Towards Real-Time Privacy-Preserving Minutiae Matching

Julia Mader
Julia.Mader@ait.ac.at
AIT Austrian Institute of Technology
Vienna, Austria

Florian Wohner
Florian.Wohner@ait.ac.at
AIT Austrian Institute of Technology
Vienna, Austria

Laurenz Ruzicka
Laurenz.Ruzicka@ait.ac.at
AIT Austrian Institute of Technology
Vienna, Austria

Thomas Loruenser
Thomas.Loruenser@ait.ac.at
AIT Austrian Institute of Technology
Vienna, Austria

Abstract

While biometric data, such as fingerprints, are increasingly used for identification and authentication, their inability to be revoked once compromised raises privacy concerns. To address these concerns, this work explores the use of Multiparty Computation (MPC), which enables secure computations on encrypted data for privacy-preserving fingerprint matching. Despite MPC's well-known drawback of slowing down computation, recent advances have made it a viable option for real-world applications. Our research focuses on implementing and optimizing a minutiae-based fingerprint matching algorithm with MPC that addresses the challenge of preserving privacy while ensuring reasonable computation times. Furthermore, this work presents a privacy-preserving implementation of SourceAFIS using the MP-SPDZ framework. In addition, we demonstrate the adaptability of the MPC implementation by integrating different minutiae extractors, leading to significant enhancements in error rates. These findings underscore the feasibility and effectiveness of using modern MPC techniques for privacy-preserving fingerprint matching and paves the way for further optimizations.

CCS Concepts

• **Security and privacy → Privacy-preserving protocols**; **Biometrics**; *Privacy protections*; *Information-theoretic techniques.*

Keywords

Fingerprint Matching, Minutiae Extractors, Minutiae Matcher, Multiparty Computation, Privacy Enhancing Technologies

ACM Reference Format:
Julia Mader, Florian Wohner, Laurenz Ruzicka, and Thomas Loruenser. 2024. Towards Real-Time Privacy-Preserving Minutiae Matching. In *Proceedings of the 23rd Workshop on Privacy in the Electronic Society (WPES '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3689943.3695049

WPES '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1239-5/24/10
https://doi.org/10.1145/3689943.3695049

1 Introduction

Biometric data such as fingerprints, retina or facial characteristics are increasingly used as a promising replacement for conventional identification and authentication such as passwords or tokens. While passwords can be changed if compromised, biometric data, with its unique, permanent, and difficult-to-forge characteristics, cannot be altered once leaked. Furthermore, there are privacy concerns about the inevitable data collection, storage and potential misuse of personal and unique physical characteristics [16]. These disadvantages clearly demonstrate the importance of including strong protections against deliberate and accidental disclosure or misuse of biometric data. For this very reason, fingerprints are typically only scanned, stored and processed locally and on dedicated hardware, i.e., secure elements. However, this approach also limits application to local authentication and prevents more advanced use cases.

An alternative approach to using special hardware for privacy-preserving fingerprint matching is based on multiparty computation (MPC) [11]. MPC allows for secure computations on encrypted data without decryption, thus mitigating the security risks of exposing sensitive information during processing. MPC ensures privacy by allowing a set of parties to collectively compute a function over their private inputs without disclosing those inputs to any other party involved in the computation [6].

However, it should be noted that MPC comes with the drawback of significantly slowing down computation and therefore the matching process [10]. While this is not a problem for matchers based on feature vector comparisons, as the matching process of computing the Euclidean distance can be also performed quickly in the encrypted domain [2], it poses a significant challenge for the more complex minutiae-based matchers [3]. However, in recent years, there have been significant advancements in MPC protocol design and implementation, making it ready for real-world application. In this work we show that the realization of a fully-fledged privacy preserving minutiae-based matcher with practical efficiency is actually feasible.

1.1 Motivation and Use Cases

Our work is motivated by a novel use case identified in collaboration with the UN to improve border safety at airports, which requires remote matching capabilities instead of relying solely on local matching [17]. For this solution, we envision fingerprints being matched against No-Fly Lists at airport security checks. The

idea is that in addition to verifying the traveler's passport data, their fingerprints are also compared to national and international No-Fly Lists available at various law enforcement agencies. If a match is found, the traveler is not allowed to board the flight.

MPC is ideally suited for the use in this scenario because it supports both objectives. First, it enables the confidential pooling of No-Fly Lists among organizations that may be unwilling to openly share them. Second, it ensures the privacy of regular travelers by keeping their biometric data confidential, thus preventing them from being subjected to mass surveillance. Only in the event of a match will the relevant data be disclosed to initiate further investigations, but fingerprints of regular travelers will never be disclosed.

More concretely, the data flow works as follows: First, the individual No-Fly Lists of different organizations are secret-shared (encrypted) and distributed among servers, e.g., three servers in this case. This corresponds to pooling input data in a way that individual servers do not learn the data they are holding, but only see encrypted data. Then, if a fingerprint is scanned, a so-called template is generated and also encrypted (secret-shared) and sent to the severs. Once the servers have received the encrypted parts, they are able to compute the result of the matching function in a joint MPC protocol and only reveal the result, a predicate in this case. If no match was found in the database, the individual at the security check preserves their privacy, because the server does not learn anything about their identity. In case of a match, an alarm is triggered, and further handling of the suspect can be initiated.

However, given the highly time-critical nature of this application, and considering that MPC significantly slows down computations, it prompts the question of whether it is feasible to compute matching results obliviously within a reasonable timeframe. The waiting time for travelers should not be extended due to the additional check. Our research objective is to implement and optimize a fully featured fingerprint matching algorithm using MPC to evaluate the readiness of current protocols for deployment in the given use case.

1.2 Related Work

The best performance for encrypted fingerprint matching so far has been achieved by feature vector-based matchers, owing to their straightforward structure and encryption-friendly nature. In fact, this approach only requires computing the Euclidean distance between two vectors of reasonable length. Both [2] and [4] have utilized this approach in their research. However, benchmarks in [11] showed that the matching accuracy of these techniques is very poor compared to minutiae-based approaches.

Fălămaş et al. [7] presented their research on privacy-preserving password and iris authentication with MPC and secret sharing, using feature vector comparison for iris authentication. While this method of comparing feature vectors is effective for iris and facial recognition [1], most fingerprint matchers remain minutiae-based, as they outperform feature-based fingerprint matchers. However, there have been some recent advancements in the field using Deep Neural Networks for the extraction of the feature vector [13], which could be relevant for future implementations.

