An Asymmetric Fingerprint Matching Algorithm for Java Card™

Stefano Bistarelli[1,2], Francesco Santini[2], and Anna Vaccarelli[2]

[1] Dipartimento di Scienze, Università "G. D'Annunzio" di Chieti-Pescara, Italy
bista@sci.unich.it
[2] Istituto di Informatica e Telematica, CNR, Pisa, Italy
{stefano.bistarelli,francesco.santini,anna.vaccarelli}@iit.cnr.it

Abstract. A novel fingerprint matching algorithm is proposed in this paper. The algorithm is based on the minutiae local structures, that are invariant with respect to global transformations like translation and rotation. The match algorithm has been implemented inside a smartcard over the Java Card™ platform, meeting the individual's need for information privacy and the overall authentication procedure security. The main characteristic of the algorithm is to have an asymmetric behaviour, in respect to the execution time, between correct positive and negative matches. The performances in terms of authentication reliability and speed have been tested on some databases from the Fingerprint Verification Competition 2002 (FVC2002). Moreover, our procedure has shown better reliability results when compared with related Java Card™ algorithms.

1 Introduction

The term "biometrics" is commonly used today to refer to the authentication of a person by analyzing the physical characteristics (like fingerprints) or behaviour characteristics (like voice or gait). Fingerprint matching is one of the most diffused biometric techniques used in automatic personal identification or verification, because of its strong reliability and its low implementation cost.

Performing a biometric verification inside a smart card is notoriously difficult, since the templates tend to eat-up a large part of the card's memory, while the biometric verification algorithms are almost beyond the processing capabilities of standard processors. With *Match On Card* (MOC) technology the fingerprint template is stored within the card, unavailable to the external applications and the outside world. In addition the matching decision is securely authenticated internally by the smartcard itself: in this way, the card only trusts itself for eventually unblocking stored sensitive information, such as digital certificates or private keys for digital signature. Our verification MOC algorithm has been developed to work in this very strictly bounded environment.

The algorithm is based on some minutiae characteristics (ridge pattern micro characteristics) and more precisely on their local structure information, so there is no need to pre-align the processing fingerprint templates, that would be a difficult task to implement inside a smartcard. Moreover it shows an asymmetric execution time between correct positive matches (same fingerprint) and correct negative matches (two different fingers), and this because the match procedure stops immediately when few minutiae

pairs result in a positive match. If this check doesn't succeed, for example if the two fingers are different or if the two acquisitions of the same finger are very disturbed, the procedure is fully executed lasting longer.

2 Background

The most evident structural characteristic of a fingerprint is the pattern of interleaved ridges and valleys that often run in parallel; at local level, other important features called *minutiae* refer to ridge discontinuities. Most frequently the minutiae types can be individuated by terminations, where a ridge line ends, and bifurcations, where a ridge bifurcates forming a "Y". The minutiae can be used in fingerprint matching since they represent unique details of the ridge flow and are considered as a proof of identity.

The template, in its generic definition, is a mathematical representation of the fingerprint "uniqueness" to be used later during the matching phase: the template acquired during enrollment is defined as the *reference template* and it is in some way associated with the system user identity, while the template acquired during the verification phase is defined as the *candidate template*.

Matching the templates represents an extremely difficult problem because of the variability in different impressions of the same fingers; most important affecting factors introduced during image acquisition are the *displacement* and the *rotation* depending on the different positioning of the finger on the acquisition sensor, *non-linear distortions* due to the skin plasticity and *partial overlap*, since a part of the fingerprint can fall outside of the acquisition area and therefore different samples of the same finger could correspond only on a smaller area.

The algorithms used to resolve fingerprint matching can be classified [1] in three main branches: *correlation-based* [6], where the match is performed by superimposing two fingerprint images and computing the correlation between corresponding pixels (in this case the template is directly the finger image); *minutiae-based*, whose theory is fundamentally the same as for manual fingerprint examiners, and *ridge feature-based* [5], where the fingerprints are compared in terms of ridge pattern features other than the minutiae or pixels intensity, like texture information or sweat pores.

