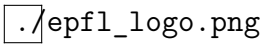

Securing Biometric Templates On Smart Cards

Master semester project report - Spring 2013

Jonathan CHESEAUX (jonathan.cheseaux@epfl.ch)

Supervisors : Andrzej Drygajlo - Leila Mirmohamadsadeghi

Contents

1 Introduction

With the rise of identity thefts, one's main concern, when storing personal biometric data on a portable medium, is privacy protection. The use of such devices leads to important security issues since the impact of an attacker stealing this sensitive data can be really devastating for the victim.

The main purpose of this project is to investigate the feasibility of fingerprint matching on a smart card. To be more secured, the matching has to be on-card exclusively, i.e. no data should leave the card for obvious security reasons, and the sensitive information stored should not lead to the original fingerprint features.

In this report, the approach taken to implement a secure and reliable system using the Java Card[1] programming platform is detailed, as well as the results of performance and security analysis of the final implementation.

[1]Java Card 2.2 is a programming development kit well suited for smart card programming

2 Background

2.1 Recognition chain

2.1.1 Minutiae Cylinder-Code (MCC)

The main concern while dealing with fingerprint recognition is the fact that it is impossible to get two fingerprints impressions from the same user that are completely identical. Indeed, different factors can alter the fingerprint quality, such as finger pressure, sweat, orientation, etc. To overcome this issue, [R. Cappelli, M. Ferrara and D. Maltoni] developed a powerful algorithm which can describe fingerprints' minutiae in a fashion that tolerates feature extraction imprecisions or errors [**?**].

This algorithm, so called Minutia Cylinder-Code (`MCC`), produces fixed-size descriptors from a fingerprint impression by analysing local structures around each minutia. It describes neighbouring minutia angular direction and distances from one another and represents this information as a cylinder (orientation defines the height of this cylinder, and distance from neighbours defines its base). The main advantages of such an algorithm are that it is computationally efficient (due to the fixed-size nature of minutiae descriptors), robust against feature extraction errors (due to noisy impressions) and does not depend on finger orientation or distortion. The produced template is a set of minutiae descriptors, consisting of floating-point values, and its length is 2048 values per descriptor.

2.1.2 Template transformation

The descriptors produced by the original `MCC` algorithm are not irreversible [**?**] nor revocable [**?**]. For complete privacy protection the descriptors need to be transformed. The solution chosen in the implementation of this project for protecting a biometric template is given in `Algorithm ??` [**?**]. It uses a key of length 2048 consisting of a random permutation of all integers from 0 to 2047. A non-invertible function (`line 6 - Algorithm ??`) is applied on the cylinder values two-by-two and the result is then binarized (`lines 7-10`) by using a specific cut-off value. The resulting template contains descriptors of length 1024 bits[2]. This transformation prevents an attacker from extracting the original fingerprint impression and permits to implement revocable templates since each protected template is unique given a certain key.

[2]Since the non-invertible function used in the Algorithm **??** combines two randomly chosen values in the descriptor, its length will be divided by two. A protected template only contains 0's or 1's (due to the binarization) and it is easily implemented on a computer by storing 128 bytes of data per descriptor

Algorithm 1: Biometric template privacy protection

Input: D (a set of minutia descriptors $\langle T_1, T_2, \ldots, T_n \rangle$)
Input: H (the encryption key)
Output: P (the transformed template)

$P \leftarrow (0, 0, \ldots, 0)$;
$index \leftarrow 0$; $A \leftarrow 5000$; $threshold \leftarrow 10^5$; \\ *Tuning parameters*
for *each descriptor* T *in* D **do**
  for $i \leftarrow 0$ ***to*** *length(*T*)* **do**
    if i ***even*** **then**
      $p \leftarrow (A * (T[H[i]] + T[H[i+1]]))^2$ **mod** n
      if $p > threshold$ **then**
        $P[index] \leftarrow 1$
      else
        $P[index] \leftarrow 0$
      $index \leftarrow index + 1$;

Figure 1: Biometric template privacy protection

Algorithm 2: Local similarity sort

input: $P1$, enrolment protected template $\langle P_1^{(1)}, P_1^{(2)}, \ldots, P_1^{(n)} \rangle$
$P2$, verification template $\langle P_2^{(1)}, P_2^{(2)}, \ldots, P_2^{(n)} \rangle$
$M1$, enrolment minutiae directions $\langle M_1^{(1)}, M_1^{(2)}, \ldots, M_1^{(n)} \rangle$
$M2$, verification minutiae directions $\langle M_2^{(1)}, M_2^{(2)}, \ldots, M_2^{(n)} \rangle$
delta, threshold
Output: *score*, the matching score between 0.0 and 1.0