There have also been implementations of privacy-preserving minutiae-based matchers. Blanton and Gasti [3] encrypted FingerCode and an iris code, as well as a minutiae-based fingerprint matcher, using garbled circuits and homomorphic encryption. While they did not include a review of their matcher's accuracy, certain conclusions can be drawn from the structure of the matcher. Since the comparison of two minutiae is not rotation- or translation-invariant and is calculated by computing their Euclidean distance, the accuracy of the matcher cannot surpass that of a feature vector-based one. In [15] a minutiae-based fingerprint matcher using homomorphic encryption was presented, but still using a very simplistic fingerprint matcher algorithm.

In essence and to the best of our knowledge, no fully fledged MPC implementation for state-of-the-art fingerprint minutiae-based matcher has been presented. Moreover, the problem was believed to be too computationally intense for MPC. In this work we show the contrary. By careful analysis and optimization of the algorithm we were able to reach matching times in the seconds regime.

1.3 Contribution and Outline

Our research focuses on achieving an optimized implementation of multiparty computation of a previously existing minutiae-based fingerprint matching algorithm with state-of-the-art matching accuracy. For our model implementation with MPC we chose the minutiae-based fingerprint matcher SourceAFIS [19] as its underlying algorithm is available as open source and showed good results in terms of accuracy and speed in previous tests. While a first preliminary evaluation of MPC-based template matching using MPyC [14] was presented in [11], it only achieved runtimes in the range of hours, which is not suitable for real-time authentication. However, it contained interesting optimization approaches which we significantly extend in this work, leading to an optimized model implementation utilizing the MPC framework MP-SPDZ [9].

All algorithms are evaluated using fingerprint databases from the FVC2000: Fingerprint Verification Competition [12]. Given the number of false matches (FM), false non-matches (FNM), correct matches (CM), and correct non-matches (CNM), all excluding self-matches, we can use the False-Match-Rate (FMR; defined as FM/(FM+CNM)), False-Non-Match-Rate (FNMR; defined as FNM/(FNM+CM)) and Equal-Error-rate (EER; the point at which FMR and FNMR are equal) for evaluation. These rates provide valuable insights into the accuracy of the fingerprint matching algorithms. For the purpose of this research, the implementation will be considered successful if it can complete the matching process within a time frame of less than 10 seconds.

In this work, we build on [11] and further optimize the MPC implementation of SourceAFIS, a minutiae-based matcher. In particular the contributions are the following:

- The main algorithms steps of the SourceAFIS matcher were carefully analyzed, implemented and optimized down to the circuit level using MP-SPDZ for final benchmarking.
- We analyze the effects of the different optimization techniques in detail and measure their impact on matching efficiency.
- Furthermore, our contribution also entails an accuracy comparison and optimization of the pre-processing phase by utilizing different minutiae extractors, which are all compatible with our MPC implementation.

The remainder of the paper is organized as follows. In Section 2 we present the minutiae matcher of SourceAFIS. In Section 3 we analyze different minutiae extractors to compare their performance and compatibility with the SourceAFIS matcher. In Section 4 we discuss our ideas to optimize the algorithm for the implementation with MPC and evaluate their performance. In Section 5 we present our implementation with MP-SPDZ and most relevant benchmark results. In Section 6 we summarize our findings and present our future work.

2 Privacy Preserving Minutia Matching

The primary focus of this work is the adaptation of a minutiae-based fingerprint matcher for MPC, i.e., to demonstrate a privacy preserving SourceAFIS [19] implementation with Multiparty Computation. The used algorithm was not implemented from scratch, but partly taken from Robert Vazan's SourceAFIS implementation in Java. Especially the feature extraction could be used as is, since it is intended to run locally on the scanner side and not in the encrypted back-end. However, the core part of the matching functionality had to be substantially refactored and rewritten to achieve reasonable runtimes with MPC, although we did not deviate from the high-level concept of the matcher regarding the computation of matching scores.

The goal of our research is to generate an understanding of the complications involved in implementing these complex algorithms using MPC and to provide optimizations as well as estimates for best achievable runtimes and accuracy with current technology.

2.1 Minutiae Matcher (SourceAFIS)

SourceAFIS is a commercially used and open-source minutiae-based fingerprint matcher by Robert Važan [19]. The algorithm can be divided into two parts: feature extraction and fingerprint matching.

During feature extraction, SourceAFIS reads in an image of a scanned fingerprint and extracts its minutiae to store them in the so-called template. The minutiae are saved with their type, namely ending or bifurcation, their respective position as x and y coordinates and their direction as a 32-bit direction angle. As these features are not rotation- and translation-invariant, the template not only stores minutiae, but also computes and stores edges, which are connections between two minutiae. For each minutia, the algorithm calculates a fixed number of edges, which are described by their length and two angles relative to the reference minutia and the neighboring minutia respectively. These parameters do not change when the edge is moved or rotated, making the algorithm rotation- and translation-invariant.

During the second part – fingerprint matching – the algorithm evaluates whether two fingerprint templates, which are called probe and candidate, belong to the same finger. To do this, the algorithm first generates a special edge hash table for the probe in a preprocessing step to speed-up later search for similar candidate edges.

The candidate's part of the algorithm comprises three phases: (i) enumerate, (ii) crawl and (iii) score.

In detail, during (i) the algorithm's goal is to find root pairs available in both fingerprints. For this, it iterates through the minutiae of the candidate and calculates an edge to every other minutia of the candidate and the respective value of the edge hash. This hash value is looked up in the probe's hash table. If an edge with the same hash value exists, the corresponding probe edge is further compared in detail to the calculated candidate edge. When the edges are similar enough, a root minutia was found and a root pair consisting of the probe's edge and the candidate' edge is created.

```
Data: SourceAFIS Fingerprint Matching
Result: Probability Score of 2 fingerprints
Load Fingerprint Templates;
Compute root pairs in EdgeComparison;
for each root pair do
    if root != 0 then
        Generate Spanning Trees with crawl;
        Compute Probability with score;
    end
end
Return highest probability score;
```

Algorithm 1: SourceAFIS algorithm (Fingerprint Matching)

Starting from the first root pair, in (ii) the algorithm then tries to build a spanning tree through all minutiae. For this, the algorithm compares the surrounding edges of a root pair in probe and candidate. If one edge is similar enough, the ending minutiae get added to the tree on both fingerprints and their surrounding edges will once again be compared to each other. This procedure is repeated until all minutiae are checked. An example of how a spanning tree of matching fingerprints could look like is shown in Figure 1. In contrast to the previous step, where edges were compared to each other, this comparison takes the edge surroundings and placement into account, as edges can look very similar but be at different locations on the fingerprint.