Focusing on the minutiae based algorithms, the match procedure essentially consists in finding the maximum number of corresponding minutiae between two templates; this problem can be addressed also as a more general *point pattern matching* problem [13]. We can subdivide minutiae matching class into two more branches: *global minutiae matching* [3] requires a first fingerprint alignment phase that subsequently permits to match the aligned templates. In *local minutiae matching* [4] two fingerprints are instead compared according to their local minutiae structures, which are characterized by attributes invariant with respect to global transformations such as translations or rotations. Local matching supplies simplicity, low computational complexity and higher distortion tolerance, while a global matching grants a high distinctiveness.

Regarding smartcard related work, in [10] is described a very simple $O(n^2)$ matching algorithm, where *n* is the minutiae number in one template. For a given minutia in the reference template, the algorithm finds all the minutiae in the candidate template for which the distance between position coordinates and the difference in orientation angles are below the predefined thresholds. If more than one can be matched with the same reference minutia, this conflict is resolved by choosing the geometrically nearest.

One specific algorithm for fingerprint matching on the Java Card™ platform, using a feature extraction environment similar to ours, is described in [11]; it uses two distinct algorithms on different feature types (hybrid matcher) and at the end the overall score is calculated as a linear combination of the two independent sub-scores. The first algorithm is based on the minutiae features and a graph structure is built starting from the core point position in the fingerprint, then visiting the neighbor minutiae; the matching procedure has been inspired from the point-pattern matching algorithm in [12] and its purpose is to find a *spanning ordered tree* touching as many nodes as possible in the two graphs. The second algorithm is a ridge feature-based and has been implemented as described in [5]; this algorithm is very fast and can be easily implemented on a smartcard, since the match consists only in finding the euclidean distance between two feature vectors.

3 Our Matching Algorithm

3.1 Features

In our algorithm implementation we have adopted the NIST Fingerprint Image Software (NFIS) [2], an open source toolkit which includes the MINDTCT minutiae data extractor used to extract the minutiae from a given fingerprint image. We have used this information to derive additional features directly used in our matching algorithm; these features are computed for each minutia in respect to its neighbors, and so each neighbor is described by the following four features (see also Fig. 1):

- the euclidean distance between the central minutia and its neighbor minutia (segment *D* in Fig. 1); referred to as Ed in the rest of the paper.
- the angle between segment *D* and the central minutia ridge direction (angle α in Fig. 1); latterly referred to as Dra.
- the difference angle between central minutia and neighbor ridge orientation angle ($\theta_1 - \theta_2$ in Fig. 1); latterly referred to as Oda.
- the ridge count between central and neighbor minutiae: given two points *a* and *b*, the ridge count between them is the number of ridges intersected by the segment $\overline{ab}$ (in Fig. 1 ridge count value is 1); latterly referred to as Rc.

Choosing the maximum number of the neighbors is very important for the system reliability performances (but in contrast with the matching speed), and so we have decided to increase this number from the default MINDTCT value (5) to the new value of 8. We have also modified the MINDTCT C source code to consider only the neighbors with a minimum reliability threshold: the modified MINDTCT finds for every minutia its eight nearest neighbors in respect to the euclidean distance, with a good reliability estimation given by a predefined threshold value. If the number of neighbors for a minutia found in this way is low (i.e. less than 5), then the neighbors are searched again with a lower reliability threshold (the reliability evaluation is found by MINDTCT); we have introduced all these changes to build a "good" neighborhood with more information, enough to face the possible lack of some minutiae in the template.

3.2 Algorithm Description

Our proposed matching algorithm computes how much the neighborhood of a minutia in the candidate template is similar to the neighborhood of each minutia in the reference

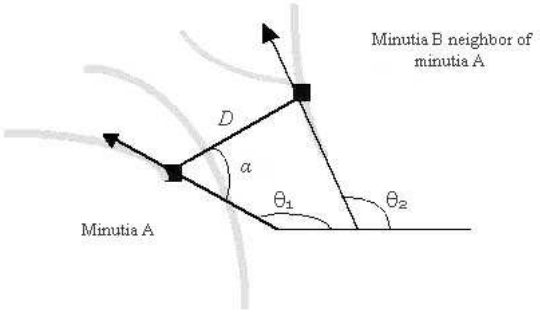

Fig. 1: Features graphical description

template; after this scan, the two most similar minutiae are matched and then discarded from subsequent scan phases concerning other different minutiae of the candidate template. All these similarity measures are summed together during the process, and at the end the algorithm can decide if the two templates match by applying a threshold on this global score. Our procedure is based on the minutiae local structures (Section 2).