$index \leftarrow 0$
$gamma \leftarrow (0, 0, \ldots, 0)$
$minNP \leftarrow 3$; $maxNP \leftarrow 10$; $muP \leftarrow 30$; $tauP \leftarrow 0.4$; \\ *Tuning parameters*
for $i \leftarrow 1$ ***to*** n **do**
  for $j \leftarrow 1$ ***to*** n **do**
    $norm \leftarrow \|P_1^{(i)}\| + \|P_2^{(j)}\|$
    if $angularDiff(M_1^{(i)}, M_2^{(j)}) \leq delta$ **then**
      $gamma[index] \leftarrow \frac{1.0 - \|\text{hammingDistance}(P_1^{(i)}, P_2^{(j)})\|}{norm}$
      $index = index + 1$
sort$(gamma)$
$z \leftarrow (1 + \exp(-tauP * (n - muP)))$
$nP \leftarrow minNP + \lfloor (z * (maxNP - minNP)) \rfloor$
$sum \leftarrow 0$
for $i \leftarrow 0$ ***to*** nP **do**
  $sum = sum + gamma[i]$
return sum/nP

Figure 2: Local similarity sort

2.1.3 Template matching

The LSS matching algorithm, described in Figure ??, compares the distance[3] of all two by two cylinders and produces a similarity score[4] based on angular distances of minutiae pairs. These distances are invariant, even after the templates are transformed using a key, but of course it differs if the fingerprints do not come from the same user or the transformation key used to protect the enrolment template and the one used to protect the verification template are not identical.

2.2 Using smart cards for recognition

Smart cards are pocket-sized cards provided with an embedded CPU and a small data storage unit. They are currently used in many different fields such as health care (Carte Vitale[5]), finance (credit/debit cards), facilities entrance control, mobile phone (SIM[6]), IT security and many more.

The key advantages of smart cards are security, portability and ease of use and since their invention in the late 1960s, their market share has exploded. Currently, there are over 7 billions smart cards produced per year[7] and this number is growing more and more each year.

2.2.1 Smart card specifications

The limited size of the embedded chip implies low hardware performance. Not every Java card share the same characteristics and the actual tendency is to use next generation cards[8] which are more powerful and can contain a larger amount of data, thanks to computer technology miniaturization. Table ?? shows a comparison between Java Card 2.2 and Java Card 3.0 specification.

[3] In this project, the distance computed is the Hamming distance (since binary data are used)

[4] The score is a real number between 0.0 (no match) and 1.0 (perfect match)

[5] "Carte Vitale" is a french health insurance card

[6] "Subscriber Identity modules" (SIMs) are used in mobile telephony devices for authentication and ciphering

[7] Eurosmart annual market study on global smart card shipments - http://www.smartcardalliance.org/pages/smart-cards-intro-market-information

[8] Java Card 3 - http://www.oracle.com/technetwork/articles/javase/javacard3-142122.html

	Java Card 2.2	Java Card 3.0
byte, short	✓	✓
int, long, char, String	✗	✓
Threads	✗	✓
Garbage collection	✗	✓
Networking	✗	✓
2D Arrays	✗	✓
CPU	8-bit CPU	32-bit CPU
RAM	8kB of RAM	24kB of RAM

Table 1: Comparison of Java Card 2.2 and Java Card 3.0

2.2.2 Communication protocol

The smart cards and the card reader (so called Card Acceptance Device or `CAD`) exchange messages using Application Protocol Data Units (`APDU`). The structure of these messages is defined by [**?**]. `Table ??` and `Table ??` highlight the different fields of such messages.

There is two types of `APDU` messages : the `APDU` requests, which are sent from the `CAD` to the smart card and the `APDU` responses which are sent from the card to the `CAD`. In `APDU` requests, the mandatory fields are `CLA`[9], `INS`[10], `P1` and `P2`[11]. The optional fields are `Lc`[12], `Data field`[13] and finally `Le`[14].
As for the response `APDU`, there are only 2 mandatory fields, `SW1, SW2`[15] and one optional field for sending data to the `CAD`.

In this project, the command instructions (`CLA`) are arbitrarily defined as shown in `Table ??`. The two last instructions (matching initialization and card resetting) are sent in `APDU` requests by the card acceptance device using an empty data field.