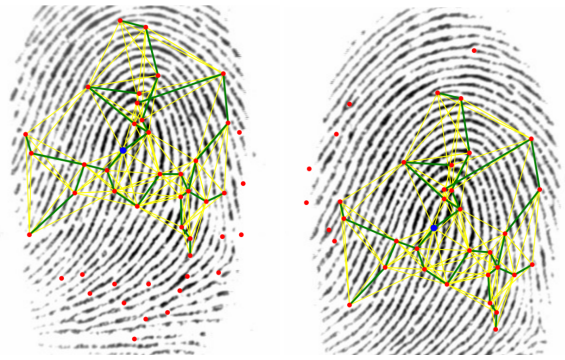

Figure 1: Spanning Tree of Matching Fingerprint Pair with minutiae marked in red, root pair blue, pairing tree green and graph of supporting edges yellow [19].

After generating a spanning tree, the algorithm computes a match probability score in (iii). To achieve this, the algorithm counts matched features, scores their similarity, and sums up the scores. The result is a representation of the matching probability, indicating whether the resulting tree is a match or just a coincidence.

The generation of the tree and the scoring are repeated for every root pair to ensure the best possible representation of the fingerprint. The best tree and probability score is returned as the similarity, which can then be compared to a threshold to declare the fingerprints matching or not.

2.2 Privacy Preserving SourceAFIS

As the feature extraction can be computed locally and without MPC, the minutiae extraction step can be incorporated unchanged from SourceAFIS, or replaced by other frameworks. To demonstrate this, we test the openly available FingerNet [18] and NBIS [20] minutiae extractors, as well as the commercially available Idkit from Innovatrix [8]. The resulting minutiae are fed into SourceAFIS to generate the fingerprint templates, consisting of the fingerprint's minutiae and edges needed for the matching process. These templates can subsequently be fed into the MPC implementation for further fingerprint matching. The results of the minutiae extractor comparison can be found in section 3 and show that the matcher also works with novel emerging machine learning based template extractors.

Regarding the encryption with MPC, the decision was made that all minutiae-describing details, including the coordinates, orientation, edge lengths and angles, have to be encrypted at all times. However, we chose that the ids of both edges and minutiae can be handled unencrypted, because they do not contain sensitive information and can be randomized between runs. By encrypting the information contained in the edges and minutiae while handling their ids unencrypted, a good balance between privacy and computational efficiency is achieved. This approach ensures that the most critical and private fingerprint data is protected while allowing some freedom for an efficient MPC implementation.

Moreover, it is stated in the SourceAFIS documentation [19] that the minutiae numbers are permuted pseudo-randomly during the feature extraction. Although the ordering is consistent, which ensures that running the algorithm with same image twice results in the same fingerprint template, thus guaranteeing no cryptographic security, the permutation adds an extra layer of protection, making it challenging for potential attackers to find sensitive data from the numbering of edges and minutiae alone. Nevertheless, we assume an additional random permutation step for the candidate templates stored in the database before entering the matching algorithm. This measure aims to prevent any linkage of templates between different runs.

Informally, the security guarantees we want to achieve are that non-matching templates are protected from re-identification and cannot be linked between different matching attempts. The current empirical results indicate that this properties can be achieved with good quality, however, a more formal analysis is needed and planned for a next step. In general, we would have preferred to start with a top-down approach, i.e., by defining a formal security model and prove its properties via the MPC protocol. However, it turned out that optimizing the different steps of the algorithm requires trade-offs to be made, which then make direct derivation of security properties from MPC protocols impossible but demand an additional security analysis to cope with the leakage.

3 Fingerprint Minutiae Extraction

Before discussing the privacy-preserving matcher implementation in detail, we also look at the impact of the pre-processing phase, i.e, template generation, to get the full picture on the processing pipeline. This section presents an exploration of four distinct minutiae extraction methods utilized in our study to evaluate their influence on the performance of the SourceAFIS MPC matcher. The selected extractors are as follows:

(1) SourceAFIS [19]
(2) Mindtct - A component of the NBIS suite [20] developed in 2007, known for its rapid processing.
(3) FingerNet - A hybrid approach combining deep convolutional networks with conventional techniques, particularly effective for latent fingerprints [18].
(4) Idkit - A commercially available minutiae extractor developed by Innovatrics, recognized for its high performance and reliability in various biometric applications [8].

The objective of this investigation is to analyze how the choice of minutiae extractor impacts the performance of the MPC Matcher, with a specific focus on two pivotal metrics: the number of extracted minutiae and the resulting Equal Error Rate (EER). The EER is computed for each dataset individually by cross-matching each recording with all other recordings in the dataset, excluding previously matched pairs. Subsequently, the mean and standard deviation of the EER across different datasets are considered. For the FVC datasets, comprising 80 entries each, this process yields 3160 matches per dataset.

Extractor	EER	Num. Minutiae
SourceAFIS	.12 ± 0.05	35.96 ± 20.2
Mindtct	.14 ± 0.14	72.34 ± 50.34
FingerNet	.03 ± 0.02	36.21 ± 9.75
Idkit	.11 ± 0.09	36.91 ± 11.81

Table 1: Comparison of minutiae extractors in terms of Equal Error Rate (EER) and number of extracted minutiae per fingerprint.

Table 1 compares the performance of the minutiae extractors in terms of EER and the number of minutiae extracted per fingerprint. It indicates that SourceAFIS, FingerNet and Idkit extract a comparable number of minutiae, while Mindtct extracts significantly more minutiae. Moreover, the performance measured in EER of the SourceAFIS matcher based on the minutiae extracted by FingerNet significantly outperformed the others. In contrast, Mindtct exhibits the lowest performance, suggesting a higher incidence of false-positive minutiae. However, both Mindtct and Idkit provide a quality score alongside the extracted minutiae, which can be utilized for minutiae filtering. To facilitate comparison with Mindtct, we normalized the quality score returned by Idkit to the range [0, 100]. Note however that the Idkit matcher does not only rely on minutiae information, resulting in an EER of 0.17% ± 0.16% when using it not only as minutiae extractor, but end-to-end matcher.

Figure 2 illustrates the distribution of minutiae quality scores across different datasets. The variability in scores returned by

Mindtct is more pronounced compared to those returned by Idkit, resulting in a wider distribution.