But as said before, matching on smartcard environment is bounded by the low computational complexity due to the hardware simplicity (CPU limitations first of all), and thus waiting for a complete minutiae match could lead to a waiting time too long for the user. In our algorithm we solve this problem by stopping the computation as soon as it is possible to assert, with satisfactory confidence, that the considered templates belong to the same fingerprint. To realize this improvement, our algorithm stops as soon as it finds some minutiae pairs (i.e. a number between 2 and 5) matching with a very good similarity measure, or even promptly when only the last examined minutiae pair has a matching value less than a very rigourous threshold; otherwise, if these two conditions are not fulfilled, the algorithm explores all the minutiae pairings space. This relaxation showed a very good security performance in our tests and provided an evident speed improvement in the matching decisions regarding positive matches (Section 4). The delay for unsuccessful matches scanning all the minutiae list is not of much interest, because it is clearly more important to gain a high execution speed while verifying the true card-owner identity than quickly rejecting an impostor!

As input, the matching procedure receives both the neighbor features information for the one by one candidate minutia to be matched, and the entire reference template. The algorithm scans sequentially the minutiae of the reference template until a good match for the input minutia is found (reference 1 in Fig 2). Both candidate and reference minutiae lists are stored according to the increasing minutia reliability value: in this way we try to stop the procedure more quickly by scanning a reduced portion of the template minutiae lists, since a minutia with a high reliability in a given template, if not cut away by partial overlapping (Section 2), will have probably a high reliability also in other templates obtained from the same finger. So the stopping conditions can be met earlier than in a casual disposition of the minutiae in the list. Moreover, it is obviously better to prematurely stop the procedure with few but “good” minutiae than with low quality ones. The chosen matching minutia in the reference template is then marked as “already matched” and it is not considered in the successive iterations.

To compute the dissimilarity between two minutiae in different templates, the algorithm uses the information about the neighbor features and executes the following four steps in sequence (4 in Fig 2):

1. To find the difference in absolute value between corresponding features: $EdDiff =\mid Ed_1 - Ed_2 \mid$, $rcDiff =\mid Rc_1 - Rc_2 \mid$, $draDiff =\mid Dra_1 - Dra_2 \mid$ and $odDiff =\mid Oda_1 - Oda_2 \mid$.
2. To check that every feature difference value is below the corresponding acceptance threshold; if only one difference value exceeds the relative threshold, the two neighbors cannot correspond in the two neighborhoods ($edDiff$ must not be greater than $edDiffThr$, $rcDiff$ than $rcThr$, $edDiff$ than $draThr$ and $odDiff$ than $odThr$). The set of the four feature difference thresholds can be globally defined as the features *bounding box*, which makes the algorithm tolerant to small non-linear distortions.
3. To multiply each feature difference for the relative weight value: $edWghtDiff = edDiff{*}edWght$, $rcWghDiff = rcDiff{*}rcWght$, $odWghtDiff = odDiff {*}odWght$ and $draWghtDiff = draDiff * draWght$. The different weight values are necessary to attribute more importance to the features that match better, for example the euclidean distance. To obtain each weight value, we have also divided by the respective feature difference bounding box threshold, since we want these differences to be normalized and homogenous.
4. To sum together all the four weighted differences to represent the global dissimilarity between the two neighbors: $NeighDissimilarity = edWghtDiff + rcWghtDiff + draWghtDiff + odWghtDiff$.

