A new similarity digest search strategy applied to Minutia Cylinder-Codes for fingerprint identification

Vitor Hugo Galhardo Moia, Marco Aurélio Amaral Henriques

[1]Department of Computer Engineering and Industrial Automation (DCA)
School of Electrical and Computer Engineering (FEEC)
University of Campinas (UNICAMP)
Campinas, SP, Brasil 13083-852

[vhgmoia,marco]@dca.fee.unicamp.br

Abstract. *One challenging problem on the fingerprint realm is the identification of individuals over large databases, where the most similar template must be found. Approximate matching is used in digital forensic investigations to deal efficiently with large amount of data. We think it can also be used to identify similar fingerprints with compact representations and become a promising technique to speed up searches. In this paper, we explore this hypothesis and present MCC-HBFT, a new fingerprint identification strategy based on the approximate matching technique HBFT and the state-of-the-art fingerprint representation model MCC. We show how MCC-HBFT identify fingerprints and outperforms a commonly used indexing strategy in some public databases of the field.*

1. Introduction

Fingerprint identification is one of the most well-known and publicized biometric traits due to its interesting characteristics: uniqueness, consistency over time, easy acquisition, and low costs. However, one problem that remains is searching an unknown fingerprint over large repositories, which poses challenging obstacles regarding accuracy and efficiency. Identifying individuals in such case requires the comparison of the input fingerprint templates to every other template in the database, in an all-against-all fashion. This process, often called brute-force, is ineffective and becomes impractical for large sets.

One trivial solution to overcome the problem mentioned above is to reduce the total number of comparisons by prefiltering techniques, such as exclusive classification. The issues with this approach are the fixed number of classes, which is small, and that fingerprints are unevenly distributed among them. A more efficient solution could be the use of indexing schemes, where the fingerprint features represent the indexes. Features can be classified as global or local. The first category gives macro-level details of the ridge flow, such as the fingerprint class, a pattern of ridges and valleys on the surface of a finger. Local features can be minutia points, which are discriminative enough for the recognition task, and are composed by local ridge discontinuities. Minutiae can be of two types: terminations (ridge endings) and bifurcations.

The minutiae-based fingerprint representation method has been proposed by ANSI-NIST and includes the minutiae location and orientation (direction of the underlying ridge at the minutia location). The state-of-the-art fingerprint representation technique used to codify minutiae is the Minutiae Cylinder-Code (MCC) [Cappelli et al. 2010], which represents each minutia and the ones around it into a single cylinder. The MCC

cylinder is invariant for translation and rotation and robust to skin distortion and small feature extraction errors. These characteristics make MCC a good representation model for a minutiae-based fingerprint. Besides, comparing MCC features (cylinders) is very efficient since they can be represented as a bit vector and compared using XOR operations.

Even though comparing the two MCC templates is very efficient due to its bit-based representation, the comparison of large data sets is not. Given that two templates which were obtained from the same finger could have a considerable variability due to numerous reasons, such as rotation, pressure, noise etc, the problem at hand becomes finding the most similar template to the queried one in a broad set in the shortest time possible. A comparable situation is found in the digital forensics field, where approximate matching techniques are used efficiently to find similar data [Breitinger et al. 2014b].

In this work, we explore the use of approximate matching techniques to deal with the fingerprint identification problem. Our contribution is towards an efficient way of finding similar fingerprints templates over large databases. To this end, we present MCC-HBFT, a new fingerprint identification strategy based on the combination of the HBFT approximate matching technique and the state-of-the-art MCC representation model. Our results indicate that the merge of approximate matching and fingerprint techniques can be beneficial to the problem at hand. Furthermore, we show that MCC-HBFT outperforms a commonly used indexing scheme on some public databases of the field.

2. The Minutia Cylinder-Code (MCC) representation model

The state-of-the-art fingerprint representation model is the MCC technique. According to Cappelli, R. et al. [Cappelli et al. 2010], the MCC encodes each minutia extracted from a fingerprint (in the form of ISO/IEC 19794-2 [ISO/IEC-19794-2:2005 2005]) into a cylinder structure, corresponding to the spatial (cylinder base) and directional (cylinder height) relationships between a given minutia and the ones around it. We highlight that each minutia *m* is a triplet $\{m = x_m, y_m, \theta_m\}$, where x_m and y_m are the minutia location and θ_m the minutia direction (in the range of [0, 2π]).

MCC cylinders can be divided into sections, corresponding to a directional difference in the range $[-\pi, \pi]$; sections are discretized into cells, and each cell receives a value related by the accumulating contributions from the minutiae around it, which depends on both spatial and directional information [Cappelli et al. 2010].

One of the most interesting characteristics of MCC is the bit-based representation of fixed length. With a negligible loss of accuracy, the value of each cell can be represented as a bit. This way, a cylinder with n cells becomes a bit-vector of length n by linearizing its cells; a fingerprint template becomes a set of binary vectors. Other characteristics of MCC is that its cylinder structure is invariant for translation and rotation and robust against skin distortion and small feature extraction errors. These singularities make MCC extremely simple, reliable and fast for matching, being also suitable for indexing approaches and use with approximate matching techniques.

3. Related work

The basic concept of fingerprints and techniques for recognition, matching and identification can be found in surveys of the field [Peralta et al. 2015, Soni and Goyani 2018].