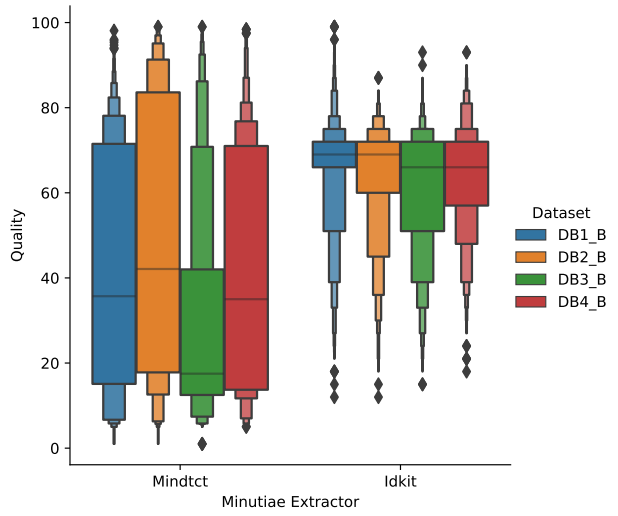

Figure 2: Minutiae quality estimated by Idkit and Mindtct normalized to the range [0, 100]

Moreover, Figure 3 depicts the EER and average minutiae count after applying a minutiae quality score threshold, removing minutiae below the given threshold. The figure highlights the impact of excluding low-quality minutiae on EER for both Mindtct and Idkit. Specifically, for Mindtct, as the number of low-quality minutiae is reduced, the EER improves until it reaches approximately the number of minutiae extracted by Idkit, after which the EER worsens again. Conversely, for Idkit, the EER decreases with the exclusion of minutiae, although the effect is relatively weak when only minutiae with a quality score of less than 50 are removed.

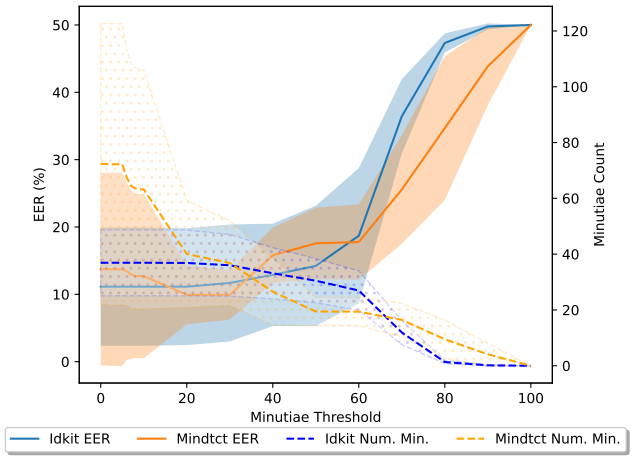

Figure 3: EER (solid line) and average minutiae count (dashed line) after applying a minutiae threshold specified on the x axis for Idkit (blue) and Mindtct (orange).

Based on these results we have seen that modern algorithms for minutiae extraction like FingerNet outperform the rather old SoureAFIS approach, which is from 2002. However, we also show that the matching stage in SourceAFIS is fully compatible with these approaches and can benefit from the improvements in feature extraction. Thus, our privacy preserving matcher presented in the following sections provides top notch accuracy if combined with FingerNet, comparable to commercial solutions. Moreover, if the feature extraction also delivers quality scores for minutiae, this could be used to dismiss unreliable minutiae and therefore reduce the template size and speed-up MPC-based matching.

4 Optimizations for MPC

A naive MPC implementation of the matching stage of SourceAFIS would simply translate the unchanged algorithm into the secure domain. However, the existing implementation heavily relies on MPC-unfriendly operations, data structures and programming patterns. For performance reasons, it uses hashes and recomputes comparisons rather than storing them, both of which are expensive in MPC. Consequently, our implementation strategy involves reusing results and leveraging pre-computed values for operations which are costly in MPC.

4.1 Matching Algorithm with MPC

In the following we describe our results on the presented main steps of SourceAFIS, which corresponding functions are called (i) enumerate, (ii) crawl and (iii) score and can be implemented with MPC independently of each other.

(i) Enumeration Step. To avoid having to compare the n edges of the probe to the m edges of the candidate, the original implementation compares edges by testing if their hashes fall into the buckets. As edges compare equal when their lengths and angles differ by not more than a certain number of pixels or degrees respectively, this method necessarily over-approximates the result and so has to later on be complemented by exact comparison. However, each candidate edge then only has to be compared to probe edges from the same buckets, resulting in a considerable speed-up of the matching process. In MPC, this is not a profitable, or even possible, optimization. No matter the method, reducing the number of candidate edges to be compared would reveal information about those edges, so an MPC implementation must always compare each candidate edge to each probe edge. In addition to this, the original implementation does the comparison in order, in a first run only considering edges whose length is above a threshold, and aborts the enumeration after a certain amount of lookups has been performed. Again, this is not (entirely) feasible in MPC.

These considerations taken into account, we decided to utilize this necessary $n : m$ comparison create a look-up table that indicates which edges match. This look-up table can then be used for both the localization of the roots and the formation of the tree.

Specifically, for the enumerate function, this means that the re-computation of edges, the calculation of the hash value, the search of edges with the same hash value and the comparison of edges, when finding matching edges in the probe's hash, can be replaced by the direct comparison of all edges, resulting in the output of a match array. To compare these edges, the algorithm

simply checks whether the difference between the lengths and angles of the probe and candidate edge fall within an acceptable error tolerance. Conceptually, the resulting algorithm can therefore be written as laid out in Algorithm 2.

```
Data: Two lists of lists of edges probe and candidate -
      edges having a length l, a reference angle r and a
      neighbour angle n
A maximum distance error MDE
A maximum angle error MAE and its complement CMAE
Result: A list of lists of binary values
res = []
for ps in probe:
 for p in ps:
  vec = []
  for cs in candidate:
   for c in cs:
    ml = abs(p.l - c.l) ≤ MDE
    a1d = abs(p.r - c.r)
    m1a = a1d ≤ MAE or a1d ≥ CMAE
    a2d = abs(p.n - c.n)
    m2a = a2d ≤ MAE or a2s ≥ CMAE
    vec.append(ml and m1a and ma2)
  res.append(vec)
return res
```

Algorithm 2: MPC-Friendly Enumeration Step (Plain Text)

The resulting match array can then be decrypted and is subsequently available in unencrypted form for further processing and as a lookup table for the fingerprint template's roots. This is permissible because the values in the array only represent the edge numbers, which are permuted, thus making the association of information challenging for a potential attacker.

(ii) Crawl Step. Continuing the encryption approach of the enumerate function, the aim of the implementation is to utilize the resulting array of root pairs from the enumerate function, which represents matching edges, to construct the best possible spanning tree. Considering the previous decision to use edge and minutia numbers unencrypted, along with the utilization of the root array, this enables an unencrypted computation of the crawl function, as described in Algorithm 3.

Since the function can be computed without encryption, the new implementation closely resembles the original implementation. The primary difference lies in the collection and comparison of the surrounding edges. Instead of re-matching the collected probe and candidate edges to each other, which would require encryption and increase the execution time, the new implementation utilizes the root array as a look-up table to determine if the edges match and subsequently can be added to the spanning tree.