Following these steps, the algorithm finds for the first neighbor (in the casual neighborhood order) of the reference minutia, the most similar neighbor in the input minutia among those satisfying the bounding box checks; the most similar is the one for which the algorithm finds the lowest *NeighDissimilarity* value. The chosen most similar neighbor in the reference minutia is then marked and not considered while matching other neighbors. The obtained *NeighDissimilarity* value is then added to the global similarity score between the minutiae, *MinDissimilarity*. The procedure is repeated exactly for all the other neighbors (excluding the already marked ones, 3 in Fig 2) or until the required minimum number *N* (i.e. 4) of neighbors is matched. At the end of the two neighborhoods scanning (at the end of the *for*, 2 in Fig 2), if the procedure has found less than *N* matching neighbor pairs between the two minutiae (6 in Fig 2), then these two minutiae are not considered as matching because their neighborhoods agree on too few evidences to be a reliable matching minutiae pair, even if the *NeighDissimilarity* value is very low. At the same time, this procedure stops immediately as we match the previous *N* threshold value of neighbors (5 in Fig 2), because we have seen that stopping before the whole neighborhood scan is sufficient to grant a good reliability and, meanwhile, the match time is considerably speeded up.

The *MinDissimilarity* score between the minutiae is finally divided by the number of matched neighbor pairs and then added to the global dissimilarity value between the candidate and reference templates (7 in Fig 2): the *TemplDissimilarity*; the same algorithm is then executed for the next candidate template minutia in reliability order. When

all of the input minutiae have been processed, this global *TemplDissimilarity* value on templates is divided by the number of matched minutiae *MinutiaeNMatched*, finding in this way the mean. A comparison between a match threshold and this mean value can consequently be used to decide if the two templates belong to the same fingerprint (if the mean is below the threshold): lower *TemplDissimilarity* expresses more affinity.

```
{MINUTIAE MATCHING PROCEDURE}

   - Input:* one candidate template minutia m1;
            * minutiae list of the reference template;

1  For each minutia m2 in reference template not yet matched{
2    For each neighbor n2 of minutia m2 {
       - MinDiff = upperLimit;
       - ChosenNbr= null;
3      For each not already matched neighbor n1 of m1 {
4        - Executes the four steps between the n1-n2
           corresponding features (directly processes next n1
           if the bounding box rejects the controls);
         If (NeighDissimilarity < MinDiff) {
           - MinDiff = NeighDissimilarity;
           - ChosenNbr = n1; }
       }
       If (ChosenNbr != null) {
         - ChosenNbr is marked as "matched";
         - MinDissimilarity += MinDiff;
         - number of matched neighbors NM= NM + 1; }
5      If (NM > N)
         - m1 and m2 are "matched": break from this For;
     }
6    If (NM < N)
       - Continue with the next minutia m2
     else {
7      - m1 and m2 are "matched": TemplDissimilarity+=
         (MinDissimilarity \ NM);
       - break from this For;
     }
   }
   - m1_m2_MatchCost = MinDissimilarity \ NM;
   If (m1 and m2 are "matched") {
     - MinutiaeNMatched++;
     - Mark reference minutia m2 as "matched";
8    If (m1_m2_MatchCost < VeryOptValue)
       - STOP: the match is accepted;
     If (m1_m2_MatchCost < OptValue)
       - OptMinNumber++;
9    If (OptMinNumber == OptNumberThreshold)
       - STOP: the match is accepted;
   }
   - Process another minutia m1 if no stopping condition
     has occurred or if m1 and m2 are not "matched";
```

Fig. 2: Matching core function; text reference is in the first column

That, therefore, is the full algorithm description, but as said before, the matching procedure will probably end before the complete minutiae list of the candidate template has been processed: if at the end of the minutiae matching routine the dissimilarity value between two matched minutiae is “very good”, that is below a tightening threshold *Opt-Value*, the counter *OptMinNumber* is incremented and as soon as it reaches a predefined

constant value corresponding to the threshold *OptNumberThreshold*, the whole matching procedure can be stopped with a positive result (8 in Fig 2). The algorithm can be positively stopped also as soon as it finds only one minutiae pair with an "exceptionally good" *MinDissimilarity* value below the *VeryOptValue* threshold (9 in Fig 2), which is intended to be much stricter than the previous *OptValue*.

The described algorithm complexity is $O(n^2)$, where *n* is the number of the minutiae in a single template, even if in practice, the approach of stopping the computation with few minutiae shows a significant speed improvement.