Here, we will present only indexing schemes. There are several approaches to deal with the fingerprint identification problem when searching a query fingerprint template against an extensive database. Among all schemes, we will focus on the minutia-based ones due to their better accuracy. Furthermore, we focus on the MCC techniques because of its bit-based representation since it is less computationally expensive and consumes less memory. We highlight that is not in the scope of this paper to present a detailed analysis of fingerprint indexing approaches. For this matter, a review is presented by Parmar, P. A. and Degadwala, S. D. [Parmar and Degadwala 2015]. We review only a few methods related to our proposal due to space restrictions.

The MCC-LSH [Cappelli et al. 2011] indexing approach has outperformed most indexing algorithms (minutia-based) on several public databases. It is based on the Locality Sensitive Hashing (LSH) technique and MCC representation model. The index structure corresponds to several hash tables, each having a particular hash function. For populating the index, each cylinder (bit-based implementation) from each fingerprint template of the database is hashed under the several hash functions and stored in their corresponding hash tables. When searching a fingerprint template, the process repeats, but instead of saving the cylinders in the table, we check for collisions with other cylinders. The more collisions we get, the more similar the templates are.

Other methods in the literature are variations of Cappelli's approach [Cappelli et al. 2011]. Wang, Y. et al. [Wang et al. 2014] use MCC and geometric hashing (Geo-MCC). Their work is extended in [Wang et al. 2015], where more compact binary hash codes are learned from MCC binary representations and used again with geometric hashing, but now combined with LSH (Geo-LSH). The authors reduced the cylinder size from 384 bits to only 24 bits, decreasing the number of hash functions and hash tables used by the system to store fingerprint templates. The benefit of their scheme comes with a reduction in accuracy but still is better than MCC-LSH according to their experiments. A similar approach is presented by Bai, C. et al. [Bai et al. 2018], where a learning-based algorithm is used to create shorter codes from MCC. However, the authors use these codes to create substrings and store them into multiple hash tables.

Su, Y. et al. [Su et al. 2016] presented another indexing scheme to speed up the search. They combine an improved LSH technique (using MCC) with a learning-based fingerprint pose estimation algorithm. The pose estimation aims to register fingerprints into a unified finger coordinate system and avoid unnecessary comparisons.

4. The MCC-HBFT fingerprint identification strategy

The MCC-HBFT is a fingerprint identification strategy that leverages the benefits of MCC fingerprint representation and the efficiency of the approximate matching field in finding similar fingerprint templates. We adapted the similarity digest search strategy HBFT [Breitinger et al. 2014c, Lillis et al. 2017], used by approximate matching tools to operate with MCC fingerprint templates. In the next subsections, we will describe how our strategy works and the necessary steps to insert and query fingerprint templates efficiently. The algorithms describing our approach as well as extra material with all tests performed can be found in our GitHub page [Moia and Henriques 2018].

4.1. The HBFT similarity digest search strategy

The Hierarchical Bloom Filter Tree (HBFT) is a data structure composed of several bloom filters arranged in a tree fashion. Bloom filters (BF) are space-efficient probabilistic data structures (bit vector) used to store a large amount of data by the use of hash functions. Upon receiving a query, they answer whether a given element belongs to a set or not. Initially, the BF is created empty (all positions (bits) are set to 0 (zero)). Inserting data on the filter requires performing k different hash functions over the given item and setting to 1 (one) all positions pointed out by the hashes. Searching for an element has the same process, but instead of setting the filter, we check for a bit 1 (one) in all positions. In the affirmative case, we know with a certain probability that the element belongs to the set; otherwise, we know for sure that it does not belong to it.

The HBFT approach uses several BFs so besides checking an element for membership, it can also identify which element matched the queried one. This way, several BFs are created and arranged in a tree fashion. In the first filter (root) at level 1, all elements of the data set are inserted. Then, the set is divided by half, and each part is inserted in a new BF, child of the root (level 2). This process of splitting the set and inserting its halves into new child BFs repeats for the subsequent levels until there is only one element per filter in the last level [Breitinger et al. 2014c].

It is important to highlight that HBFT is meant to identify similar objects. Instead of hashing data directly, HBFT uses the approximate matching tool sdhash/mrsh-v2 to extract *features* from the objects (small pieces of the object) and stores them in its structure. Later, the same *features* are used for identification purposes [Breitinger et al. 2014c]. In the end, objects that share a certain amount of *features* are considered similar.

Upon receiving a query request, the queried item is hashed using the k hash functions and first checked in the root BF node for a match (all positions in the filter set to bit 1). In a negative case, we stop the search right away and conclude there is no similar item in the set; otherwise, the search goes to the next levels until we reach the last level of the tree (there is a match, and we know the similar item) or all filters from the same level give a non-match (there is no similar item) [Breitinger et al. 2014c].

4.2. The proposed MCC-HBFT scheme

MCC-HBFT has the same working principle as HBFT, except with a few modifications and additional resources necessary for its operation with the MCC representation. Here, we explain the singularities of our approach.

4.2.1. Features and multiple trees

Just like HBFT, instead of inserting the template itself into the data structure, we insert small pieces of it (*features*). In an MCC template, there is one cylinder (fixed-size bit vector) representing each minutia. One possible candidate to *feature* is the cylinder itself, but for a more accurate version, we choose to work in the cylinder section level. Since we have s sections per cylinder, we will have s features for each cylinder.

