JOHANNES KEPLER
UNIVERSITY LINZ

Gerald Schoiber
Institute of
Networks and Security

@ gerald.schoiber@ins.jku.at
https://www.digidow.eu/

October 2021

Privacy Preserving Hash for Biometrics

Technical Report

Christian Doppler Laboratory for
Private Digital Authentication in the Physical World

This work has been carried out within the scope of Digidow, the Christian Doppler Laboratory for Private Digital Authentication in the Physical World, funded by the Christian Doppler Forschungsgesellschaft, 3 Banken IT GmbH, Kepler Universitätsklinikum GmbH, NXP Semiconductors Austria GmbH, and Österreichische Staatsdruckerei GmbH.

JOHANNES KEPLER
UNIVERSITY LINZ
Altenberger Straße 69
4040 Linz, Austria
jku.at

Contents

Abstract

A service that has to interact with multiple potential biometric sensors, needs to share information about an individual with them. Although it is possible that there will no interaction with such an sensor, the data is shared nevertheless. Every shared information about biometric data of an individual could lead to a potential leakage of sensitive data. To prevent this we introduce fuzzy hash which prevents this problem by generating a hash that cannot be tracked back to the original biometric data. Still, this hash can be compared against other embeddings which allows the sensor to interact with the correct service without an interactive protocol.

1. Introduction

Authorization is facing new challenges in a more ubiquitous computing world. Traditional passwords are getting replaced by biometric information which are often used to pre-register on sensors in order to save time on interactions. However, those biometric information should not be shared to any sensor until the individual it belongs to are interacting with them. Fuzzy hash provides a solution for this issue by preventing traceability of an individual by generating different hashes from a collection of biometric features. The generated one-time hash can still be compared against other embeddings without revealing information about the owner.

There are multiple different approaches that try to solve this issue as well. Most of them are using computational power to obtain this but this causes problems on scaling especially on low powered distributed sensors [3, 4, 8, 10]. Other approaches are using interactive protocols which do need to actively exchange information during the process of interaction with an individual [9, 6, 12, 11]. If the number of potential matches is increasing it will get harder to exchange data in time. Our approach is non interactive and can compare multiple hashes without much computational power.

The following sections will explain the details about the concept and the implementation as well as the corresponding tests to proof the functionality and resilience against attacks.

1.1 Problem

In order to interact with a face detection sensors in the Digidow[1] project, it is necessary to register to sensors before there is an interaction with an individual. The reason for this is because the sensor does not have any information about an individual except the information the individual provides beforehand. With the registration, the sensor get the information needed to find a match and knows who to contact if it is needed. However, thus would give each sensor the information about the biometric information even if there will be no physical interaction with them. If an attacker would be able to gain access to such sensors, all registered information could be linked to individuals.

[1] https://www.digidow.eu

1.2 Idea

To prevent leaking biometric data on a breached sensor, we introduce the concept of fuzzy hash. We use a modulo operation as a trap door function to generate a new hash from one or multiple embeddings. This prevents reversing the hash but still hold enough information to make comparisons possible. As the generated hash cannot be reversed, it makes it very hard for an attacker to make any assumptions to which person a hash belongs to. That is also the case if the same embedding is used multiple times as the hash is salted with random values each time it is generated. Therefore, the correct embedding needs to be known on the sensor side in order to compare it against the hash. All this does not require an interactive protocol nor does it take a lot of computational power to make comparisons, which does make scaling easier.

2. Fuzzy hash

Image recognition does not use an image directly for the comparison process. Instead, it will extract features which then can be compared against each other. This features are also called embeddings and they differ in size and value range depending on the used neuronal network which produce them. In our setup, a vector of 512 32-bit floating point numbers represents a single embedding. Although the concept of fuzzy hash is not restricted by the size of an embedding, it is conceivable that less information might perform worse. The sections bellow will describe the process of creation and comparison in more detail along with different possible attack scenarios and how they can be defeated.