3.3 Algorithm Implementation

The fingerprint matching algorithm described in Sec. 3.2 has been fully developed in Java Card™ using the Java Card™ 2.1.2 API and finally deployed on Cyberflex Access 32Kb Java Card™ with the Cyberflex Access SDK (version 4.3). The chosen smartcard has 32Kbyte of EEPROM, about 1Kbyte of RAM memory distributed between the transaction mirror, stack and transient space, 8 bit CPU at up to 7.5Mhz external clock frequency and the transmission protocol used is the *T=0* at 9600 bit/sec.

The algorithm has been developed by implementing the Java Card™ Biometric API [9] realized by Java Card Forum™ (JCF): this application programming interface ensures the interoperability of many biometric technologies with Java Card™ and allows multiple independent applications on a card to access the biometric functionalities (like verification); this is ideal to secure the digital signature, storing and updating account information, personal data (i.e. health information) and even monetary value. Clearly, our application manages even the enrollment and match requests coming from the external PC applications through several *Card Acceptance Device* (CAD) sessions.

Java Card™ technology [7] adapts the Java™ platform for the use on smartcards, smart buttons or other simple devices, like USB tokens. This adaptation produces a global reduction of the platform functionalities and its result is a substantial decreasing of the expressive capacity. Benefits and drawbacks are identical to those of its "mother technology": high portability and programming/developing quickness, but also a reduced execution speed due to the additional bytecode interpretation layer.

Due to the environment constraints like the EEPROM space, we have limited the maximum number of minutiae forming the template to the 20 most reliable, and the neighbor feature values have been sampled to be then stored in the low capacity Java Card™ data types as *byte* type (the maximum minutia occupation is 40 *byte*).

4 Performance Results

To measure the performance, we used the *Finger Verification Competition 2002* [8] edition (FVC2002) fingerprint databases, which, as we know, is the only public benchmark (together with the other editions of the same contest, FVC2000 and FVC2004[3]) allowing the developers to unambiguously compare their algorithms. In particular, we have adopted the two databases respectively collected using the optical sensor "FX2000" by Biometrika and the optical sensor "TouchView II" by Identix; each of the databases is

[3] We have used the databases from FVC2002 since the FCV2004 collections were granted to us only a few weeks ago. For the permission to use these collections, we acknowledge Raffaele Cappelli, Dario Maio and Davide Maltoni from the Biometric Systems Lab (University of Bologna).

100 fingers wide and 8 impressions per finger deep. Moreover, we have analyzed our algorithm in respect to the one described in [11], using the proprietary image database provided to us by the authors[4] and made up of about 550 samples collected using the FX2000 optical scanner model; henceforth referred to as the "Hybrid Database".

The most important biometric systems evaluation parameters are the *False Acceptance Rate* (FAR), *False Rejection Rate* (FRR), the *Equal-Error Rate* (EER), which denotes the error rate for which FAR and FRR are identical, and the *Receiver Operating Characteristic* (ROC) curve. Other interesting performance indicators can be derived to show the algorithm's behaviour for applications that need high security: for example, the FAR100 (lowest achievable FRR for a FAR $\leq$ 1%), FAR1000 (lowest FRR for FAR $\leq$ 0.1%) and ZeroFAR (the lowest FRR for FAR = 0%). Another important factor to be considered, especially for MOC algorithms, is clearly the average matching time.

The test distribution between positive and negative matches can greatly influence the declared performances, so we decided to run the same tests as the FVC2002 competition [8] between the same fingerprint images: 2,800 iterations to find FRR and 4,950 to find FAR. The same test distribution criteria of FVC2002 were adopted also for the Hybrid Database (1,485 FAR tests and 2,449 for FRR). We also kept the same algorithm parameters configuration during the tests of all these three image collections. However, better results could be obtained by suitably adapting the parameters to each database. In Table 1 we present the obtained reliability results, where "-" means that the score is not achievable with the particular parameter configuration used.