The choice of working in the section level from the cylinder perspective have some advantages. We can reduce the *feature* size (s times) and the number of hash functions

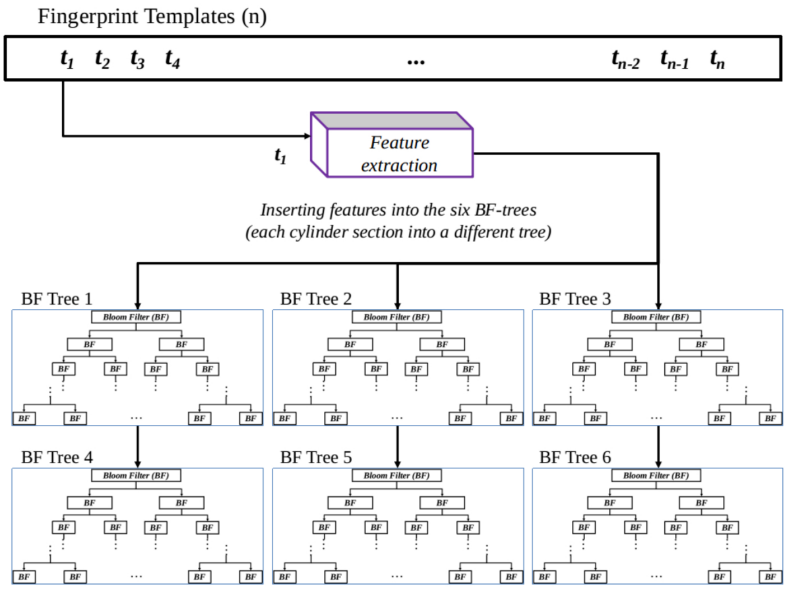

Figure 1. Inserting fingerprint templates into the MCC-HBFT strategy for MCC representations with six-cylinder sections.

used by inserting them into the BF structure for the same accuracy. The downside is that the number of *features* is multiplied by s, which will demand an increase of the BF size. Since BFs are space-efficient structures and MCC are compact representations, the increment is not significant. Another possible issue is how to distinguish a *feature* from one section to another to avoid misleading collisions. The solution adopted in this work for this problem is based on the same paradigm used by the HBFT approach: divide and conquer. We choose to use different HBFT data structures for storing the features of each section separately, one BF tree for each cylinder section, as shown in Fig. 1.

To get the number of features (z) for each BF tree structure, we use Eq. 1:

$$z = n \cdot c, \tag{1}$$

where n is the number of templates and c the average number of cylinders in a template.

4.2.2. Hash functions

One efficient way to create different hash functions for setting bits on bloom filters is to hash the given element with a cryptographic hash function (e.g., MD5, SHA-1 etc.) and split the result into k parts, where each piece corresponds to a different hash. However, in biometric systems, we have considerable variability in different fingerprint templates even though they belong to the same finger. Numerous factors can contribute to this fact, including displacement (same finger placed at different locations on the acquisition sensor), rotation, pressure, skin condition, noise etc. A common practice is to establish a threshold of the acceptable difference between two templates to consider them a match. This way, our problem here is to find *"similar"* templates, not precisely identical ones.

Due to the characteristics of the problem at hand, using common cryptographic hash functions on fingerprint *features* will not produce good results since they can not

stand minor changes in the input. Changing a single bit in the data will create an entirely different output (avalanche effect). For this reason, we need a different kind of function. Here, we will use a new version of the functions presented by Cappelli, R. et al. [Cappelli et al. 2011]. Given a fingerprint feature (cylinder section), we will use k_{max} functions where each of them will randomly choose b bits from the each feature; next, a cryptographic hash function is used to create the hash value from the selected bits. In the end, each feature will have k_{max} hash values. Since we expect that some bits may differ from one template to another due to biometric constraints, we only require that k_{min} hash functions ($k_{min} < k_{max}$) match against the BF for taking the feature as similar. Using many hashes per feature and establishing a minimum number of functions (k_{min}) to have a match, allows our approach to detect similar fingerprint templates.

4.2.3. Bloom filters size and number of bits of the hash functions

The BF size depends on three factors: number of elements inserted in the filter, false positive rates, and number of hash functions. Since our structure is arranged in a tree fashion, we can create fixed-size or variable-size filters. Considering we want to keep the same number of hash functions and false positive rate for all filters, the latter option is more space-efficient because in each level we have half elements than the previous one inserted in the filter, which decreases the BF size. The root filter (level 1) is the largest one, containing all elements of the data set. Its children and all other filters from other levels have half of their parent size. Given that we established a maximum and a minimum number of hash functions (section 4.2.2), we had to change the root BF size formula slightly [Breitinger et al. 2014a]. The adapted formula is shown in Eq. 2.

$$m_{root} = \frac{-k_{max} \cdot z}{ln(1 - \sqrt[k_{min}]{p})} \quad (bits) \tag{2}$$

where m_{root} is the root BF size, z the number of features, k_{max} and k_{min} are the maximum and a minimum number of hash functions, respectively, and p the false positive rate.

