $t_i < t_{ii} < t_{ie}$, $R$ requests the secret $K_i$ the application retrieves it from the file and sends it over.

An overview of the Alarms system offered by the Android OS is presented in Figure 4.12 (p. 26) where the specific described case is shown.

Since the *Alarm* system given by the Android OS is reset at device reboot, to ensure the timely deletion of the stored secrets, upon device boot-up a background service is started that sets up *Alarms* for each stored secret present on the device. To achieve this we listen for boot completed events, in particular for the

```
android.intent.action.BOOT_COMPLETED
```

intent that signals a successful boot of the Android OS. For this reason it is important to store the absolute timestamp $t_{ie}$ in order to reset the alarms for the correct time.
Figure 4.12: Overview of the Alarms system offered by the Android OS.

Analysis of the Alarm system shows that on average the desired operation is run roughly 0.581633 ms after the reception of the alarm.

Such implementation grants security of the stored data as long as the Android OS security is not compromised. In particular we depend on the security granted by the Android OS concerning the files stored on internal memory. If an exploit is found in the security mechanism employed by the OS to grant access to files only by the owner application, such that other applications can access the files stored by our application, then any malicious application running on the device would be able to retrieve stored secrets at will. Figure 4.13 (p. 27) shows expected behavior and problematic behavior. The expected behavior is the one granted by the Android OS.

4.4.6 Secure Delete on the Storage Device

The Android OS uses, as its filesystem, the YAFFS2 \[\text{[12]}\] (Yet Another Flash File System version 2) filesystem. YAFFS2 is a filesystem specifically designed and written for devices which use flash storage. YAFFS2 is a journaled true log structured filesystem and therefore secure deletion is harder to achieve.

While considering secure deletion, the best solution would be to have it embedded directly within the filesystem, as we proposed earlier on in this work in Chapter 3. At the moment of writing most filesystems do not make such secure deletion operations available directly to developers although patches have been provided by researchers for both
non-journaled (ext2, vfat, ramfs, NFS, Base0fs) \[11\] and journaled filesystems (ext3, ext4) \[11\]. A possible implementation of secure deletion for YAFFS and YAFFS2 based on encryption of every file has been designed \[23\] but not implemented.

It is important to notice that, until Android OS or any other mobile OS provide the secure deletion calls to the developers, it is impossible to achieve secure deletion on the device from a high level developer perspective.

### 4.4.7 Secure Delete on the Communication Devices

Given the details over applicable secure deletion, we outline two different scenarios for S and R:

1. The underlying filesystem is not journaled (EXT2, \ldots)
2. The underlying filesystem is journaled (EXT3, EXT4, HFS+, \ldots)

In both cases the best solution to prevent software-based attacks is that of overwriting data on the file before removing it. This overwriting operation is achieved through the use of the *NIX utility `srm`. Such utility overwrites the file following the Gutmann 35-pass algorithm \[8\] and has been found available throughout the tested systems. In cases where `srm` is not installed on the system we provide a packaged binary to achieve secure deletion.
4.5 Analysis of Implementation

The implementation of the protocol has been benchmarked to have an idea of running times for the protocol.

4.5.1 Storing and Retrieving

Storing and retrieving secrets to and from the porter device $P$ requires both communication between the two devices and actual file writing and reading. Due to caching and Just in Time Compilation times tend to become lower (in the order of $3/4$ lower) with repetitive storage or retrieval operations. Figure 4.14 (p. 28) shows the average time of 100 storage and retrieval operations with a cold boot of both the $P$ service and the $R$ application.

![Protocol benchmark - client perspective](image)

**Figure 4.14:** Storage and Retrieve operation from a client ($R$) perspective. Showing average over 100 runs and standard deviation

As we can see the average time for storing a new key on the phone takes slightly more (approximately 20ms) and has a higher variance than a retrieve operation. This is due to the fact that the file needs to actually be created and written (storage operation) rather than just read (retrieve operation).

4.5.2 Deletion

Deletion of a secret is an operation that does not involve any communication from the porter device $P$ and the receiver device $R$. Figure 4.15 (p. 29) shows the average time of 100 deletion operations for a secret $K$. Given this results we can assume that data is timely deleted with a $\Delta_{\text{deletion}}$, on average, of 8 ms. To this $\Delta_{\text{deletion}}$ we should add the
0.581633 ms needed, on average, to call the deletion method after receiving an Alarm notice. Thus on average it takes approximately 8.6 ms after $t_e$ to mark the stored secret as deleted on the Android OS. We note, once again, that the data on the device is not accessible through the standard API once marked as deleted, but could potentially be retrieved if it has not been overwritten yet. At the moment of writing it is not possible to know at which point in time the actual erasure of the blocks containing the data will happen.