Table 1: Overall performance results of our algorithm

Fingerprint database	EER	FAR100	FAR1000	ZeroFAR
FVC2002 FX2000	8.5%	10.6%	12.5%	-
FVC2002 TouchView II	8.5%	10.6%	12.3%	-
Hybrid Database	0.48%	0.44%	0.53%	0.57%

Fig. 3 shows the FAR-FRR and the ROC curves for the tests on the FVC2002 FX2000 database. The strange shape of the graph lines (in respect to classic ones) comes from the decision to stop the algorithm even with few but "good" minutiae pairs. This decision is independent from the final matching score and so setting the match threshold to a low or a high value does not correspondingly results in a FAR or a FRR of 0% and 100%. For example, the FRR curve in Fig. 3 starts with a match threshold equal to 0 from about 14% and not 100%, since most of the matches has been however accepted using one of the stopping conditions described in Sec. 3.2. Moreover, these conditions introduce only less than 5% errors in the total of the false matches accepted.

For our main parameters configuration and our purposes we were essentially interested in giving the best FAR1000 performance, but we have tested other configurations that can improve EER to about 7% or take ZeroFAR to 15.6%.

[4] For the permission to use their fingerprint database, we thank Tommaso Cucinotta and Riccardo Brigo from ReTiS Lab of Sant'Anna School of Advanced Studies (Pisa).

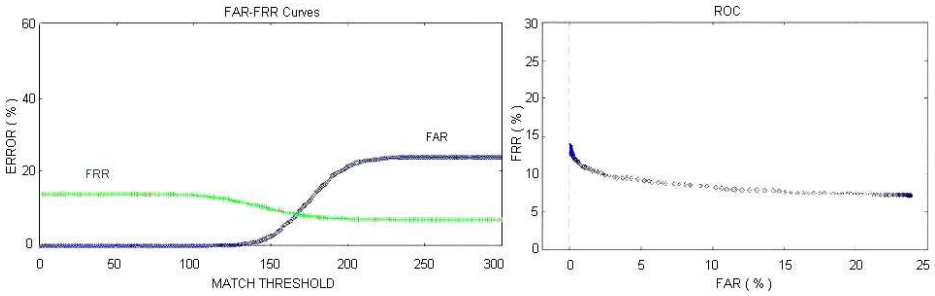

Fig. 3: FAR-FRR curves and ROC for FVC2002 FX2000 database

In Fig. 4 we report, for two databases, the algorithm match time distribution with respect to the correct FRR tests. We can observe that an on card matching time of about 1-8 seconds is obtained for nearly all of the matches (about 90% for the Hybrid Db.). We have noticed that the minimum time of 1 second has been achieved frequently in these tests and can be obtained even more often using a good enrollment fingerprint image, since in this case the two stopping conditions of Sec. 3.2 can be met very often. The maximum time is instead about 45 seconds, but this result is obtained only when the two acquisitions belong to different fingerprints (not interesting for our purposes) or they are very disturbed: this prevents the algorithm for quickly stopping without exploring all the minutiae pairings (image quality affects the average match time).

All the tests have been run on a PC with Java™, but using the exact same Java Card™ code downloaded in the card, since the second language is a subset of the first; in this way the same security performances are fully achievable even on the smartcard. We derived the matching time for the card application from the average time needed to match one single minutia on the card (measured directly in this environment), and multiplying for the minutiae number needed to stop the match (calculated in PC tests).

We have also compared our work on the Hybrid Database provided by the authors of [11], developed to be executed in a similar Java Card™ environment. Results show that we nearly halved the EER percentage of 0.8% achieved by that algorithm, obtaining a value of 0.48% (Table 1). Our algorithm is better also for the matching execution time: 1-8 seconds for nearly all of the matches (ours), against 11-12 seconds in [11].

It is important to point out that using a good quality enrollment image considerably improves the overall security performances: FAR1000 value can be reduced to about 5-6% as measured from other tests, mitigating also the partial overlapping problem. The hypothesis of having a good quality template is not too restrictive and it is easily applicable, since the enrollment phase is accomplished only one time at the release of the smartcard and the quality of the enrolling image can be easily checked. Therefore, using a good image at enrollment improves both reliability and speed performances.