Mandatory header				**Optional body**		
CLA	INS	P1	P2	Lc	Data field	Le

Table 2: Command `APDU` structure

[9]Instruction class - indicates the type of command
[10]Instruction code - indicates a specific command, e.g. "enrol minutiae", "match", ...
[11]Instruction parameters for the command
[12]Data field length (in bytes)
[13]Data to be sent to the card
[14]Maximum number of response bytes expected
[15]Response code, for example 90 00 indicates success of the transaction

Optional body	Mandatory trailer	
Data field	SW1	SW2

Table 3: Response APDU structure

2.2.3 Smart card simulator

A smart card simulator is embedded in the Java Card Workstation Development Environment (JCWDE [?]), it allows the developer to test the applet throughout its development process. This tool basically emulates a smart card by listening to a communication socket and replies to APDU requests with APDU responses, as does a real smart card when linked to a card acceptance device.

Unfortunately, this tool does not provide debugging commands, is not verbose at all and the only way of troubleshooting problems is by sending APDU requests and analysing APDU responses, if any are received. These constraints were really time-consuming and made it difficult to develop the fingerprint matching process.

Nevertheless, the first implementations were made with the original Java Runtime in order to offer more debugging functions. This early version of the project respected the Java Card API limitations (no floating-points, no 32-bits integers) but permitted to print values in a console and use the Java Debugger to inspect the code. Despite this advantage, it was really challenging to switch to the Java Card environment.

CLA code	Corresponding instruction
0x01	Load verification template
0x02	Load verification minutiae
0x03	Unroll user's template
0x04	Unroll user's minutiae
0x05	Initiate fingerprint matching
0x06	Reset the card

Table 4: CLA codes for the project

2.2.4 Java Card programming environment

The first challenge encountered in this project was to deal with a restrictive Java development kit, namely Java Card, in order to implement the matching on the smart card. Indeed, in order to run a software on a smart card that is not provided with a powerful CPU nor large data storage units, the Java Virtual Machine (JVM) implemented on the chip must be limited to the most necessary features. The main difficulty was to overcome the limited API provided by this library.

2.2.4.1 Library limitations

Only a small portion of the original JVM is installed on a smart card chip. The provided Java Card JVM forbids the use of advanced types such as 32-bits integers, floating-point

values or characters. On the other hand, it provides 16-bits integers (Short), boolean, bytes and utility methods to handle byte to short conversion (and vice versa), deep copy arrays and others.

The major consequence of not being able to use 32-bits integers and using 16-bits integers instead, is that the maximum integer value that can be represented is 32767 and no array of more than 32767 elements can be built. This is not really a problem for the project since only 128 bytes per minutia descriptor are needed and the fingerprint feature extraction results in no more than 60 minutia [16].

[16] For 60 minutiae, $128 * 60 = 7680$ bytes of data need to be stored

3 Materials and methods

To have a completely secure system, no sensitive data should flow out of the smart card. An on-card matching technique ensures that data only enters the smart card and no sensitive data is shared with the card reader. When a user wants to unlock a secure feature of the smart card, he needs to scan his fingerprint and its impression is first transformed, then sent to the smart card along with its minutiae angular directions. The smart card then applies the `LSS` algorithm by comparing these data to the enrolled fingerprint. If the matching score is high enough, the smart card can, for example, deliver a decryption key or a secured asset stored on the card to the user.

3.1 Hardware

3.1.1 Set-up connection to the card (simulator)

In order to test the application, a smart card acceptance device needs to be emulated by using the `JCWDE` utility tool. A Java Card applet is selected and identified by its application identifier (`AID`). It is also necessary to provide the package name in the project hierarchy.

For example, in this project, the simulator can be launched by opening a Windows command console and typing `jcwde <config_file_location>` where `<config_file_location>` is the path to a file containing the package name and its associated `AID`[17]. The simulator is now listening for connections on a defined port in the local-host network.

Now that the simulator is working, the connection with the `CAD` is initiated by running the file `JavaCardReader.java` (`Figure ??`) in the project `CardAcceptanceDevice`. The connection to the simulator is done by a `TCP` socket connecting to the local server socket of the simulator.

Once the connection is made, it is still necessary for the card reader to select and run the applet on the smart card. It is done by setting the `INS` field of the `APDU` request to `0xA4` and the `Data` field to the application `AID` : `0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x00, 0x00` and sending it to the card.

CLA	INS	P1	P2	Lc	Data field
0x00	0xA4	0x04	0x00	0xA0	0x01 0x02 0x03 0x04 0x05 0x06 0x07 0x08 0x09 0x00 0x00

Table 5: Command `APDU` content for the applet selection

[17] In this project, the config file contains the following instruction :
`monpackage.MonApplet 0x01:0x02:0x03:0x04:0x05:0x06:0x07:0x08:0x09:0x00:0x00`
This file should be located in the same directory as the package

3.1.2 Data transfer to the card

Once the connection is established, the user's template and minutiae are enrolled. Since a transformed template can contain a lot of data[18], it has to be broken into several chunks to respect the `APDU` requests size limit (255 bytes). In order to keep track of the packet numbers, the `P1` and `P2` fields of the request are used to indicate the current packet number sent. On the Java Card side, it simply fills a byte-array, chunk by chunk, with the received bytes (see `Figure ??`).