Given the size of the root BF (largest filter of the entire structure), we can compute the number of bits (b) necessary for the hash functions address the bits into BFs using Eq. 3. For efficiency purposes, we compute the feature hashes once and discard one bit per level when working with filters from other levels, since BFs have half of its parent size.

$$b = \lceil log_2(m_{root}) \rceil \tag{3}$$

4.2.4. Final score

When searching for a template T_i, for each feature, we count the number of hashes that matched against the queried BF and, in case we have at least k_{min} matches, that feature is said to be part of the filter; otherwise, we drop it; then, we move on and search for the next feature in the same BF. We stop the search in this particular BF when $hits_{min}$ features are found, or we are out of features. In the first case, we assume a similar template lies in this BF, and we can continue searching in the next levels of the tree. In the second case, we understand there is no similar template in this BF nor the subsequent levels of this node.

Once we have reached the last level of the tree (single fingerprint template per filter), we count the number of features found in that filter. Additionally, we determine a match score for the queried template and the one belonging to the BF. Eq. 4 shows our score formula, which seeks to normalize the number of matching features by the average number of features found in the queried template T_i (F_Q) and BF template BF_j (F_{BF}).

$$score(T_i, BF_j) = \frac{H_M}{(k_{max} \cdot (F_Q + F_{BF}))/2}. \quad (4)$$

It is important to highlight that H_M corresponds to the number of matching hashes of all features from T_i that had at least k_{min} hashes values matched.

4.2.5. Additional resources

Here we present additional resources integrated to MCC-HBFT to improve its efficiency.

Fingerprint classes: To reduce the number of template comparisons, we added a new component to our approach: fingerprint classes. Each bloom filter has a flag indicating the classes of fingerprints that lies on it. We adopted the six-class model used by NIST: Arch (A), Tented Arch (T), Left Loop (L), Right Loop (R), Scar (S) and Whorl (W) [Ko 2007]. One can use NIST PCASYS (Fingerprint Pattern Classification) [Ko 2007] system to predict the class of a fingerprint or any other method, including establishing it manually. The classes help to decrease the number of unnecessary comparisons. When creating the BF-tree structure, we group the fingerprints by classes and insert them in the same or near BFs. In the search process, when the queried fingerprint template belongs to a different class from the ones of a particular filter, we stop the search in that BF and all subsequent levels. Since some fingerprints may have a pattern that classifies it in more than one class, we allow an assignment of at most two classes per fingerprint.

Compatible function: Like other indexing approaches [Cappelli et al. 2011, Bai et al. 2018], a compatible function is used to narrow down the search. Two minutiae m_1 $(x_{m1}, y_{m1}, \theta_{m1})$ and m_2 $(x_{m2}, y_{m2}, \theta_{m2})$, are only considered a match if their angular difference $d_\theta(\theta_{m1}, \theta_{m2}) > \sigma_\theta$ and euclidean distance $d_{xy}((x_{m1}, y_{m1}); (x_{m2}, y_{m2})) > \sigma_{xy}$. According to Cappelli, R. et al. [Cappelli et al. 2011], this is done to ensure a minimum rotation and displacement between them. In the proposed strategy, the minutiae attributes (x_m, y_m, θ_m) are stored into a hash table along with a minutia identification. Each fingerprint template has an exclusive table for keeping its attributes.

As mentioned before, the problem handled here is to deal with similarity cases, where fingerprint templates are not identical. For this reason, we use k_{max} and k_{min} as a maximum and a minimum number of hash functions, respectively, to set and query bits in our BF tree structure. This allows us to have different bits between the database template and the queried one. The same problem applies for creating indexes for storing the minutiae attributes (represented by k_{max} hash values) into the hash table. Since each feature has several hash values and we allow that only part of them match, we can not create a single index for a feature to insert it into the table. If we use all hashes to derive

an index value, any similar feature that has at least one different bit will probably have a different index and will not be correlated to their similar ones.

The solution proposed here follows the idea adopted in section 4.2.2. We use all k_{max} hash values from a feature to derive many indexes and, for each one, we insert the feature in the corresponding hash table bucket (we only store the feature identification, while its attributes (x_m, y_m, θ_m) are stored elsewhere to avoid redundancy and save memory). Upon a query request, we check the hash table for the features that collide at least k_{min} times with the queried one before taking their attributes and performing the compatible function. Only if the function's result is true, we count a match for that feature.

4.3. Creating the MCC-HBFT data structure: The preparation phase

The preparation phase, often called offline stage, consists of creating the MCC-HBFT data structure and inserting all database fingerprint templates on it. First, we create s (cylinder sections) empty BF trees and a set of hash functions for each tree according to the cylinder section it lies on. The next step is grouping the fingerprint templates according to their classes and insert them into MCC-HBFT. Since we could have more than one class per fingerprint, we classify the fingerprints prioritizing the first class.

The insertion process goes as follow: The first template of the first group is inserted into BF_x (where $x = 2^{L-1}$ is the number of the BF in the tree at level L), the first filter of the last level of the tree, and all its parents' nodes. In BF_x, we insert the template features, some attributes, and create a new hash table for keeping the minutiae information. Besides, we set the class tag of the current BF and all its parents according to the given template classes. Then, we proceed to the next template and insert it into BF_{x+1}, the second BF of the last level and all its parents. Again, we set the classes of the BFs accordingly and proceed until we have inserted all templates. Fig. 2 shows the MCC-HBFT structure of a single BF tree.

We highlight that the last level of each tree stores only one template per filter (which has several cylinders, thus having several features). Furthermore, we keep some fingerprint attributes for later identification and comparison purposes, such as an identification (ID), fingerprint classes (class 1 and 2), and number of features. A hash table for each template is also stored to keep the minutiae information.