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{deletion_benchmark.png}
\caption{Deletion operation executed locally on the porter device P. Showing average over 100 runs and standard deviation.}
\end{figure}

4.5.3 Alarm Reset

Resetting the Alarms after a reboot of the device could potentially be a time-consuming operation. The main idea is that it should scale with the number of secrets currently stored on the device. Such is shown exactly on Figure 4.16 (p. 30). In particular, as shown in the graph, the time needed to reset 1 Alarm is 8.6 ms, for 10 it is 41 ms, for 100 it is 381, increasing in a linear way with the time needed to reset 1000 Alarms being 5592 ms.

4.6 Implementation Facts

The protocol implementation has been one of the major (and most time consuming) activities throughout the development of this thesis.

We now present a brief statistical analysis of the implementation in terms of lines of code and programming languages used. With “Lines of Code” we mean actual lines of code expunged from comments and blank lines.
Figure 4.16: Resetting Alarms after a boot of the device with a different number of secrets stored. Showing average over 10 runs for each number of stored keys and standard deviation.

<table>
<thead>
<tr>
<th>Programming/Markup language</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>4823</td>
</tr>
<tr>
<td>Javascript</td>
<td>470</td>
</tr>
<tr>
<td>XML</td>
<td>308</td>
</tr>
<tr>
<td>Objective C</td>
<td>223</td>
</tr>
<tr>
<td>Bourne Shell</td>
<td>221</td>
</tr>
</tbody>
</table>

The most challenging aspects of the implementation phases have been

- recompiling and making the bluecove bluetooth library work both on Linux and on MacOSX
- understanding the Thunderbird extension ecosystem
- integrating external tools within the Thunderbird extension, in particular Java integration both as an external tool and as an internal invocation through Live-Connect

4.7 Known Problems

The current SVN revision (3044) for the bluecove library has a bug that we have reported for a problem that would not allow acknowledgement of a successful write over the established RFCOMM channel. The fix devised to test the protocol is presented in the following patch.

1http://code.google.com/p/bluecove/issues/detail?id=104
CODE 4.1: Dirty patch to prevent crash

Index: OSXStackRFCOMM.m

--- OSXStackRFCOMM.m (revision 3040)
+++ OSXStackRFCOMM.m (working copy)
@@ -172,10 +172,12 @@

```c
void RFCOMMChannelController::rfcommChannelWriteComplete(void* refcon, IOReturn error) {
+    /*
+     if (refcon != NULL) {
+         ((RFCOMMConnectionWrite*)refcon)->rfcommChannelWriteComplete(error);
+     }
+     */
+}
```

```c
IOReturn RFCOMMChannelController::close() {
```
Chapter 5

Conclusion

Throughout this work we have analyzed the implementation of the protocol proposed in \cite{4} and in particular we focused on the two building blocks that ensure \textit{Forward secrecy under full compromise}.

Starting from a first implementation in which the sender $S$ was emulated by the receiver $R$ to focus on the \textit{righthand} part of the protocol we later decided to focus also on the \textit{lefthand} part of the protocol by implementing an extension for an email client. This implementation decisions find their roots in the DY channels described in the protocol. In fact both e-mail transmission and bluetooth communication are instances of such channels. Furthermore by using a mobile phone as a porter device and e-mail exchanges between the sender and the receiver we instantiated the protocol in a widespread used real life scenario.

We have found in the \textbf{secure deletion} and \textbf{timely deletion} the two major problems of the protocol to be able to ensure its intended goal. We analyzed the two problems both in a detailed, implementation specific, way and in a more generalized perspective.

As a conclusion we note that both problems have been discovered to be harder to be solved than we first imagined. In particular the \textbf{timely deletion} problem is solvable only if the underlying API permits it, as described. This would not be a major API change, and we are confident that in the near future a solution will be available. The \textbf{secure deletion} problem, instead, requires a deeper change in the filesystem architecture and is, at the moment of writing, not solvable.

In general, the best solution would be that to have a specialized device to act as a porter device, rather than a mobile phone. On the other hand our current solution can be widely adopted in a cheap and fast way.
5.1 Future Work

When filesystems are designed and implemented secure deletion is not a driving factor, especially when compared against performance, reliability and recoverability after a crash or a failure. We think that research for ways to embed secure deletion into specialized existing filesystems, such as the new emerging ones for flash hard drives, or design of new filesystems has to be taken into serious consideration. Once secure deletion is embedded into widely spread filesystems a new instantiation of the protocol should be provided and further tests and analysis carried out to evaluate the overall performance of the protocol and in particular the impact of true secure deletion.

With respect to the timely deletion problem and in general we think that a high level developer API with a stronger stress on security is required for modern smart phones.