The enrolment should be done only once. In a real life scenario, this can be done at a bank's premises for example, where fingerprints of the customer are acquired, transformed with a unique key and loaded on the smart card. When the customer wants to authenticate with his smart card, he simply inserts it in a card acceptance device and scans his fingerprint. The card reader then transforms the currently scanned finger impression and loads it in the card along with the minutiae angular directions. This transfer is done in the exact same way as the enrolment process.

CLA	**INS**	**P1**	**P2**	**Lc**	**Data field**
0x00	0x03	Packet number		Packet size	Packet content

Table 6: Command `APDU` content for enrolling a user's template

3.2 Software

3.2.1 `LSS` algorithm for Java Card environment

One of the great challenge and certainly the main goal of this project was to translate the fingerprint matching algorithm to the Java Card specification. As said before, the absence of floating-points numbers was a real inconvenience since there are numerous complex computations involved in `Algorithm ??`. For example, lines 11-12 of this algorithm can simply not be computed with integer numbers, since precision losses are too much important. The chosen workaround is to pre-compute these formulas for every possible number of minutiae and round the results at the end of the computations, so that the precision loss is smaller. These results are stored in a lookup table in the Java Card part of the application.

This solution is not sufficient since there are still integers divisions in the algorithm that can lead to floating-points values, but this problem is overcome by multiplying floating-point values in a given formula by a constant. For example, to compute $\frac{\pi * 0.035}{1.45} \approx 0.0758315$, the denominator and the numerator are multiplied by a constant, say 10'000 : $\frac{31415 * 350}{14500} = 758$. Once out of the smart card, the result is simply divided by 10'000, giving an approximation of the result, here : 0.0758. Of course this approximation results in a loss of precision, but as it is seen later in the Section **??**, this precision loss is not a sensible problem for the matching algorithm. To find the best value for this threshold, several

[18]Recall that a protected template is a collection of minutia descriptors, each descriptor contains 128 bytes of information and there can be up to 60 minutiae extracted from a fingerprint.

tests using different values are conducted. By printing the false match rates (FMR)[19] and the false non-match rates (FNMR)[20] for different threshold values, the value giving the best results appears to be 1550 (see Figure ??). The matching score is thus be between 0 and 1550.

CLA	INS	P1	P2
0x00	0x05	0x00	0x00

Table 7: Command APDU content for initiating matching process

Concerning the minutia angular directions, radians are not convenient because of the same precision issue explained before, so degrees are used instead. This does not affect the result but the feature extraction part of the project is modified in order to produce minutiae angular directions in degrees.

3.2.2 How to use the developed program

1. Start the smart card simulator (JCWDE)
   (a) Open a Windows command console in the SmartCardDevice/bin directory
   (b) Type : `jcwde -p <portNumber> monapplet.app`
   The option `- p <portNumber>` is not required. It defines which port the simulator will be listening to.
   (c) Press RETURN and the simulator will start
2. Launch the CardAcceptanceDevice Java project
   (a) Open a new Windows command console in the JavaCardReader directory.
   (b) Type : `java -jar javacardreader.jar <enrollTemplate> <Verification-Template> <saveFolder> <portNumber> <differentKey>`
   - `<enrollTemplate>` is the file location of the enrolment fingerprint picture
   - `<verificationTemplate>` is the file location of the verification fingerprint picture
   - `<saveFolder>` is the folder in which the matching score will be stored
   - `<portNumber>` is the port number listened by the simulator.
   - `<differentkey>` 0 = same key, 1 = different key for verification and enrolment template transformation

   (c) Press RETURN and the program starts
3. The score is printed in the console along with other information and is also saved in the chosen directory in a .csv file

[19]The FMR also called False Acceptance Rate (FAR) denotes the likelihood that the access is granted for an imposter

[20]The FNMR also called False Rejection Rate (FRR) denotes the likelihood that the access is denied for a genuine user

Figure 3: JCWDE Simulator listening for a connection

4 Results

By using workarounds to respect the Java Card limitations, poor results were expected. However, the accuracy of the on-card matching implementation is surprisingly encouraging, as explained in the subsections below.