5 Conclusion

In this paper we have proposed a new fingerprint matching algorithm tolerant to typical problems such as rotation, translation and ridge deformation. Our procedure achieves a very good speed performance for the Java Card™ platform restrictions: 1-8 seconds for most of the positive match tests. The high reliability, as determined from our anal-

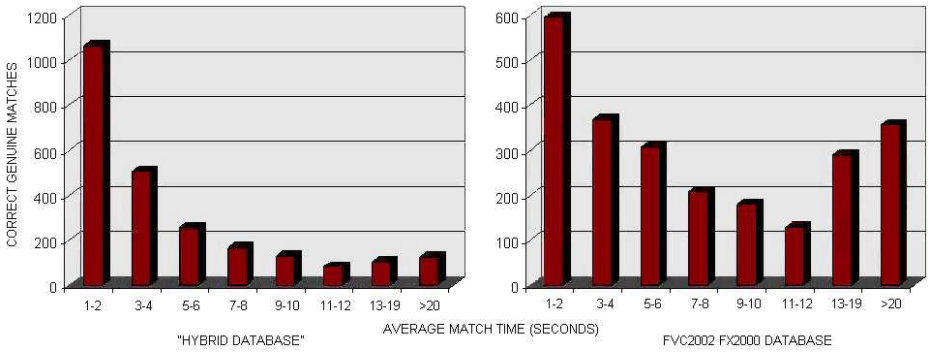

Fig. 4: Average smartcard match time on correct FRR tests, sampled in time intervals

ysis, can be further greatly improved using a good enrollment image, thus scoring a FAR1000 result of about 5-6%, which makes the algorithm implementation feasible in the live-scan applications for identity verification (like a MOC system). Our procedure is stopped as soon as the two templates are considered to belong to the same finger, and so the algorithm stops before in correct FRR tests and later in correct FAR ones, showing an asymmetric behaviour.

References

1. D. Maltoni, D. Maio, A.K. Jain, S. Prabhakar, *Handbook of Fingerprint Recognition*, Springer, 2003, ISBN 0-387-95431-7.
2. *User's Guide to NIST Fingerprint Image Software (NFIS)*, NISTIR 6813, National Insitute of Standards and Technology.
3. J.H.Wegstein, *An Automated Fingerprint Identification System*, U.S. Government Publication, Washington, 1982.
4. N. K. Ratha, R. M. Bolle, V. D. Pandit, V. Vaish, *Robust Fingerprint Authentication Using Local Structural Similarity*, IEEE 2000.
5. A. K. Jain, S. Prabhakar, L. Hong and S. Pankanti, *Filterbank-based Fingerprint Matching*, IEEE Transactions on Image Processing, Vol. 9, No.5, pp. 846-859, 2000.
6. T. Hatano , T. Adachi, S. Shigematsu, H. Morimura, S. Onishi, Y. Okazaki, H. Kyuragi, *A Fingerprint Verification Algorithm Using the Differential Matching Rate*, ICPR02, III volume: pp. 799-802, 2002.
7. C. Enrique Ortiz, *An Introduction to Java Card™ Technology*, Part 1-2-3, 2003.
8. D. Maio, D. Maltoni, R. Cappelli, J. L. Wayman and A. K. Jain, *FVC2002: Second Fingerprint Verification Competition*, Proc. of International Conference on Pattern Recognition, pp. 811-814, Quebec City, August 11-15, 2002.
9. *Java Card™ Biometric API White Paper (Working Document)*, Version 1.1, NIST/Biometric Consortium, 2002.
10. Y.S.Moon, H.C. Ho, K.L. Ng, *A Secure Card System with Biometric Capability*, IEEE Conference on Electrical and Computer Eng., Volume 1, pp 261-266, 1999.
11. T. Cucinotta, R. Brigo, M. Di Natale, *Hybrid Fingerprint Matching on Programmable Smart-Cards*, TrustBus 2004, Springer LNCS volume 3184/2004 p. 232.
12. P. B. van Wamelen, Z. Li, S. S. Iyengar, *A Fast Algorithm for the Point Pattern Matching Problem*, 2000.
13. S. Bistarelli, G. Boffi, F. Rossi, *Computer Algebra for Fingerprint Matching*, Proc. International Workshop CASA'2003, Springer LNCS vol. 2657 2003.