Another difference to the original SourceAFIS lies in the sorting of the edges within the queue. In the original implementation, edges are sorted in ascending order by length upon addition to the queue, ensuring the next shortest edge is always at the front. However, this step was omitted in the MPC implementation because, depending

```
Data: One root pair found in enumerate passed as
      parameter, which will be the starting point for the
      spanning tree
Result: Subsequently iterates through all root pairs and
        returns the found spanning tree.
Append passed root pair to queue;
while queue not empty do
    Use first root pair of queue as reference;
    if reference not yet connected to spanning tree then
        Add reference to spanning tree;
        Collect edges originating reference and add them to
         the queue if they are true in the root array;
    else
        Add reference as support to tree;
    end
end
```

Algorithm 3: Crawl Step: Generate Spanning Trees

on the framework, it is either not feasible to encrypt this with MPC or associated with very long execution times.

However, this absence of re-sorting results in the generation of different spanning trees compared to the original, consequently leading to the computation of a different score. While according to our tests this modification does not lead to a change in the average error rates, our results, however, differ significantly from the original in individual cases, where matches found by the original are not found or vice versa.

This is illustrated in Figure 4, where each row and column represent one fingerprint image and their overlap the match of the respective fingerprints. For comparisons highlighted in white, our implementation obtains the same result as the original (evaluated at the respective EER as threshold), while for green ones, we make correct matching decisions that the original does not make, and vice versa for red ones.

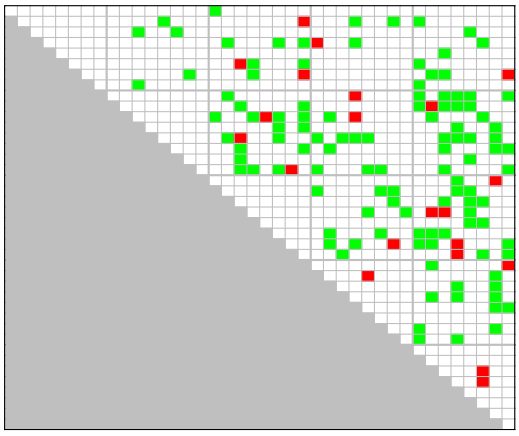

Figure 4: Comparison Original vs MPC Implementation on fingerprints 6 to 10 of DB3, indicating equal results in white, better in green and worse matches in red (evaluated at the respective EER as the threshold).

(iii) Scoring Step. As previously mentioned, the scoring function computes the match probability score of the probe and candidate fingerprint image. Furthermore, certain operations that do not involve sensitive data, but are solely based on the number of minutiae in a tree, the number of supporting edges or found root pairs, can be computed without encryption. Consequently, only a few comparison operations need to be encrypted. These parts include the minutia type, as well as the edge length and angles. All parts except the calculation of the accuracy of the paired edges in a tree can be reused from the original implementation. Instead of recalculating the edges, this implementation reuses the locally determined edges, which reduces the execution time for the MPC implementation.

4.2 Validation of Approach

To analyze the impact of the different optimization steps on the accuracy of the overall algorithm we evaluated and compared it to the original version as follows: The MPC-optimizations proposed in Section 4.1 were first implemented and tested in a plaintext version. We call this step 'Caching', as this is the most prominent change from the original SourceAFIS matcher. In a second step, we changed the floating-point arithmetic to a fixed point arithmetic, as true floating-point arithmetic is expensive in MPC.

The fingerprint databases used for the evaluation were originally created for the FVC2000: Fingerprint Verification Competition [12] and can be accessed online. There are four databases consisting of 80 fingerprint images, each database collected by different means. The databases 1 and 2 were collected by one optical and one capacitive sensor; both were small-size and low-cost sensors, and no attention was paid to correct finger centering and the sensor platens were not systematically cleaned. Database 3 was collected by a higher quality optical sensor with the sensor platens getting cleaned between images, and database 4 consists of synthetically generated fingerprint images. Each of the databases includes 80 images, consisting of 10 fingers with eight images each.

As indicated in Table 2, despite our modifications, we still achieve similarly good results, except for DB3, where we obtain even significantly better results. This disparity can be attributed to the fact that Database 3 contains fingerprint images of very high quality, resulting in an exceptionally large number of minutiae and consequently edges being detected, also indicated by the number of found root pairs in Table 10. As pointed out above, the original matcher only compares a fixed number of edges per match before aborting the process, thus excluding a considerable amount of information in the case of Database 3. In our implementation, however, all available edges must always be compared, leading to longer execution times but enabling better results in this particular case.

	DB1†	DB2	DB3	DB4
Original	.04	.12	.18	.07
Caching	.04	.11	.09	.07
FixPoint	.04	.11	.09	.07

Table 2: Comparison of Equal Error Rate (EER) of different implementation steps with 4 fractional digits for FixPoint.

	DB1†	DB2	DB3	DB4
Original	11.8	3.7	2.8	10.5
Caching	12.2	6.0	10.4	11.0
FixPoint	12.2	6.0	10.4	11.0

Table 3: Comparison of Threshold at Equal Error Rate (EER) of different implementation steps with 4 decimal points for FixPoint.

The † above DB1 in Table 2 and others signifies that this database originally contained fingerprints from which SourceAFIS was not able to extract any minutiae. As fingerprints without any minutiae will never match any other fingerprints (nor even themselves), we removed them from the dataset.

The fact that the EERs did not change significantly for the different implementation steps (apart from DB3) shows that the privacy preserving variant performs as good as the original plaintext version, as also presented in Figure 5. In fact, due to the missing abort mechanism in oblivous edge comparison we even achieve slightly better results compared to the original version.

It is also worth noting that transitioning from floating-point arithmetic to fixed-point arithmetic does not yield significant changes. As indicated by Table 4, the results with only one decimal place remain as good as those with floats and only marginally change with increasing precision. This is attributed to the fact that the encrypted implementation relies solely on comparisons with constants which already have good accuracy without using fixpoint scaling, e.g. by using degrees for angle comparisons, and are thus unaffected by the additional precision of switching to fixpoint representation.

fractional digits	EER	FM	FNM
∞	.04	82	9
4	.04	81	9
2	.04	92	10
1	.04	83	9

Table 4: Comparison of EER, Number of False Matches (FM) and False Not-Matches (FNM) for the FixPoint Implementation with different numbers of fractional digits in DB1†.

4.3 Minutiae Extractors and MPC

As previously mentioned, it is feasible to separate feature extraction and fingerprint matching. Features such as minutiae and edges are extracted locally and without multiparty computation, serving as the foundation for matching independently of this process. Thus, the insights from Section 3 can be directly applied to the MPC implementation, allowing us to use a different feature extractor instead of the original SourceAFIS feature extractor and reduce the error rates of our implementation. Consequently, we can present an optimized MPC implementation with an average EER of 0.055, as shown in Table 5.