4.1 Test database

Three different databases [?][?][?] are used for parameter tuning and testing the accuracy of the on-card matching. These databases are provided by the Fingerprint Verification Competition (FVC)[21], an international competition where industrial, universities and independent developers test the accuracy of their recognition algorithms. These databases contain impressions of several users' fingers (8 fingerprints per user) and even synthetic fingerprints [?]. The quality of the impressions can vary a lot between users due to rotation, displacement, wet/dry impressions, age of the users, etc.

The databases are preprocessed with FingerJetFX[22], an open-source program that performs minutiae extraction from a fingerprint scan.

Every results presented in this report are computed from the entire FVC 2004 - DB1 database. To compute the genuine scores, the 8 fingers' impression of a user are matched together. For the imposter scores, several users are randomly chosen from the database and matched against all the other users.

4.2 Imposters/Genuine scores

A graphic way to test the performance of the developed program is by computing the genuine and imposter scores histogram. For genuine scores, one user's fingerprint is enrolled on the card and is then matched against the seven remaining finger impressions of this user. The key used for enrolment template protection is the same for each transformation of the seven other impressions.

To compute imposter scores, one user's fingerprint is chosen and matched against other users'. Two scenario are also distinguished : in the first scenario ("unknown key"), the imposter does not know which key was used for the enrolment template protection, so he generates one at random. In the second scenario ("stolen key"), it is assumed that the imposter can extract the transformation key from the smart card.

[21] https://biolab.csr.unibo.it/fvcongoing/UI/Form/Home.aspx
[22] http://www.digitalpersona.com/fingerjetfx/

4.2.1 Original recognition algorithm

`Figure ??` shows that the separation between the genuine scores and the imposter scores becomes slightly smaller if the attacker can extract the template transformation key but class separation still remains noticeable.

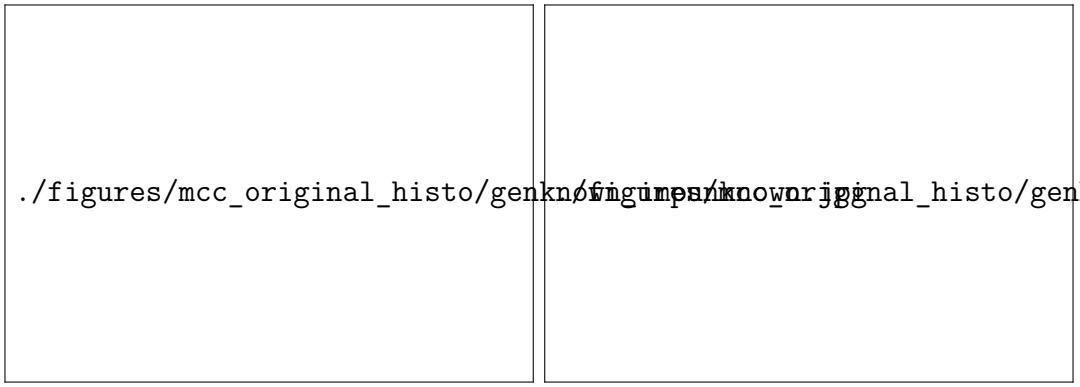

Figure 4: Original matching - Histograms of imposters (**red**) /genuine (**blue**) score repartition. On the left : unknown key scenario. On the right : stolen key scenario.

4.2.2 On-card matching

There is a similar behaviour with the on-card matching, but it is more visible than in the original algorithm. This is certainly due to the precision loss occurring during the computation of the matching scores. It is also visible that the genuine scores are more spread than in the previous algorithm, leading to a larger standard deviation (see `Section ??` for detailed statistics).