2.1 Creating a fuzzy hash

The basic idea of the fuzzy hash relies on the modulo operation. By choosing a very small divisor, it will create another vector with the same amount of elements but way smaller values. This transforms values from the original value space E into the new divisor value space D. Keep in mind that the value space E might differ on other neuronal network models.

$$E := \{x \in \mathbb{R} | min(embedding) \leq x \leq max(embedding)\}$$
$$E := \{x \in \mathbb{R} | -0.3 \leq x \leq 0.3\}$$

Values from the divisor are defined by the *exponent* which will define the value space for D like:

$$divisor = random(1, 10) * 10^{-1 * exponent}$$
$$D := \{x \in \mathbb{R} | x < divisor\}$$

The default value for the exponent is 16 which was found out to work best on our embeddings. Higher exponents can be chosen but might end up performing very poorly later on in the comparison process. Whereas lower values will make the hash more prune to exhaustive brute force attacks as later discussed.

In order to create a new fuzzy hash we need input values. There are two different types of input variables; static and dynamic generated. Static input variables can be defined once and be used for each generation. Although called static,

they do not need to be strictly static on each generation. In fact, the validation does not change because all necessary parameters are included in the hash. This makes it possible to make adjustments in the generation process of a new hash if needed.

2.1.1 Static input

Static input values do not have to be change on each generation of a hash. Therefore it is save to define them once and reuse them.

- *Embeddings* In order to create a more stable hash it is recommended to use multiple embeddings from a single person.
- *Exponent* The exponent is needed to randomly creating a divisor for the modulo operation.
- *Gamma* To make the hashes more resilient against correlation attacks, we add noise to the hash. The gamma value defines the percentage used to vary each element in the hash.
- *Phi* For the comparison process we use the phi value to define a range in where two hashes are assumed to be equal. It is defined as a percentage which is added to the calculated maximal distance at the creation process.

2.1.2 Random input

As fuzzy hashes can be generated multiple times with the same embeddings as input, it is necessary to add random generated input values as well. This prevents correlation between hashes which is shown later in this report.

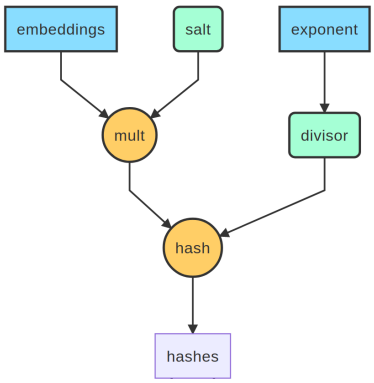
- *Salt* The salt is a randomly generated vector with the same amount of elements as the embedding. It gets element wise multiplied with all embeddings used in the process.
- *Divisor* The divisor is randomly generated with respect to the *exponent*.
- *Noise* A randomly generated vector with the same size of the embedding where the values are derived element wise from the hash and the *gamma*.

2.1.3 Modulo function

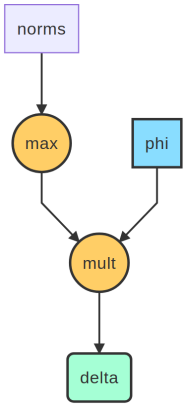
There are multiple modulo functions defined so we first need to define the modulo function we are using for our fuzzy hash:

$$hash(f) := \begin{cases} n_i - divisor * \lfloor \frac{n_i}{divisor} \rfloor, \text{if } n_i \geq 0 \\ n_i - divisor * \lceil \frac{n_i}{divisor} \rceil, \text{if } n_i < 0 \end{cases}$$

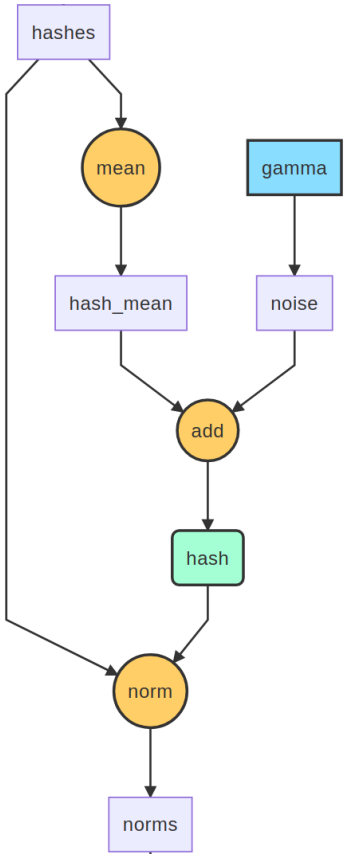
As the definition shows, we treat the positive and the negative values in a different way to preserve the value best possible when rounding the remainder to the direction of zero. This particular modulo operation is also known as the *euclidean division* defined by Raymond T. Boute [1].