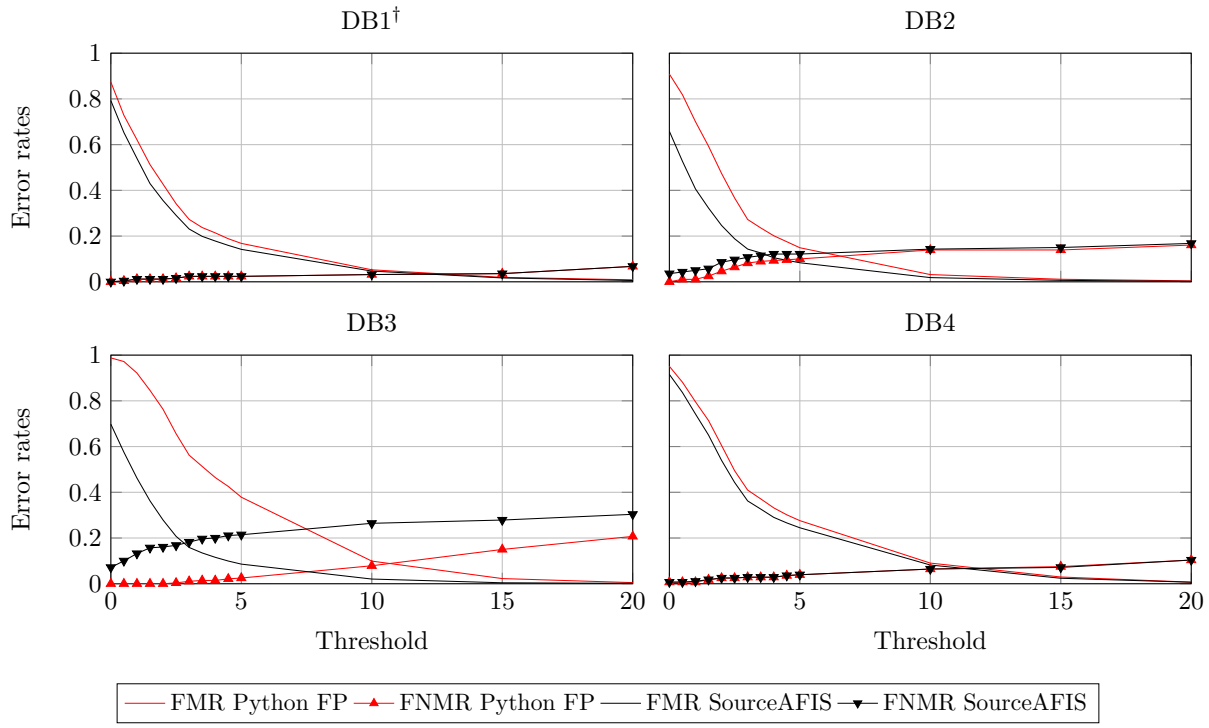

Figure 5: Error Rates Original SourceAFIS vs FixedPoint Implementation for all DB separately.

	DB1	DB2	DB3	DB4
SourceAFIS	.04†	.11	.09	.07
Idkit	.05	.07	.10	.23
FingerNet	.01	.05	.03	.03

Table 5: Comparison of Equal Error Rate (EER) of different minutiae extractors combined with MPC implementation.

5 Performance Evaluation with MP-SPDZ

Our first test implementation with MPC presented in [11] was developed using an open-source package called MPyC: Python Package for Secure Multiparty Computation [14]. For this we only implemented the (i) enumerate step as described in 4.1 as a benchmark. While there are no apparent weaknesses in the functionality of the MPyC implementation, its primary disadvantage is the execution time, which takes approximately 7 minutes for an average sized fingerprint template. Despite achieving a significant improvement compared to our initial MPyC implementation with an execution time of around 3 hours, the resulting time still exceeds the requirements for the intended use-case. Thus, the framework was concluded to decrease the algorithm's speed too much for its intended use case.

However, this work also includes an estimation of the execution time for a full implementation with MP-SPDZ. This estimation is based on benchmarks of the framework found in [10], resulting in an achievable execution time of approximately 7 seconds. This execution time is well suited for the intended purposes, which is why we decided to conduct a full implementation using MP-SPDZ instead of MPyC.

	Min.	Max.	Avg.
Minutiae	5	100	39

Table 6: Number of Minutiae per Fingerprint

	Matches			Non-Matches		
	Min.	Max.	Avg.	Min.	Max.	Avg.
DB1	5	45	7.4	1	7	2.3
DB2	4	77	6.3	2	7	2.8
DB3	5	100	5.5	2	9	2.9
DB4	4	33	6.2	1	7	3.0
Total	4	100	6.4	1	9	2.8

Table 7: Tree Size per Fingerprint

5.1 MP-SPDZ Implementation

The biggest difference between MPyC and MP-SPDZ is that while the former is a Python library that can be directly integrated into a normal Python program, using the latter results in a separately running compiled program that takes its input in the form of a "tape" whose length has to be known when compiling the program.

```
p_minutiae = public_input()
c_minutiae = public_input()

p = sint.Tensor([100, 9, 3])
c = sint.Tensor([100, 9, 3])
end0 = p.read_from_file(0)
c.read_from_file(end0)

@for_range_opt(p_minutiae)
def _(pm):
 @for_range_opt(c_minutiae)
 def _(cm):
  @for_range_opt(9)
  def _(pe):
   @for_range_opt(9)
   def _(ce):
    ld = abs(p[pm][pe][0] - c[cm][ce][0])
    a1d = abs(p[pm][pe][1] - c[cm][ce][1])
    a2d = abs(p[pm][pe][2] - c[cm][ce][2])
    ml = ld <= MDE
    ma1 = (a1d <= MAE) + (a1d >= CMAE)
    ma2 = (a2d <= MAE) + (a2d >= CMAE)
    print_ln('%s', (ml * ma1 * ma2).reveal())
```

Listing 1: Enumerate (MP-SPDZ)

This means that when rewriting a program from using MPyC to using MP-SPDZ, it has to be re-organized around getting data into and out of the MPC part, and all Python data structures have to be expressed in terms of fixed-length arrays and matrices, which is a challenge when working with trees or other inputs of varying size or structure.

In our case, for the enumeration step we assume, based on the characteristics of the databases we use, presented in Table 6, that the number of minutiae per fingerprint is capped at 100, and that each of them has exactly 9 edges. At runtime, the actual number of minutiae in the probe and candidate fingerprints are passed as plaintext values and used as loop bounds. For the scoring step, we also assume that trees have not more than 100 nodes, referencing Table 7, and pass the real size at runtime to use as loop bounds as well.