5.2 Lessons Learned

Throughout this work I have learned many different aspects of communication security. In particular how a protocol can be put into practice through an implementation and all the challenges that such task implies. While implementing the protocol I have come across the Java security stack and actually implemented a Diffie-Hellman key exchange, after having only read about it several times. At last I have refined my skills in writing a scientific report longer than the usual two-paged ones, an aspect of the thesis that should not be minimized.
List of Figures

2.1 Implemented Protocol. The established symmetric key $K$ is used to encrypt the time-limited data $m$. All messages are authenticated, denoted by the authentication function $A_X(\cdot)$ (representing the digital signature of principal $X$ over the function input). The *squared brackets* notation represents symmetric encryption of data, $[K]_L$ means data $K$ encrypted with key $L$. The inverse function $[K]_L^{-1}$ represents the decryption function using key $L$ to decrypt $K$. Where the delete keyword is used the protocol expects a *secure deletion* operation to be carried out.

4.1 An overview of the protocol and communication between the 3 parties

4.2 Overview of the system, on the left the Android OS classes, on the right the PC client ones and in the middle the shared classes

4.3 The two classes in *ch.ethz.maclaudi.shared* that are shared between the Android code and the PC code

4.4 Class diagram for the classes running on the pc client

4.5 Class diagram for the classes running on the Android device

4.6 Channels used for the implementation of the protocol

4.7 SecureDelete Thunderbird extension overview, all elements are packed within the extension. In the dashed box is everything that works within Thunderbird, external tools are invoked but are actually other programs.

4.8 The prompt for a recipient and a $t_e$ date to start the protocol inside Thunderbird

4.9 Example of a message block generated and recognized by our Thunderbird extension

4.10 A general packet structure

4.11 Different structures of messages used in the protocol implementation

4.12 Overview of the *Alarms* system offered by the Android OS

4.13 *Expected behavior* (a) and *Problematic behavior* (b) concerning security mechanisms granted by the Android OS related to files ownership and permissions

4.14 Storage and Retrieve operation from a client ($R$) perspective. Showing average over 100 runs and standard deviation

4.15 Deletion operation executed locally on the porter device $P$. Showing average over 100 runs and standard deviation

4.16 Resetting *Alarms* after a boot of the device with a different number of secrets stored. Showing average over 10 runs for each number of stored keys and standard deviation

34
Bibliography


Appendix A

Code Appendix

Throughout the Appendix we display some parts of the code used for the implementation with a small description of what the code achieves. For the complete code you can download it, in a compressed format from: [http://www.cloudgoessocial.net/eth/thesis_code.zip](http://www.cloudgoessocial.net/eth/thesis_code.zip)

A.1 Protocol Code

In this Section we present the code that handles creating and sending messages throughout the whole protocol.

CODE A.1: checkBody Javascript function to check for SecureDelete blocks in a message and take appropriate action

```javascript
checkBody: function(msgBody, emailSender, auto) {
    var startSD = "+++++ SECUREDELETE +++++";
    var startSD = /\+\+\+\+\+ SECUREDELETE \+\+\+\+\+\+/g;
    var endSD = /===== SECUREDELETE =====/g;
    var startIndex = msgBody.search(startSD);
    var endIndex = msgBody.search(endSD);
    if ((startIndex == -1) || (endIndex == -1)) {
        if (!auto) alert("The selected message doesn't have any SECUREDELETE information");
    } else {
        var encodedString = msgBody.substring(startIndex + startSDString.length, endIndex);
        if (encodedString.length > 50) {
            var cryptoInstance = utilities.getCryptoInstance();
            var msgBytes = cryptoInstance.checkMsgSignature(encodedString);
            var mID = cryptoInstance.getMID(msgBytes);
            if (mID == "MI") {
                utilities.logToConsole("checking MI");
                this.summonJavaClient(mID, encodedString, emailSender, false);
            } else if (mID == "M5") {
                try {
                    var k = cryptoInstance.generateK(msgBytes, emailSender);
                    var te = cryptoInstance.getTime(msgBytes);
                    this.writeKToTmpFile(emailSender, k, te);
                    utilities.secureRemoveFile(emailSender, te, ".kp", false);
                    this.openMsgWindowToEncrypt(emailSender, te);
                } catch (e) {
                    alert("Could’t check M5, probably some Key is missing!");
                }
            } else if (mID == "M6") {
```
Appendix 38

```java
utilities.logToConsole("checking M6");
this.summonJavaClient(mID, encodedString, emailSender, true);
var te = cryptoInstance.getTime(msgBytes);
var encodedK = utilities.loadKey(emailSender, te, ".key");
utilities.secureRemoveFile(emailSender, te, ".key", true);
alert("Message decrypted:\n" + decrypted);
}
} else {
    if (!auto) alert("SECUREDELETE information missing");
}
}
}

decripted = cryptoInstance.decryptMessage(msgBytes, encodedK);
}
}

alert("Message decrypted:
" + decrypted);
39 } else {
    if (!auto) alert("SECUREDELETE information missing");
}
41 }

Code A.2: forwardBytesToProtocol Java method that is invoked on the R client when communicating to the P mobile device. In particular at the beginning of the method we check if the packet is a chained one and wait for the second bit before processing it.