4.4. Searching fingerprint templates: The operational phase

The operational phase, also known as online stage, follows the preparation and consists of performing searches on the MCC-HBFT structure. Given the MCC template and its class, the proposed strategy performs the feature extraction process and look for each feature according to their cylinder section in the BF trees. A match is found when at least $hits_{min}$ features are located in a BF. In the case of a non-match in the root filter, we discard that feature and move to the next one. When matching, we go further and look for matches in the BF children of that node. We proceed with the search until we reach a filter on the last level of the tree or we have a non-match in the current node and all other ones.

In the last level of the tree, we have one template per filter. Besides looking for all features, we also compute a match score for the queried template (sec. 4.2.4), and after searching into all s trees, we summed up the results and present an ordered list (by match score) with all possible candidates to similarity to the queried template.

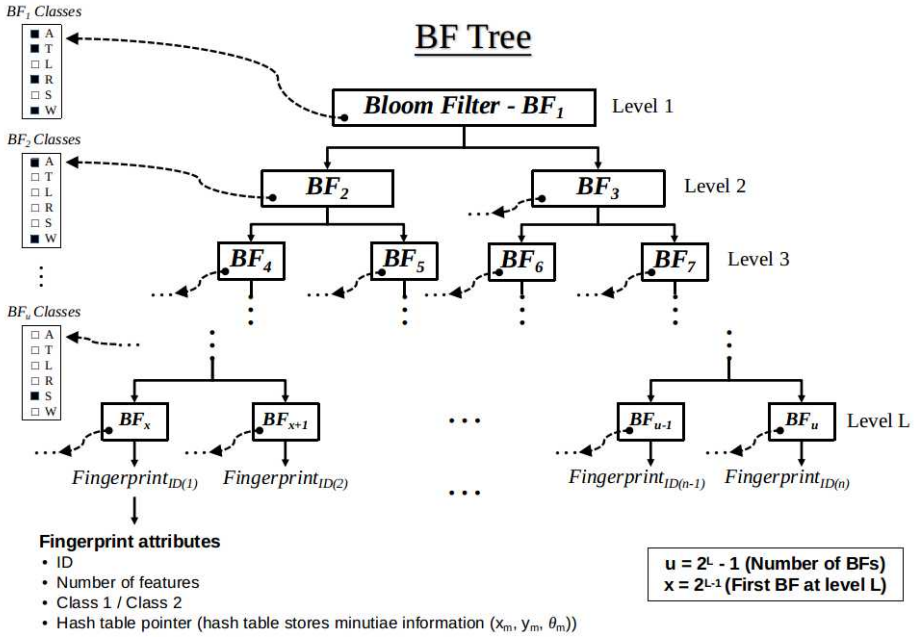

Figure 2. MCC-HBFT: Single BF tree structure

5. Assessment

5.1. Evaluation setup, databases, and parameters

The tests performed in this paper used a machine running a dual boot of Elementary OS 0.4.1 Loki 64-bit (built on Ubuntu 16.04.2 LTS) and Microsoft Windows 10 64-bit, with an i7-5500U CPU @2.40GHz processor, 8 GB of memory, and NVIDIA GeForce 920M. The proof of concept MCC-HBFT was developed using C language.

The performance of MCC-HBFT was measured using public domain fingerprint databases, such as the NIST Special Databases 4 [NIST 2018] and some FVC databases ([FVC2002 2018] and [FVC2004 2018]). All database details are shown in Tab. 1.

Table 1. Detailed information of public databases used in the experiments

Databases	Size	Resolution	Subject	Impressions	Sensor	Format
NIST DB4	512x512	500dpi	2000	2	Ink-rolled	PNG
FVC2002 DB1a	388x374	500dpi	100	8	Optical	TIF
FVC2002 DB3a	300x300	500dpi	100	8	Capacitive	TIF
FVC2004 DB1a	640x480	500dpi	100	8	Optical	TIF

The tests presented here followed the strategy used in the literature [Bai et al. 2018, Cappelli et al. 2011]. For the NIST database, the first fingerprint impression of DB4 is used for indexing and the second one for querying. On the FVC database, the first impression was used for index and remaining seven for querying. To estimate some parameters of MCC-HBFT, we used 600 fingerprints from NIST DB4 (500 first impressions for index and 100 second impressions for query) and 200 fingerprints from FVC2002 DB1a (25 first impressions for index and 175 impressions for query).

The fingerprint minutiae extraction was performed in two different ways. For the NIST fingerprints, we used the open source NBIS[1] software. For all FVC sets, we used a set of manually extracted minutiae (FM3) [Kayaoglu et al. 2013]. Next, the minutia information of both databases is used to create the MCC representation using MCC SDK v2.0[2], creating a 384-bit-based template for each fingerprint. We adopted the same parameters as reported in [Cappelli et al. 2011] for the cylinder creation.

We limited the comparison of our strategy to only the state-of-the-art index structure MCC-LSH [Cappelli et al. 2011] since it has a free implementation available by MCC SDK v2.0. All other indexing schemes that work with MCC do not have their source code available for comparison. Most approaches only compare their proposals to MCC-LSH, the only readily available one on the literature. The tests performed here used the same MCC and LSH parameters available in [Cappelli et al. 2011]. A C# routine was developed in the Windows operating system to create the index structure, perform the queries, and then consolidate the results.