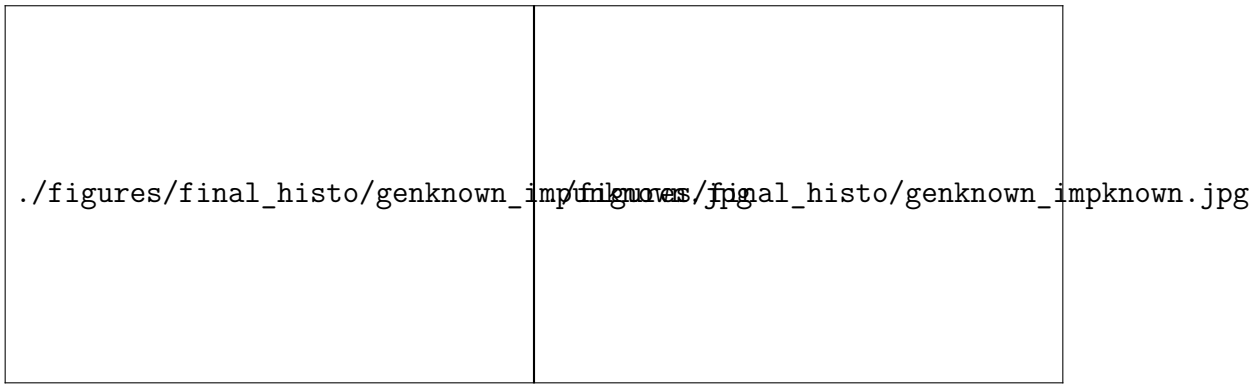

Figure 5: On-card matching - Histograms of imposters (**red**) /genuine (**blue**) score repartition. On the left : unknown key scenario. On the right : stolen key scenario.

4.3 False match rate (FMR) and False non-match rate (FNMR)

An alternative way to see the separation between the distribution of genuine and imposter scores is the DET[23] curve which describes the rate of FNMR for a certain FMR (and vice versa).

[23]Detection error tradeoff

The DET curves below are built on the same samples used in the previous section. The red line represents the Equal Error Rate (EER), i.e. the rate where the FNMR and the FMR are equal. The FMR1000, FMR100 and FMR10 lines represent the corresponding FNMR for 1‰, 1% and 10% FMR.

4.3.1 Original matching algorithm

On the left part of Figure ?? the DET curve is constantly decreasing. This implies that as the FNMR grows, the FMR decreases and vice versa. For example, at the crossing of the FMR1000 line and the DET curve, the corresponding FNMR for a FMR of 0.1% is about 10%, meaning that 1 out of 1000 imposters are falsely accepted but a genuine user is falsely rejected 1 out of 10 times. If a system requires fast authentication (i.e. FNMR of 1%), the FMR ratio is 1 (every imposter is accepted) and this scenario is not secure at all. A good compromise is to fix the matching score threshold such that the FMR and FNMR are equal (EER). On the right part, where imposters steal the transformation key from the smart card, when the FMR grows, the FNMR diminishes slower than on the left part. This shows the importance of protecting the transformation key on the smart card[24].

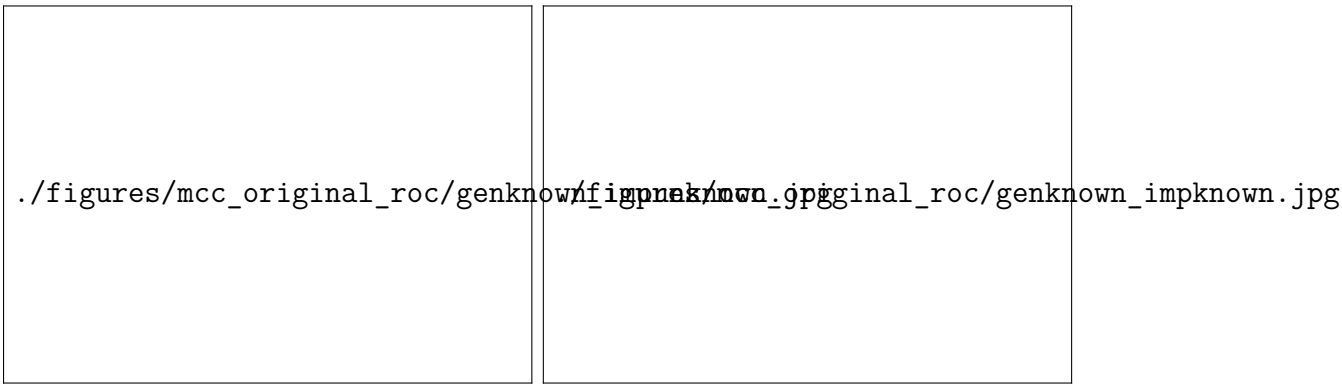

Figure 6: Original matching - DET (blue) and EER (red). On the left : unknown key scenario. On the right : stolen key scenario.

4.3.2 On-card matching

The left sub-figure of Figure ?? shows surprisingly good results. Indeed, as the FNMR grows, the FMR drops to zero very quickly. For example, when the FNMR reaches 9%, the FMR is at about 0.4%, and when it reaches 15%, the FMR is exactly 0%. The consequence is that a highly secured authentication can be built over this model by setting the threshold value so that it produces a FNMR of 15%. Genuine users will be more likely to be falsely rejected but no imposter will be falsely accepted, thus offering the best security protection. On the other hand, the scenario of stolen key (right part of this table) gives somehow really bad results. The EER is greater than on the left and for a FNRM between 0.1% and 1%, the FMR lays between 50% and 100%. No safe system can be built in this scenario, and this enforces again the idea that the key needs to be encrypted on the card.