(a) Create hashes from embeddings



(b) Create delta for hash



(c) Create norms from hashes

Figure 1: Create fuzzy hash

2.1.4 Generation

Starting by generating a new *salt* vector which is element-wise multiplied with each provided embedding. Although it is possible to use a single embedding to generate a hash, it is not recommended as a single embedding might be not as stable as using multiple different embeddings of the same person. After generating the *divisor*, all salted embeddings are getting hashed giving us multiple hashes derived from the provided embeddings as shown in figure 1a.

The next step is to calculate the hash vector and the norm values which are the distances between the mean hash and the individual hashes of each embedding as shown here 1c. By calculating the mean of the different hashes we are getting a more stable hash as outlier are getting smoothed out. Optionally, we can add noise to this hash defined by the *gamma* value. The reason why this is optional is because we are using a salt vector anyways which is far more effective to make it impossible to correlate hashes without the downside of making the comparison worse. After this step we are done by calculate the *hash* of the embeddings which can be used to register to a sensor.

The last step is to calculate the norm of each individual hash against the mean hash. Those values are needed to generate the delta value which defines whenever another hash is likely to be the same embedding 1b. The maximum distance is then picked and multiplied by *phi* to calculate the *delta*.

2.1.5 Output

The generated hash is not a single value. In order to compare it against another embedding, we do need some additional information:

- *Hash*
- *Salt*
- *Divisor*
- *Delta*

The Rust implementation we are using does calculate all values with a floating point precision of 64-bits. This would lead to a rather big fuzzy hash if we keep in mind that the hash and the salt do have 512 elements each. Therefore it is possible to use 32-bit representation of those values as well because it will not change the comparison outcome that much. This saves 4KB of data for each fuzzy hash!

2.2 Comparing hashes

To compare an embedding against a fuzzy hash, we need to perform the following steps shown in the figure 2. The embedding we want to compare the hash against needs to be element-wise multiplied with the salt first. After that, the hash can be calculated with the divisor and the norm of both hashes can be calculated. If the norm is less equal than the provided delta, the hash most likely belongs to the embedding.

3. Accuracy

To analyse how accurate the fuzzy hash is, we are setting up a comparison with a standard embedding comparison dataset. We are using the *Labeled Faces in the Wild*[2] dataset, specifically the *pairs* dataset to make our comparison which includes 6000 tests. This dataset provides pairwise images of the same or another person and our face recognition model as well as our fuzzy hash will use this information to check if we can find the correct matches.

However, there is a bit of a pitfall with our fuzzy hash in this comparison as it usually takes multiple embeddings of a single person in order to calculate a stable hash from it. As this is not possible in this scenario, we do generate additional hashes by adding noise to the input embedding to get multiple embeddings.

3.1 Face recognition model

In order to see how accurate our face recognition model is, we use the *pairs* dataset and calculate the embeddings from the provided images. Each comparison will be checked with different thresholds to see which threshold is the best. The result is shown in the figure 3 where we can see on the threshold on the x-axis, starting from 0 to 2. The lower bound of the threshold values we get the most false negative which makes sense as the threshold is so low that it allows anything. At the upper bound we see the opposite as the threshold allows more distance as the distances from the embedding getting bigger. The sweet spot for the threshold is around *1.5* with an accuracy about *97%*.