```
byte[] iv = CryptoHelper.generateRandomIV();
IvParameterSpec ivSpec = new IvParameterSpec(iv);
SecretKeySpec keySpec = new SecretKeySpec(key, SharedConstants.M5ALGO);
cipher.init(Cipher.ENCRYPT_MODE, keySpec, ivSpec);
byte[] encrypted = cipher.doFinal(K);
// Create a long S from a String (email)
long S = CryptoHelper.hashToLong(email);
// Pack everything in a message
byte[] msgToSend = CryptoHelper.createMessageToSend(SharedConstants.M4.getBytes(), null, null, iv, encrypted);
// Using Base64 for encoding to send over data
thread.send(Base64.encodeBase64(msgToSend));
SharedConstants.logger.info("Sending M4...");
// Open up Thunderbird with a new message ready to be sent with the proper information
Desktop desktop;
if ((Desktop.isDesktopSupported()) && (desktop = Desktop.getDefault()).isSupported(Desktop.Action.MAIL)) {
try {
byte[] M5Contents = CryptoHelper.createMessageToSend(SharedConstants.M5.getBytes(), key, usedStoreKey.getPublic().getEncoded(), -1, null, null);
String tmpEncodedString = URLEncoder.encode("+++++ SECUREDELETE ++++", "UTF-8") + Base64.encodeBase64String(M5Contents) + "+++++ SECUREDELETE ++++", "UTF-8")
// javascript encodeURIComponent
String URLencodedMailto = "mailto:" + TBemail + "?subject=SDM5&body=" + tmpEncodedString;
URI mailto = new URI(URLencodedMailto);
desktop.mail(mailto);
} catch (URISyntaxException e) {
SharedConstants.logger.severe("URI Syntax error for mail");
e.printStackTrace();
} catch (IOException e) {
SharedConstants.logger.severe("IOException while creating mail");
e.printStackTrace();
}
} else {
// TODO fallback to some Runtime.exec(...) voodoo?
SharedConstants.logger.severe("No support for Mail in Desktop...");
}
// nullify L
L = null;
// Close application
//System.exit(0);
} catch (NoSuchAlgorithmException e) {
SharedConstants.logger.severe("DH algorithm not found");
e.printStackTrace();
} catch (InvalidKeyException e) {
SharedConstants.logger.severe("Invalid Key");
e.printStackTrace();
} catch (InvalidAlgorithmParameterException e) {
SharedConstants.logger.severe("Invalid Algorithm Parameters");
e.printStackTrace();
} catch (InvalidKeySpecException e) {
SharedConstants.logger.severe("Invalid Key Spec");
e.printStackTrace();
} catch (NoSuchPaddingException e) {
SharedConstants.logger.severe("Padding exception");
e.printStackTrace();
} catch (IllegalBlockSizeException e) {
SharedConstants.logger.severe("Illegal block size");
e.printStackTrace();
} catch (BadPaddingException e) {
SharedConstants.logger.severe("Bad Padding");
e.printStackTrace();
}
```java
else if (msgID.equals(SharedConstants.M8)) { 
    try {
        // Generate L
        KeyFactory keyFactory = KeyFactory.getInstance(SharedConstants.DH_ALGO);
        X509EncodedKeySpec receivedKeySpec = new X509EncodedKeySpec(CryptoHelper.getKey1(msg));
        DHKey keyFactory = (DHKey) keyFactory; 
        receivedKey = keyFactory.generatePublic(receivedKeySpec);
        KeyAgreement agreement = KeyAgreement.getInstance(SharedConstants.DH_ALGO);
        agreement.init((DHKey) keyFactory/privateKey, receivedKey.getParams());
        Lprime = agreement.generateSecret();
        // Use L to decrypt K
        Cipher cipher = Cipher.getInstance(SharedConstants.SYMMENTRIC_ENCRYPTION_ALGO);
        MessageDigest sha = MessageDigest.getInstance(SharedConstants.SHA_ALGO);
        byte[] key = sha.digest(Lprime);
        byte[] iv = CryptoHelper.getKey(msg);
        IvParameterSpec ivSpec = new IvParameterSpec(iv);
        SecretKeySpec keySpec = new SecretKeySpec(key, SharedConstants.AES_ALGO);
        cipher.init(Cipher.DECRYPT_MODE, keySpec, ivSpec);
        byte[] kReceived = cipher.doFinal(CryptoHelper.getEncryptedPayload(msg));
        if (kReceived != null) { 
            SharedConstants.logger.info("s aving Key to file ...");
            if (this.dumpKey(kReceived))
                SharedConstants.logger.fine("Key saved, close this program and return to Thunderbird for decryption!");
            else
                SharedConstants.logger.severe("Key couldn’t be saved to default file, check that ‘+ TB_path + ‘ is writeable");
        } else
            SharedConstants.logger.severe("Key couldn’t be retrieved, sorry.");
    } catch (NoSuchAlgorithmException e) {