[24]Key encryption is discussed in Section ??.

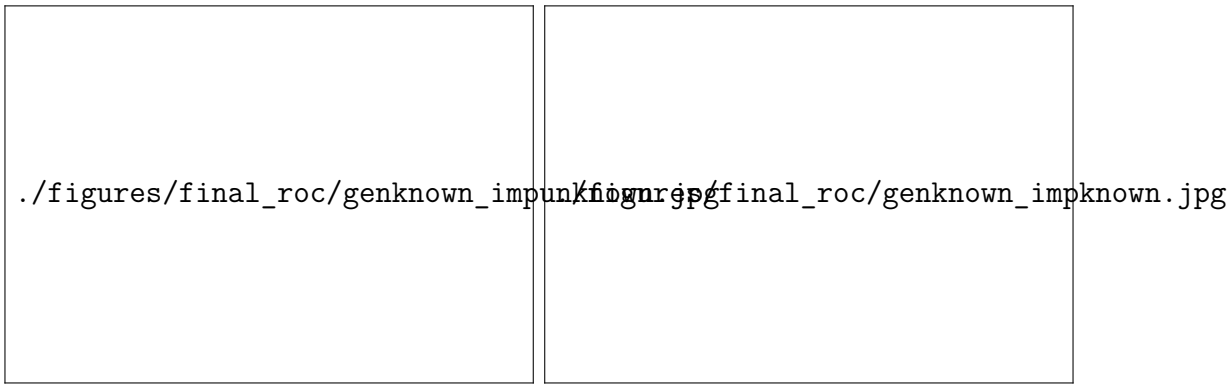

Figure 7: On-card matching - DET **(blue)** and EER **(red)**. On the left : unknown key scenario. On the right : stolen key scenario.

The whole process (applet selection, templates upload, matching) takes in average less than 4 seconds (see **Table ??**). The template verification takes up to 2 seconds, which is really fast. To this value we must add the fingerprint scanning time and the feature extraction, which will be highly dependant on the type of devices used.

Card connection	49	49	49	49	49	49	49	49
Enrolment template uploading	2008	2049	2049	1900	2058	2049	1949	2051
Verification template uploading	1902	1500	1900	1951	1751	1552	1999	1999
Matching	93	94	95	90	94	92	94	86
Total	**4052**	**3692**	**4093**	**3990**	**3952**	**3742**	**4091**	**4185**

Table 8: Speed measures (in milliseconds) for 8 random genuine fingerprint matching

4.4 Performance comparison

The results presented in `Table ??` show some detailed statistics about the score repartition for both algorithms and both scenarios (unknown/stolen key). The statistics are quite identical for both original and on-card algorithms but it is noticeable that for the on-card algorithm, the means of both genuine and imposter scores (unkown key) are more spaced than in the original algorithm. This leads to a better class separation between imposter and genuine scores and thus gives a better accuracy.

	Genuine		Imposters			
			Unknown key		*stolen key*	
	mean	**deviation**	**mean**	**deviation**	**mean**	**deviation**
Original	0.53	0.11	0.37	0.05	0.41	0.05
On-card	0.56	0.09	0.34	0.04	0.44	0.04

Table 9: Mean and standard deviation of the score distributions for on-card matching and original algorithm

`Table 10` maps fixed `FMR` values to their corresponding `FNMR`. For example, with the original algorithm, a 10% `FMR` leads to 5.34% `FNMR`. This means that a genuine user's access is unlikely denied but on the other hand, 1 out of 10 imposters' fingerprint are accepted by this algorithm. If the imposter can steal the key, the `FNMR` raises to 11.52%.

Better results for the on-card matching technique can be read on the right part of this table, especially for a `FMR` of 0.1% that gives 7.55% `FNMR` for the original algorithm and 6.71% `FNMR` for the Java Card version. These values are a great indicator for the security analysis of the implementation. If a system does not tolerate any unauthorized access, the threshold used to determine if the score is genuine or not is bigger. This reduces the `FMR` at a very small rate but then genuine users are more often rejected by the system. On the other hand, if a system needs fast authentication, this threshold is fixed to a smaller value, thus reducing the `FNMR` and raising the `FMR`.