MCC-HBFT makes use of fingerprint classes. Here, we adopted the free NIST PCASYS [Ko 2007] software to perform the class assignment for first class and a manual adjustment to insert the second class, when necessary. Since the focus of this research is based on the use of classes to reduce the number of comparisons and not in the classification process itself, we are not interested whether PCASYS assigns a right class to a fingerprint or not. We are only concerned that two mate fingerprints have the same class.

Other parameters of MCC-HBFT include the number of hash functions k_{max} and k_{min}, established after several experiments over the test databases (presented next). We also defined $hits_{min}$=20%, which is the number of feature matches to conclude that a similar feature is inserted in a BF. This value is a proportion of features found in the filter by the total number of queried features, also found experimentally. The other parameters are s = 6 and SHA-1 as cryptographic hash function. The values used in the compatible function are: $\sigma_\theta = \pi/4$ and σ_{xy} = 256.

5.2. Results

The evaluation of the accuracy and efficiency of fingerprint indexing schemes is measured by the trade-off between Error Rate (ER) and Penetration Rate (PR). The first metric is based on the number of queried fingerprints not found in a search, while the latter one corresponds to the proportion of the database explored by the indexing approach in a query. The best scenario is the smallest possible error to the lowest penetration rate.

In our experiments, we have randomly generated the hash functions (bits selected in the cylinders) each time we ran a full trial with MCC-HBFT. Even though we require fixed hash functions to always produce the same results, our tests changed it because *i.* the size of the databases is different, requiring more or less bits for each function; *ii.* we wanted to verify the impact of "good" and "bad" bit selections. We also have chosen three versions of MCC-HBFT using different numbers of hash functions to find the best cost/benefit setting. We highlight that the more hash functions used, the higher the costs (time) are. The settings include a low, mid, and high cost versions.

[1]NIST Biometric Image Software (NBIS) v5.0.0, http://www.nist.gov/itl/iad/ig/nbis.cfm
[2]MCC SDK v2.0, http://www.biolab.csr.unibo.it/mccsdk.html

Determining the values of k_{max} and k_{min} required the two test databases. First, we set $k_{max} = x \cdot k_{min}$, where x is a variable controlling the level of changes accepted over two templates. The best results for x were different for FVC and NIST. In the former one, the best values varied between $x = 5$ and $x = 6$, while the other had $x = 3$ and $x = 4$. These results helped us to find the different settings of our approach.

We have run each experiment 10 times and selected the worst, average, and best case scenarios, and compared them to MCC-LSH. To be fair, we have compared our approach under its average case scenario. Fig. 3 shows the results for the FVC2002-DB1a set with the average results got from MCC-HBFT using the three different settings. We can see that all MCC-HBFT versions had better outcomes compared with MCC-LSH on average. Even though MCC-LSH had better results for a low penetration rate ($PR \leq 7.0\%$), MCC-HBFT settings presented better results from this point on. Besides, our low-cost version reached $ER = 0\%$ with $PR = 39\%$, while MCC-LSH with $PR = 48\%$.

Fig. 4 shows the three scenarios of the low-cost setting ($k_{max} = 20/k_{min} = 4$) and the baseline MCC-LSH. The worst case had $ER = 0.28\%$ ($PR = 100\%$). However, the average scenario had $ER = 0.0\%$ and $PR = 39\%$. We chose to show this particular setting because it had the worst performance with respect accuracy of all three and because it is a low-cost version. Adopting a more accurate hash function means increasing the number of hashes, as shown in Fig. 3, where the high-cost version ($k_{max} = 72/k_{min} = 12$) performed better than the others ($ER = 0.0\%$ and $PR = 28\%$).

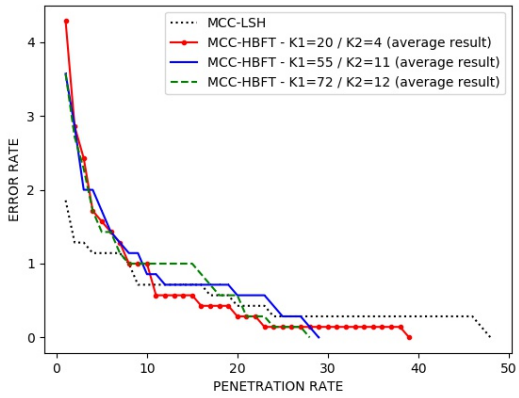

Figure 3. Performance evaluation on FVC2002 DB1: Average case scenario of three different MCC-HBFT versions

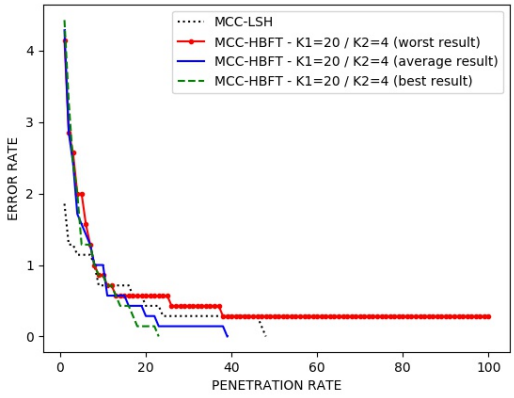

Figure 4. Performance evaluation on FVC2002 DB1: The worst, average, and best case scenarios (low-cost)

Fig. 5 shows the experiments under FVC2002-DB3 database. The low-cost version ($k_{max} = 20/k_{min} = 4$) performed worst, while the others competed with MCC-LSH but had a higher error rate. It is important to highlight that none of the approaches reached $ER = 0\%$. MCC-LSH had $ER = 1\%$. On the other hand, over FVC2002-DB1 (Fig. 6), MCC-HBFT beats the state-of-the-art proposal significantly, except for the mid-cost version with $PR \geq 82\%$, when it ends with a superior error rate.