        SharedConstants.logger.severe("No Such Algorithm");
        e.printStackTrace();
    } catch (InvalidKeySpecException e) {
        SharedConstants.logger.severe("Invalid Key Spec");
        e.printStackTrace();
    } catch (InvalidKeyException e) {
        SharedConstants.logger.severe("Invalid Key");
        e.printStackTrace();
    } catch (InvalidAlgorithmParameterException e) {
        SharedConstants.logger.severe("Invalid Algorithm Parameter");
        e.printStackTrace();
    } catch (IllegalBlockSizeException e) {
        SharedConstants.logger.severe("Illegal Block Size");
        e.printStackTrace();
    } catch (BadPaddingException e) {
        SharedConstants.logger.severe("Bad Padding");
        e.printStackTrace();
    } catch (NoSuchPaddingException e) {
        SharedConstants.logger.severe("No Such Padding");
        e.printStackTrace();
    } else
        SharedConstants.logger.severe("ERROR = message signature doesn’t match!");
}
```
String msgID = new String(CryptoHelper.getMID(msg));
if (msgID.equals(SharedConstants.M2)) {
    try {
        // Log.i(TAG, "RECEIVED M2");
        KeyPairGenerator keyGen = KeyPairGenerator.getInstance(SharedConstants.DH_ALGO);
        keyGen.initialize(new DHParameterSpec());
        KeyPair kp = keyGen.generateKeyPair();
        KeyFactory keyFactory = KeyFactory.getInstance(SharedConstants.DH_ALGO);
        X509EncodedKeySpec receivedKeySpec = new X509EncodedKeySpec(CryptoHelper.getKey1(msg));
        DHPublicKey receivedKey = (DHPublicKey) keyFactory.generatePublic(receivedKeySpec);
        KeyAgreement agreement = KeyAgreement.getInstance(SharedConstants.DH_ALGO);
        agreement.init(kp.getPrivate(), receivedKey.getKeyParams());
        agreement.doPhase(receivedKey, true);
        L = agreement.generateSecret();
        if (SharedConstants.B) sendMessageToUI("Diffie–Hellman exchange successful — L");
        byte[] msgToSend = CryptoHelper.createMessageToSend(SharedConstants.M3, getBytes(), kp.getPublic().getEncoded(), receivedKey.getEncoded(), -1, -1, null, null);
        // Log.i(TAG, "SENDING M3");
        write(Base64.encodeBase64(msgToSend));
    } catch (NoSuchAlgorithmException e) {
        Log.e(TAG, "DH Algorithm not present");
        e.printStackTrace();
    } catch (InvalidKeyException e) {
        Log.e(TAG, "Problem with the received public key of R");
        e.printStackTrace();
    } catch (InvalidAlgorithmParameterException e) {
        Log.e(TAG, "Invalid Algorithm Parameters");
        e.printStackTrace();
    }
    else if (msgID.equals(SharedConstants.M4)) {
        try {
            // Log.i(TAG, "RECEIVED M4");
            Cipher cipher = Cipher.getInstance(SharedConstants.SYMMETRIC_ENCRYPTION_ALGO);
            MessageDigest sha = MessageDigest.getInstance(SharedConstants.SHA_ALGO);
            byte[] key = sha.digest(L);
            byte[] iv = CryptoHelper.getIV(msg);
            IvParameterSpec ivSpec = new IvParameterSpec(iv);
            SecretKeySpec keySpec = new SecretKeySpec(key, SharedConstants.AES_ALGO);
            cipher.init(Cipher.DECRYPT_MODE, keySpec, ivSpec);
            // Log.e(TAG, "ENCPAYLOAD: " + new String(Base64.encodeBase64(CryptoHelper.getEncryptedPayload(msg))));
            byte[] kReceived = Base64.encodeBase64(cipher.doFinal(CryptoHelper.getEncryptedPayload(msg)));
            if (SharedConstants.D) Log.d(TAG, "kReceived == K ? " + Arrays.equals(kReceived, SharedConstants.K));
            long rTimestamp = CryptoHelper.getTime(msg);
            if (rTimestamp > System.currentTimeMillis()) {
                long rS = CryptoHelper getTime(msg);
                String fileName = CryptoHelper.generateFileName(rTimestamp, rS);
                try {
                    FileOutputStream fos = context.openFileOutput(fileName, Context.MODE_PRIVATE);
                    fos.write(CryptoHelper.long2byte(rTimestamp));
                    fos.write(kReceived);
                    fos.close();
                } catch (FileNotFoundException e) {
                    Log.e(TAG, e.getLocalizedMessage());
                } catch (IOException e) {
                    Log.e(TAG, e.getLocalizedMessage());
                }
            }
        }
    }
}
Log.e(TAG, e.getLocalizedMessage());
}