Original algorithm			
Unknown key		Stolen key	
FMR	**FNMR**	**FMR**	**FNMR**
10%	5.34%	10%	11.52%
1%	7.07%	1%	19.34%
0.10%	7.55%	0.10%	21.55%
EER : 6.41%		EER : 11.97%	

Java Card implementation			
Unknown key		Stolen key	
FMR	**FNMR**	**FMR**	**FNMR**
10%	4.00%	10%	23.29%
1%	5.86%	1%	32.71%
0.10%	6.71%	0.10%	37.00%
EER : 4.63%		EER : 19.34%	

Table 10: `FNMR` values for fixed `FMR` rates for two scenarios (unkown and stolen key)

5 Conclusion

5.1 State of the project

As it is explained in the introduction of this report, the main purpose of this project was to investigate the feasibility of an accurate on-card matching technique built in a smart card's chip. In the view of the previous results and performances, it is noticeable that this goal is successfully reached. However, enhancements are still needed to build a professional solution and a non-exhaustive list of future improvements is suggested in the `Section ??`.

5.2 Future directions

5.2.1 Testing on real hardware

During this project, the opportunity to test the fingerprint matching on a real smart card did not present itself since the laboratory[25] did not own such devices[26]. The hardware needed is a fingerprint reader, a card acceptance device linked to a computer and some smart cards.

Despite the fact that the project is compatible with Java Card 2.2, it does not imply that it will be compatible with a real smart card as it is. Indeed, the deployment will be highly dependent of the smart card's firmware specifications.

5.2.2 Precision loss

As it is explained in `Section ??`, precision losses occur due to the absence of floating-point in a smart card device. There is another solution to overcome this limitation by manually implementing floating-point values following the standard way of expressing real values in a computer [**?**]. With this implementation, it is possible to perfectly match the performance of the original algorithm. However, this will lead to more memory consumption and further analyses are needed to see if this solution is suitable for smart card chips.

5.2.3 Encryption of the biometric template and minutiae

Transformed templates are assumed to be irreversible, so even if an attacker can steal information from the card, he will not be able to reconstruct the original fingerprint from it. Another sensitive data stored on the card is the minutiae angular directions. If a fully secured system is required, a layer of encryption needs to be implemented. An asymmetric encryption method (such as `RSA` encryption) seems specifically well suited for

[25] `LIDIAP`, `IDIAP` laboratory at `EPFL` - http://idiap.epfl.ch/

[26] There is several website on which it is possible to acquire such devices.
The company `Gemalto`, for example, sells Java cards (http://smartware2u.com/category/1-contact-smart-cards.aspx) and Java card readers with integrated fingerprint scanner (http://smartware2u.com/category/2-contact-smart-card-readers.aspx?pageindex=2)

such an application. Furthermore, the Java Card environment provides utility functions for RSA encoding and its deployment should not be a difficult task.

5.2.4 Encryption of the transformation key

As illustrated in Section ??, the results are not satisfying if an attacker manages to extract the key from the smart card. In order to build a secure system we absolutely need to encrypt this key and RSA seems to be a good fit for this purpose.

5.2.5 Fine tuning parameters

Throughout the whole project, assumptions were made and empirical values were chosen, such as the precision parameter introduced in Section ?? or the template protection algorithm parameters. By introducing machine learning concepts, parameters can be chosen in an optimal way to produce better results. Since the matching process opposes two fingerprints, it should be possible to use a Multi-layered Perceptron for binary classification [?].

5.3 Personal conclusion

Since my first year at EPFL, I was really concerned and interested in biometrics recognition. From a personal point of view, I find this project really interesting but also really challenging. I had no experience in smart card programming before and thus had to go through a lot of literature to get a small draft project working on the simulator. I learned that programming for chip cards is a meticulous process and does not tolerate any mistakes.

Finally, the results of the tests on the FVC databases are very encouraging and better than imagined. In fact, it was not even certain that an implementation on a smart card was possible at all in the beginning of the project, due to the numerous limitations of the Java Card environment.

A Appendices

A.1 Precision parameter

Figure A.1.1: Plot of FNMR/FMR of imposters/genuine test for different values of precision to find the best parameter (here : 1550)

A.2 Classes UML[27]

Figure A.2.1: UML graph of the smart card application. The main class is `MonApplet.java`, it handles the APDU exchanges with the card reader and store the data received (minutiae and protected templates). `Util.java` offers convenient function for handling short and byte values. `LSSMatcher.java` performs the template matching.

Figure A.2.2: Java classes of the biometrics package implemented on the card reader. The `Minutia` class holds information on minutia direction and spatial location. `ReadMinutiaFromISOFile` is the feature extractor and `MCCBase` handles the template privacy protection

Figure A.2.3: `JavaCardReader` represents the card acceptance device. It basically acquires a template (from a fingerprint scanner, for example) and send it to the card for fingerprint matching. It can also enrol a user's biometrics data on the smart card.

[27]Unified Modelling Language