A big point in favor of MP-SPDZ is that it implements eda-bits[5] which can significantly speed up non-linear operations such as comparisons. It can be turned on by either add `-edabit` to the compiler invocation or adding the line `program.use_edabit(True)` to the program. In our tests, turning this feature on for the enumeration step reduced both running time and the amount of data transferred to a tenth of the original values.

The MP-SPDZ version of this step is also somewhat different from the MPyC version because we found that the compiler is able to optimize the two (fixed-length) inner loops, and therefore we can directly compute the result for every element and print it. A slightly reduced version of the resulting program is in Listing 1.

The MP-SPDZ version of the scoring step only computes and then reveals the elements of the score that are based on private data: the minutia type hits, the distance error sum and angle error sum. To cut down on unnecessary overhead for restarting MP-SPDZ (establishing connections, reading in data, ...) for each tree, our implementation computes these score elements for all trees. It therefore first reads in the number of trees as a public value (`count`), then for each tree its actual length. Fixed parameters are the inner distance radius (`IDR`) and the inner angle radius (`IAR`). The code for this is in Listing 2. The rest of the score is based on plain-text data and is computed in the clear.

```
count = public_input()

@for_range(count)
def _(_):
  actual_length = public_input()
  th = sint(0)
  des = sint(0)
  aes = sint(0)
  @for_range_opt(actual_length)
  def _(i):
        i1 = public_input()
        i2 = public_input()
    hit = p_m[i1].bit_xor(c_m[i2]).bit_not()
    th.iadd(hit.if_else(sint(1), sint(0)))
    pe = public_input()
    ce = public_input()
    d_err = abs(p_e[pe][0] - c_e[ce][0])
    des.iadd(d_err.max(IDR))
    ref_d = abs(p_e[pe][1] - c_e[ce][1])
    nei_d = abs(p_e[pe][2] - c_e[ce][2])
    ref_d = (ref_d < PI).if_else(ref_d, TAU - ref_d)
    nei_d = (nei_d < PI).if_else(nei_d, TAU - nei_d)
    aes.iadd(ref_d.max(IAR))
    aes.iadd(nei_d.max(IAR))
  th_p = th.reveal()
  des_p = des.reveal()
  aes_p = aes.reveal()
  print_ln('%s␣%s␣%s', th_p, des_p, aes_p)
```

Listing 2: Score (MP-SPDZ)

Size	*rep-field* (s)	*shamir* (s)
1	0.01	0.02
2	0.02	0.02
5	0.02	0.04
10	0.03	0.06
20	0.05	0.10
50	0.03	0.08
100	0.04	0.14

Table 9: Time for scoring trees of size *n* on MP-SPDZ with different protocols

5.2 Benchmarking

The MP-SPDZ implementations were benchmarked using a 16-core Intel(R) Xeon(R) W-2245 CPU running at 3.90GHz under Ubuntu 22.04.4 LTS. Execution times are given for the Replicated Field and Shamir protocols with 3 parties and no network latency.

Minutiae	*rep-field* (s)	*shamir* (s)
10	1.04	3.32
20	3.25	11.78
30	7.03	26.17
40	12.25	46.57
50	18.94	72.09

Table 8: Time for finding roots (candidate and probe both with *n* minutiae) on MP-SPDZ with different protocols

For the scoring algorithm, MP-SPDZ reported unused eda-bits for the scoring algorithm and recommended changing the batch size. We chose the tree size multiplied by ten and achieved the results detailed in 9. Without this change, the times measured were much higher and remained almost equal across different tree sizes. (The anomaly in our present results at tree size 20 is caused by this specific choice of batch size and is robust).

Further speed-ups should be achievable by using more of the optimization options offered by the MP-SPDZ compiler, for example running loops in parallel. The optimizations performed by the compiler, however, are rather fragile and require careful structuring of the program - for example, an optimized `for_range_opt` loop cannot contain any control structures other than further `for_range_opt` loops. Moreover, there are quite a few different loop constructs with different performance characteristics (next to `for_range_opt`, there is the basic `for_range` loop, but also `for_range_multithread`, `for_range_opt_multithread`, and `for_range_parallel`), some of them with further tuneable parameters. Achieving optimal performance, therefore, requires thorough and exhaustive benchmarking of all available options, which we leave for future work, together with a more realistic setting that runs the MPC nodes on different servers, or simulates such a setup.

The average values listed in the tables above are utilized as follows: Considering a total of 39 minutiae, the current execution time for traversing the enumerate function is 12.25 seconds. Furthermore, assuming finding 250 root pairs, where we have to do a crawl and scoring iteration for each, we obtain a runtime of $250 \cdot 0.02 = 5$ seconds for an average treesize of 6, resulting in a total runtime of 17.25 seconds if we do the scoring sequentially.

However, the constraining factor in this scenario is the enumerate function, as we can execute the scoring iterations independently of each other. Hence, we anticipate that in the future, we can parallelize these iterations, resulting in a significantly improved execution time for the scoring part in the sub-second range for realistic scenarios.

5.3 Further optimizations

Considering Table 7, it is evident that there are no matches with a tree size < 4 and no non-matches with a tree size ≥ 10. Therefore, for these cases, there would be no need to calculate a score, instead, a decision could be made directly. We have already implemented this on a smaller scale by filtering out all trees with a size < 2, thus saving 20% of the scoring calls, which would otherwise happen once per found root pair, shown in Table 10. The aforementioned option would further expand these savings, especially when considering the distribution of tree sizes in two randomly selected comparisons (one match and one non-match) shown in Table 11. The number of trees to score could also be further reduced by filtering out all permutations or subtrees.

	Matches			Non-Matches		
	Min.	Max.	Avg.	Min.	Max.	Avg.
DB1	8	899	194	0	428	65
DB2	29	1887	367	2	803	153
DB3	112	4007	1176	25	2484	590
DB4	16	619	178	3	321	58
Total	8	4007	479	0	2484	217

Table 10: Root Pairs per Fingerprint

Tree Size	0 - 3	4 - 6	7 - 9	≥10
Nodes (M)	270	63	0	136
Nodes (NM)	30	18	0	0

Table 11: Distribution of Tree Size during comparison of fingerprints 102-5 and 102-1 (Match), or 102-5 and 105-4 (No-Match) in DB1

However, the greatest potential for time improvement would come from the preselection of minutiae, as one encounters the toughest limitations during the enumerate step. For this, minutia quality scores as used by the Idkit and Mintct Matcher, presented in 3, could be used to exclude minutiae of bad quality.