// nullify store L
L = null;
writeBenchmark(System.currentTimeMillis() - startTime);

} catch (NoSuchAlgorithmException e) {
Log.e(TAG, "DH Algorithm not present");
e.printStackTrace();
}

} catch (NoSuchPaddingException e) {
Log.e(TAG, "No Such Padding");
e.printStackTrace();
}

} catch (InvalidKeyException e) {
Log.e(TAG, "Invalid Key");
e.printStackTrace();
}

} catch (InvalidAlgorithmParameterException e) {
Log.e(TAG, "Invalid Algorithm Parameter");
e.printStackTrace();
}

} catch (IllegalBlockSizeException e) {
Log.e(TAG, "Illegal Block Size");
e.printStackTrace();
}

} catch (BadPaddingException e) {
Log.e(TAG, "Bad Padding");
e.printStackTrace();
}

}

else if (msgID.equals(SharedConstants.M7)) {
try {

// Generate L'
KeyPairGenerator keyGen = KeyPairGenerator.getInstance(SharedConstants.DH_ALGO);
keyGen.initialize(new DHParameterSpec());
KeyPair kp = keyGen.generateKeyPair();
KeyFactory keyFactory = KeyFactory.getInstance(SharedConstants.DH_ALGO);
X509EncodedKeySpec receivedKeySpec = new X509EncodedKeySpec(CryptoHelper.getKey1(msg));
DHPublicKey receivedKey = (DHPublicKey) keyFactory.generatePublic(receivedKeySpec);

KeyAgreement agreement = KeyAgreement.getInstance(SharedConstants.DH_ALGO);
agreement.init(kp.getPrivate(), receivedKey.getParams());
agreement.doPhase(receivedKey, true);
Lprime = agreement.generateSecret();

if (!(SharedConstants.B) sendMessageToUI("Diffie-Hellman exchange successful – L’")) {

// Retrieve secret from file
long rTimestamp = CryptoHelper.getTime(msg);
long rS = CryptoHelper.getS5(msg);
String fileName = CryptoHelper.generateFileName(rTimestamp, rS);
byte[] encodedK = new byte[SharedConstants.K.length]; // FIXME: how to get right size?
FileInputStream fis = ctxt.openFileInput(fileName);
fis.skip(8); // long timestamp
fis.read(encodedK);
fis.close();

byte[] decodedK = Base64.decodeBase64(encodedK);

// Encrypt decoded K
Cipher cipher = Cipher.getInstance(SharedConstants.SYMMETRIC_ENCRYPTION_ALGO);
MessageDigest sha = MessageDigest.getInstance(SharedConstants.SHA_ALGO);
byte[] key = sha.digest(Lprime);
byte[] iv = CryptoHelper.generateRandomIV();
IvParameterSpec ivSpec = new IvParameterSpec(iv);
SecretKeySpec keySpec = new SecretKeySpec(key, SharedConstants.AES_ALGO);
cipher.init(Cipher.ENCRYPTMODE, keySpec, ivSpec);
byte[] encrypted = cipher.doFinal(decodedK);

// Send back message to client
if (!(SharedConstants.B) sendMessageToUI("Sending secret to the client");
byte[] msgToSend = CryptoHelper.createMessageToServer(SharedConstants.M8, kp.publicKey().getEncoded(), receivedKey.getEncoded(), -1, -1, iv, encrypted);
write(Base64.encodeBase64(msgToSend));

// nullify Lprime
Lprime = null;
}

} catch (NoSuchAlgorithmException e) {
Log.e(TAG, "DH Algorithm not present");
}

} catch (InvalidKeyException e) {
Log.e(TAG, e.getLocalizedMessage());
e.printStackTrace();
}
A.2 Javascript Helper Functions

In this Section we present some helper functions used throughout the Javascript code, in particular two interesting helper functions used in the Thunderbird extension to create a Java instance and to summon an external program (srm) through bash.