The tests using NIST BD4 database have adopted different parameters for the hash functions, but we still use the idea of the three settings: a low, mid, and high cost versions. Fig. 7 shows the results of three settings and the baseline MCC-LSH. Even with

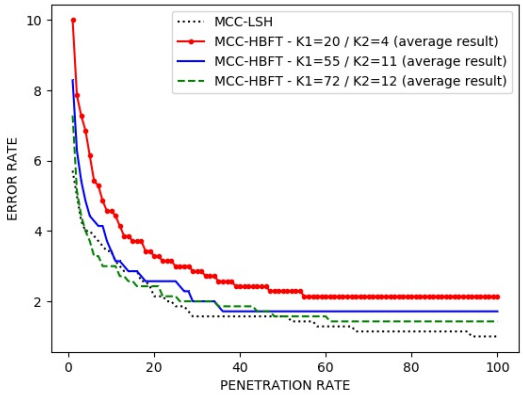

Figure 5. Performance evaluation on FVC2002 DB3: Average case scenario of three different MCC-HBFT versions

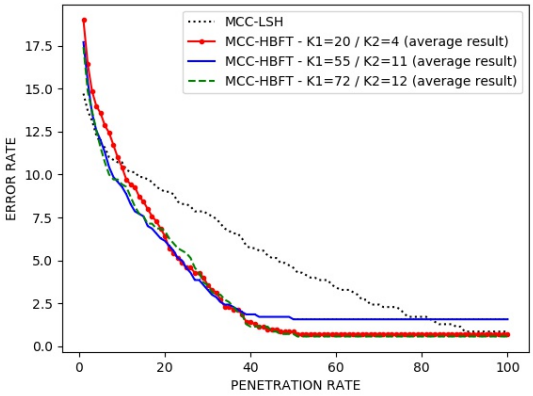

Figure 6. Performance evaluation on FVC2004 DB1: Average case scenario of three different MCC-HBFT versions

the low-cost version ($k_{max} = 18/k_{min} = 6$), we had better results than MCC-LSH from $PR \geq 7.95\%$ and reached $ER = 0.0\%$ before it (with $PR = 41.3\%$ in comparison to $PR = 100.0\%$ of MCC-LSH). MCC-HBFT presented similar results over NIST database in all of its executions. The worst case scenario of all experiments always had results close to the best one, except in one case, where the search could not find all templates in the search, as illustrated in Fig. 8. In this setting, we had $ER = 0.05\%$, which corresponds to a single fingerprint not found. The other two experiments were successful in finding all candidates with $PR = 42.05\%$ (on average) for their worst-case scenario.

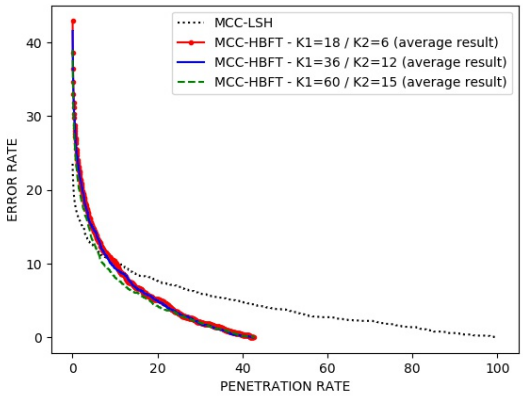

Figure 7. Performance evaluation on NIST DB4: Average case scenario of three different MCC-HBFT versions

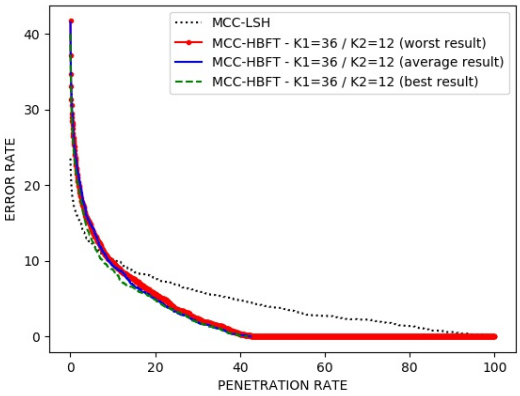

Figure 8. Performance evaluation on NIST DB4: The worst, average, and best case scenarios (mid-cost)

5.3. Discussion

MCC-HBFT can be used in different settings. We expected that the more hash functions we use, the higher the accuracy. However, the benefits were quite small given high k_{max} values. In general, the results for the different settings were quite similar. In opposition, the time costs increased significantly since we had to perform many more hashes per

feature. The low-cost version appeared as the most cost/benefit combination. Future work encompasses a cost/benefit analysis of different MCC-HBFT settings.

To understand the hash functions parameters changing from FVC to NIST database, we analyzed the details of each set and summarized it in Tab. 2. The number of minutiae per template in each database varied significantly. NIST database has more minutiae per template, allowing a smaller proportion of k_{max} and k_{min} since it has many minutiae to compare between two templates (more chances to get matches). The FVC sets have fewer minutiae, decreasing its chances for matching. For this reason, the difference between k_{max} and k_{min} must be higher.