Furthermore, our recent analysis revealed that most edges exist twice in candidate and prove edge list inputs of enumerate with swapped incoming and outgoing minutiae and respective angles. This is because, for each minutia, edges are computed with the nine closest minutiae, and it is highly likely that the outgoing minutia is also among the nine closest for these minutiae. Thus, when comparing a probe and candidate edge that have already been compared with each other in a swapped version, the costly comparison does

not need to be repeated, and the result can be reused for the second comparison in the root array. Taking these new insights into account, it is expected that the number of oblivious comparisons in the enumerate function can be reduced by approximately a half, based on a combinatorial argument. However, detailed analysis of this optimization is left for future work.

Still another avenue is doing away with the necessity of directly comparing all candidate edges to all probe edges. Algorithmically, computing the lookup table in the enumerate step is an instance of the fixed-radius near neighbor problem, which we solve via the (quadratic) brute force approach, and which is efficiently solved by hashing in the original SourceAFIS implementation. It is worth investigating whether there are in fact MPC-friendly algorithms among the efficient solutions to this problem. For example, could we construct a locality-sensitive hashing function that preserves input privacy but retains enough precision to keep the EER at about the current level?

6 Conclusions

In conclusion, our work introduced a privacy-preserving implementation of SourceAFIS with promising prospects for real-time applications utilizing the MP-SPDZ framework. Moreover, we demonstrated the versatility of the MPC implementation, as it seamlessly integrates with various minutiae extractors, resulting in significant improvements in error rates. These findings underscore the feasibility of employing MPC techniques for secure and efficient fingerprint matching in sensitive domains.

However, our results only lay the basis and more effort and multiple steps are needed to achieve the aspired goal of real-time minutiae matching. Firstly, further potential optimization on MPC algorithm and protocol level for *enumerate* and *score* as discussed in Section 5.3 should be investigated. Secondly, it will also be important to formally model and proof security by analyzing and quantifying the leakage trade-offs in detail, i.e., by leveraging differential privacy methods. Thirdly, it could be interesting to study how leakage can be further reduced by removal or replacement of the *score* step or even making *crawl* more oblivious.

Acknowledgments

The work was partially funded by the Austrian Security Research Programme KIRAS of the Federal Ministry of Finance via grant agreement no. 905287 ("PASSENGER") and from the European Union's Horizon Europe Programme via grant agreement no. 101-073821 ("SUNRISE") and no. 101114675 ("HARMONIC"). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

References

[1] Sahar Adnan, Fatima Ali, and Ashwan A Abdulmunem. 2020. Facial feature extraction for face recognition. In *Journal of Physics: Conference Series*. Vol. 1664. IOP Publishing, 012050.

[2] Mauro Barni, Tiziano Bianchi, Dario Catalano, et al. 2010. Privacy-preserving fingercode authentication. In *Proceedings of the 12th ACM workshop on Multimedia and security*, 231–240.

[3] Marina Blanton and Paolo Gasti. 2011. Secure and efficient protocols for iris and fingerprint identification. In *ESORICS 2011*. Springer, 190–209.

[4] Hendrik Eerikson. 2020. *Privacy Preserving Fingerprint Identification*. University of Tartu. Bachelor's Thesis.

[5] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. 2020. Improved primitives for mpc over mixed arithmetic-binary circuits. Cryptology ePrint Archive, Paper 2020/338. https://eprint.iacr.org/2020/338. (2020). https://eprint.iacr.org/2020/338.

[6] David Evans, Vladimir Kolesnikov, and Mike Rosulek. 2018. A pragmatic introduction to secure multi-party computation. *Foundations and Trends® in Privacy and Security*, 2, 2-3, 70–246. DOI: 10.1561/3300000019.

[7] Diana-Elena Fălămaş, Kinga Marton, and Alin Suciu. 2021. Assessment of two privacy preserving authentication methods using secure multiparty computation based on secret sharing. *Symmetry*, 13, 5, 894.

[8] Innovatrix. 2019. IDKit SDK https://support.innovatrics.com/support/solutions/folders/5000267908, accessed on 06 03 2023. en. (2019). Retrieved Mar. 6, 2023 from https://support.innovatrics.com/support/solutions/folders/5000267908.

[9] Marcel Keller. 2020. Mp-spdz: a versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, 1575–1590.

[10] Thomas Lorünser and Florian Wohner. 2020. Performance comparison of two generic mpc-frameworks with symmetric ciphers. In *ICETE (2)*, 587–594.

[11] Julia Mader and Thomas Lorünser. 2024. Feasibility of privacy preserving minutiae-based fingerprint matching. In *ICISSP 2024: 10th International Conference on Information Systems Security and Privacy*, 899–906.

[12] Dario Maio, Davide Maltoni, Raffaele Cappelli, et al. 2002. Fvc2000: fingerprint verification competition. *IEEE transactions on pattern analysis and machine intelligence*, 24, 3, 402–412.

[13] Tim Rohwedder, Dailé Osorio Roig, Christian Rathgeb, and Christoph Busch. 2023. Benchmarking fixed-length fingerprint representations across different embedding sizes and sensor types. *2023 International Conference of the Biometrics Special Interest Group (BIOSIG)*, 1–6. https://api.semanticscholar.org/CorpusID:259937809.

[14] Berry Schoenmakers. 2018. Mpyc secure multiparty computation in python. Last accessed 01-07-2023. (2018). https://lschoe.github.io/mpyc/.

[15] Siamak F Shahandashti, Reihaneh Safavi-Naini, and Philip Ogunbona. 2012. Private fingerprint matching. In *Australasian Conference on Information Security and Privacy*. Springer, 426–433.

[16] Koen Simoens, Pim Tuyls, and Bart Preneel. 2009. Privacy weaknesses in biometric sketches. In *2009 30th IEEE Symposium on Security and Privacy*, 188–203. DOI: 10.1109/SP.2009.24.

[17] Bernhard Strobl and Margherita Natali. 2022. Enhancing biometric data security by design. *ERCIM NEWS*, 131, 25–26.

[18] Yao Tang, Fei Gao, Jufu Feng, and Yuhang Liu. 2017. FingerNet: An unified deep network for fingerprint minutiae extraction. en. In *2017 IEEE International Joint Conference on Biometrics (IJCB)*. IEEE, Denver, CO, (Oct. 2017), 108–116. ISBN: 978-1-5386-1124-1. DOI: 10.1109/BTAS.2017.8272688.

[19] Robert Važan. 2004. Sourceafis fingerprint matcher. Last accessed 09-11-2023. (2004). https://sourceafis.machinezoo.com/.

[20] Craig I Watson, Michael D Garris, Elham Tabassi, Charles L Wilson, R Michael McCabe, Stanley Janet, and Kenneth Ko. 2007. User's guide to NIST biometric image software (NBIS). en. Tech. rep. NIST IR 7392. Edition: 0. National Institute of Standards and Technology, Gaithersburg, MD, NIST IR 7392. DOI: 10.6028/NIST.IR.7392.