CODE A.4: `getCryptoInstance` Javascript function to create a Java instance of the Crypto class used for cryptographic methods. The methods of the Crypto class can then be called directly from within Javascript thanks to the LiveConnect scripting utility.

```javascript
getCryptoInstance : function() {
    Components.utils['import']('resource://securedelete/LiveConnectUtils.js', this.LiveConnect);
    var jars = ['URLSetPolicy.jar', 'SecureDelete.jar'];
    var [loader, urls] = this.LiveConnect.initWithPrivs(java, 'securedelete@syssec.ethz.ch', jars);
    var cryptoClass = java.lang.Class.forName('Crypto', true, loader);
    var cryptoInstance = cryptoClass.newInstance();
    cryptoInstance.initCrypto(this.getInstallDirPath() + '/java/');
    return cryptoInstance;
}
```

CODE A.5: `secureRemoveFile` Javascript function that invokes a bash script to securely remove stored data.

```javascript
secureRemoveFile : function(sender, te, suffix, tmp) {
    if (tmp === true) {
        var file = Components.classes['@mozilla.org/file/directory-service;1'].getService(Components.interfaces.nsIProperties).get('TmpD', Components.interfaces.nsIFile);
    } else {
        var file = Components.classes['@mozilla.org/file/local;1']
            .createInstance(Components.interfaces.nsIFile);  
    
    ```
A.3 Java Helper Functions Found in the CryptoHelper Class

In this Section we present some helper functions used both by the PC client and the Android service, as shown in Section 4.3.1.