Table 2. Detailed information of public databases used in the experiments

Database	**FVC**			**NIST**
Parameters	**2002DB1**	**2002DB3**	**2004DB1**	**DB4**
Number of minutiae (avg) per template	35.02	21.49	38.86	120.0
Max. number of minutiae per template	81	46	77	237
Min. number of minutiae per template	5	2	8	13
Number of bit 1 in each minutiae (avg)	11.0681	9.1413	10.5342	12.7070

One difference to call attention between NIST and FVC databases results is the error rate of small values of database penetration. We had significantly lower errors for FVC than NIST set. We attribute that to the extraction minutia process, which was manually and carefully done for FVC database and automatic for NIST set. The quality of the minutiae possibly influenced the results, but here we are more concerned with the performance of our approach against the state-of-the-art scheme. Since we used the same minutiae for both approaches, we believe this will not affect the relative results.

6. Conclusion

Identifying individuals over large databases using fingerprints is a challenging problem, mainly due to efficiency reasons. The approximate matching techniques are efficient solutions in digital forensics to find similar content, the same issue faced by the fingerprint field. In this work, we have presented MCC-HBFT, a new fingerprint identification strategy that leverages the efficiency of the approximate matching field and the accuracy of the state-of-the-art fingerprint representation MCC. We showed how our strategy works and outperforms a commonly used fingerprint indexing approach on public domain databases. Future work encompasses a cost/benefit analysis of different MCC-HBFT settings. We also plan to analyze the use of more efficient hash functions to decrease the overall costs without impacting the accuracy.

7. Acknowledgment

This work is partially supported by CAPES FORTE Project (23038.007604/2014-69).

References

Bai, C., Wang, W., Zhao, T., and Li, M. (2018). Fast exact fingerprint indexing based on compact binary minutia cylinder codes. *Neurocomputing*, 275:1711–1724.

Breitinger, F., Baier, H., and White, D. (2014a). On the database lookup problem of approximate matching. *Digital Investigation*, 11:S1–S9.

Breitinger, F., Guttman, B., McCarrin, M., Roussev, V., and White, D. (2014b). Approximate matching: definition and terminology. *NIST Special Publication*, 800:168.

Breitinger, F., Rathgeb, C., and Baier, H. (2014c). An efficient similarity digests database lookup-a logarithmic divide & conquer approach. *JDFSL*, 9(2):155.

Cappelli, R., Ferrara, M., and Maltoni, D. (2010). Minutia cylinder-code: A new representation and matching technique for fingerprint recognition. *IEEE TPAMI*, 32(12):2128–2141.

Cappelli, R., Ferrara, M., and Maltoni, D. (2011). Fingerprint indexing based on minutia cylinder-code. *IEEE TPAMI*, 33(5):1051–1057.

FVC2002 (2018). The second fingerprint verification competition. `http://bias.csr.unibo.it/fvc2002/`. Accessed 2018 Jun 20.

FVC2004 (2018). The third international fingerprint verification competition. `http://bias.csr.unibo.it/fvc2004/`. Accessed 2018 Jun 20.

ISO/IEC-19794-2:2005 (2005). Information technology - biometric data interchange formats - part 2: Finger minutiae data.

Kayaoglu, M., Topcu, B., and Uludag, U. (2013). Standard fingerprint databases: Manual minutiae labeling and matcher performance analyses. *arXiv preprint arXiv:1305.1443*.

Ko, K. (2007). User's guide to nist biometric image software (nbis). Technical report.

Lillis, D., Breitinger, F., and Scanlon, M. (2017). Expediting mrsh-v2 approximate matching with hierarchical bloom filter trees. In *ICDF2C*, pages 144–157. Springer.

Moia, V. H. G. and Henriques, M. A. A. (2018). MCC-HBFT: A fingerprint identification strategy. `https://github.com/regras/mcc-hbft`. Accessed 2018 Jun 30.

NIST (2018). Special database 4. `https://www.nist.gov/srd/nist-special-database-4`. Accessed 2018 Jun 20.

Parmar, P. A. and Degadwala, S. D. (2015). Fingerprint indexing approaches for biometric database: A review. *IJCA*, 130(13):0975–8887.

Peralta, D., Galar, M., Triguero, I., Paternain, D., García, S., Barrenechea, E., Benítez, J. M., Bustince, H., and Herrera, F. (2015). A survey on fingerprint minutiae-based local matching for verification and identification: Taxonomy and experimental evaluation. *Information Sciences*, 315:67–87.

Soni, U. A. and Goyani, M. M. (2018). A survey on state of the art methods of fingerprint recognition. *IJSRSET*, 4.

Su, Y., Feng, J., and Zhou, J. (2016). Fingerprint indexing with pose constraint. *Pattern Recognition*, 54:1–13.

Wang, Y., Wang, L., Cheung, Y.-m., and Yuen, P. C. (2014). Fingerprint geometric hashing based on binary minutiae cylinder codes. In *ICPR*, pages 690–695. IEEE.

Wang, Y., Wang, L., Cheung, Y.-M., and Yuen, P. C. (2015). Learning compact binary codes for hash-based fingerprint indexing. *IEEE TIFS*, 10(8):1603–1616.