```java
createMessageToSend

Java method to generate all the packets used throughout the protocol. This is a general method that is invoked with different parameters depending on the kind of packet to be sent.

```
Appendix 45

// M3: | SIGNATURE | ID | KEY | KEY
finalSize = SharedConstants.M3_OFFSETS[SharedConstants.M1_OFFSETS.length - 1];
tmpSize = finalSize - SharedConstants.SIG_SIZE;
finalMag = new byte[finalSize];
tmpMsg = new byte[tmpSize];
System.arraycopy(mID, 0, tmpMsg, 0, SharedConstants.PACKET_ID_SIZE);
System.arraycopy(key1, 0, tmpMsg, SharedConstants.M3_OFFSETS[1] - SharedConstants.SIG_SIZE, key1.length);  
System.arraycopy(key2, 0, tmpMsg, SharedConstants.M4_OFFSETS[2] - SharedConstants.SIG_SIZE, key2.length);
}
} else if (stringMsgID.equals(SharedConstants.M4)) {
    // M4: | SIGNATURE | ID | S | TIME | IV | ENCRYPTED
finalSize = SharedConstants.M4_OFFSETS[SharedConstants.M4_OFFSETS.length - 1] + encryptedPayload.length;
tmpSize = finalSize - SharedConstants.SIG_SIZE;
finalMag = new byte[finalSize];
tmpMsg = new byte[tmpSize];
System.arraycopy(mID, 0, tmpMsg, 0, SharedConstants.PACKET_ID_SIZE);
System.arraycopy(long2byte(time), 0, tmpMsg, SharedConstants.M4_OFFSETS[1] - SharedConstants.SIG_SIZE, SharedConstants.S_TIME_SIZE);
System.arraycopy(V, 0, tmpMsg, SharedConstants.M4_OFFSETS[3] - SharedConstants.SIG_SIZE, key1.length);
}
} else if (stringMsgID.equals(SharedConstants.M5)) {
    // M5: | SIGNATURE | ID | TIME | KEY | KEY
finalSize = SharedConstants.M5_OFFSETS[SharedConstants.M5_OFFSETS.length - 1];
tmpSize = finalSize - SharedConstants.SIG_SIZE;
finalMag = new byte[finalSize];
tmpMsg = new byte[tmpSize];
System.arraycopy(mID, 0, tmpMsg, 0, SharedConstants.PACKET_ID_SIZE);
System.arraycopy(long2byte(time), 0, tmpMsg, SharedConstants.M5_OFFSETS[1] - SharedConstants.SIG_SIZE, SharedConstants.T_TIME_SIZE);
System.arraycopy(V, 0, tmpMsg, SharedConstants.M5_OFFSETS[3] - SharedConstants.SIG_SIZE, key1.length);
}
} else if (stringMsgID.equals(SharedConstants.M6)) {
    // M6: | SIGNATURE | ID | TIME | IV | ENCRYPTED
finalSize = SharedConstants.M6_OFFSETS[SharedConstants.M6_OFFSETS.length - 1] + encryptedPayload.length;
tmpSize = finalSize - SharedConstants.SIG_SIZE;
finalMag = new byte[finalSize];
tmpMsg = new byte[tmpSize];
System.arraycopy(mID, 0, tmpMsg, 0, SharedConstants.PACKET_ID_SIZE);
System.arraycopy(long2byte(time), 0, tmpMsg, SharedConstants.M6_OFFSETS[1] - SharedConstants.SIG_SIZE, SharedConstants.T_TIME_SIZE);
System.arraycopy(V, 0, tmpMsg, SharedConstants.M6_OFFSETS[3] - SharedConstants.SIG_SIZE, key1.length);
}
} else if (stringMsgID.equals(SharedConstants.M7)) {
    // M7: | SIGNATURE | ID | TIME | S | KEY
finalSize = SharedConstants.M7_OFFSETS[SharedConstants.M7_OFFSETS.length - 1];
tmpSize = finalSize - SharedConstants.SIG_SIZE;
finalMag = new byte[finalSize];
tmpMsg = new byte[tmpSize];
System.arraycopy(mID, 0, tmpMsg, 0, SharedConstants.PACKET_ID_SIZE);
System.arraycopy(long2byte(time), 0, tmpMsg, SharedConstants.M7_OFFSETS[1] - SharedConstants.SIG_SIZE, SharedConstants.S_TIME_SIZE);
System.arraycopy(V, 0, tmpMsg, SharedConstants.M7_OFFSETS[3] - SharedConstants.SIG_SIZE, key1.length);
}
} else if (stringMsgID.equals(SharedConstants.M8)) {
    // M8: | SIGNATURE | ID | KEY | KEY | IV | ENCRYPTED
finalSize = SharedConstants.M8_OFFSETS[SharedConstants.M8_OFFSETS.length - 1] + encryptedPayload.length;
tmpSize = finalSize - SharedConstants.SIG_SIZE;
finalMag = new byte[finalSize];
tmpMsg = new byte[tmpSize];
System.arraycopy(mID, 0, tmpMsg, 0, SharedConstants.PACKET_ID_SIZE);
}
System.arraycopy(key1, 0, tmpMsg, SharedConstants.MSG_OFFSETS[1] - SharedConstants.SIG_SIZE, key1.length);
System.arraycopy(key2, 0, tmpMsg, SharedConstants.MSG_OFFSETS[2] - SharedConstants.SIG_SIZE, key2.length);
System.arraycopy(IV, 0, tmpMsg, SharedConstants.MSG_OFFSETS[3] - SharedConstants.SIG_SIZE, SharedConstants.IV_SIZE);
System.arraycopy(encryptedPayload, 0, tmpMsg, SharedConstants.MSG_OFFSETS[4] - SharedConstants.SIG_SIZE, encryptedPayload.length);
}
byte[] signature = signMessage(tmpMsg);
System.arraycopy(signature, 0, finalMsg, 0, SharedConstants.SIG_SIZE);
System.arraycopy(tmpMsg, 0, finalMsg, SharedConstants.SIG_SIZE, tmpMsg.length);
return finalMsg;
}

Code A.7: signMessage Java method to sign a message before sending it.

private static byte[] signMessage(byte[] msg) {
    byte[] DECODED_MAC_KEY = Base64.decodeBase64(SharedConstants.MAC_KEY);
    try {
        Mac hmac = Mac.getInstance(SharedConstants.MAC_TYPE);
        SecretKeySpec sks = new SecretKeySpec(DECODED_MAC_KEY, SharedConstants.MAC_KEY_TYPE);
        hmac.init(sks);
        return hmac.doFinal(msg);
    } catch (InvalidKeyException e) {
        System.err.println("Invalid Key Exception");
        e.printStackTrace();
    } catch (NoSuchAlgorithmException e) {
        System.err.println("HmacSHA256 not found");
        e.printStackTrace();
    }
    return null;
}

Code A.8: checkMsgSignature Java method to check that a message has been received correctly and hasn’t changed during transmission.

public static byte[] checkMsgSignature(byte[] msg) {
    byte[] msgSignature = new byte[SharedConstants.SIG_SIZE];
    byte[] msgToCheck = new byte[msgSizeWithoutSig];
    System.arraycopy(msg, SharedConstants.SIG_SIZE, msgToCheck, 0, msgSizeWithoutSig);
    System.arraycopy(msg, 0, msgSignature, 0, SharedConstants.SIG_SIZE);
    long msgToReturn = (Arrays.equals(msgSignature, signMessage(msgToCheck))) ? msgToCheck : null;
    return msgToReturn;
}

Code A.9: long2byte and byte2long Java methods to encapsulate and decapsulate a long to a byte for sending and receiving within a packet.

public static byte[] long2byte(long l) {
    byte[] b = new byte[8];
    ByteBuffer bBuffer = ByteBuffer.wrap(b);
    LongBuffer lBuffer = bBuffer.asLongBuffer();
    lBuffer.put(l, 0, 1);
    return b;
}

public static long byte2long(byte[] b) {
    ByteBuffer bBuffer = ByteBuffer.wrap(b);
    return bBuffer.getLong();
}

Code A.10: generateRandomIV Java method to generate a fresh random IV used when encrypting some data.
public static byte[] generateRandomIV() {
    SecureRandom sr = new SecureRandom();
    byte[] IV = new byte[SharedConstants.IV_SIZE];
    sr.nextBytes(IV);
    return IV;
}