[2] http://vis-www.cs.umass.edu/lfw/

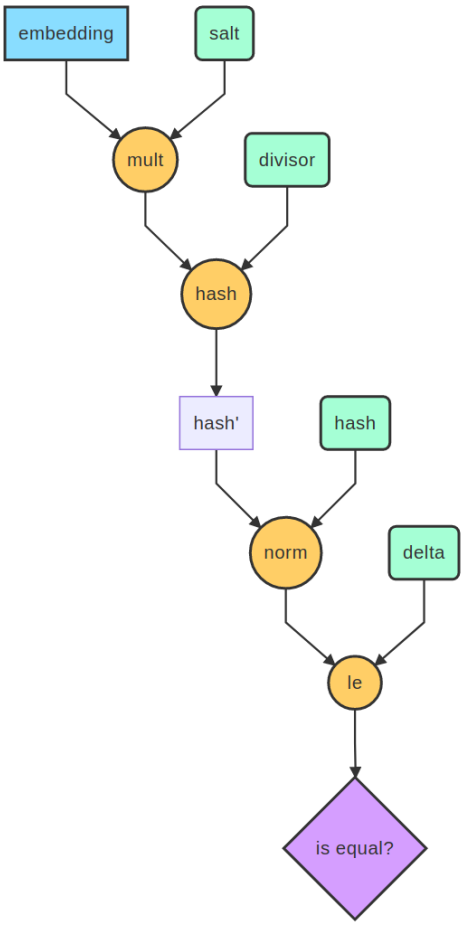


Figure 2: Comparing an embedding against a hash

3.2 Fuzzy hash

In order to compare it to the face recognition above, we need to exchange the threshold value for the *phi* value from the hashes. Adjusting it should give us a similar output which we then can compare. As already stated above, the fuzzy hash will suffer from the fact that we only get a single embedding from the dataset. To make it a bit more comparable, we take the first embedding and create a second and a third one by adding noise to it before we create our hash. This makes the results more robust and the comparison better.

Taking a look at the output in figure 4 we can see similar values. The biggest difference however is the steepness of our false-positives and false-negatives at the lower and upper bound of the figure. The sweet spot is around *0.31* with an accuracy of *91%*. This does not differ that much from our usual values with a phi of *0.35* where we still get an accuracy of *83%*. Considering the fact that we usually use multiple distinguished embeddings for the hash generation, we only drop about *12%*.

4. Attacks

Along the way of creating the concept of fuzzy hash and find methods to improve it, we also did create different attack scenarios. Various attacks were per-

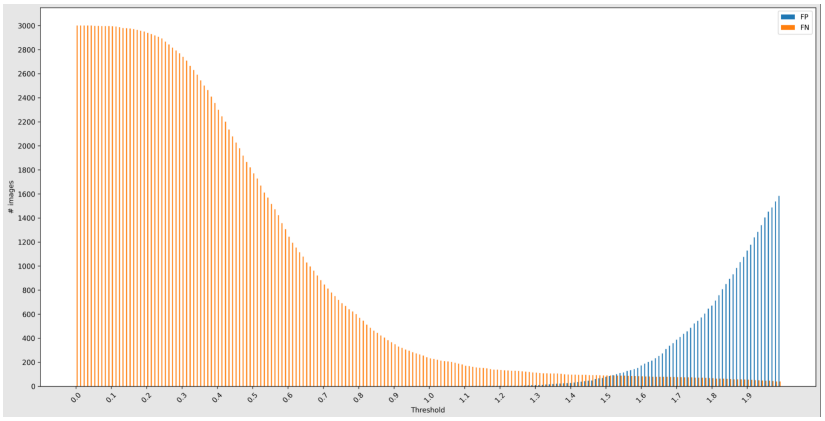


Figure 3: Threshold comparison with the *pair* dataset

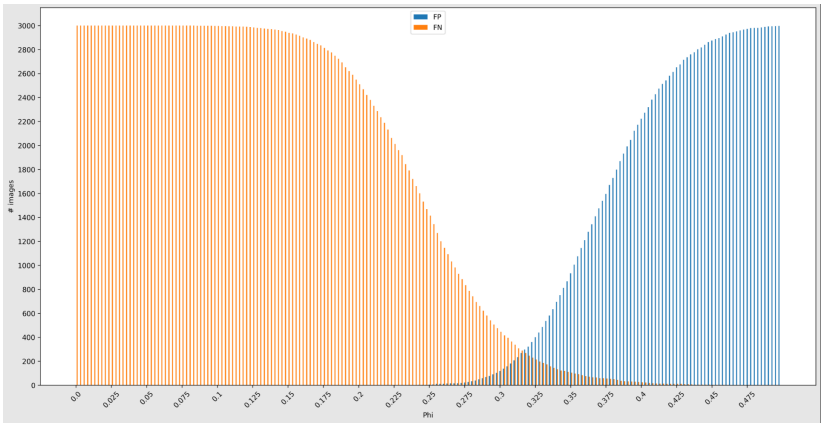


Figure 4: Phi comparison with the *pair* dataset

formed to see if the hash can hold up. The following sections will describe how they were performed and also why the hash persists them.

4.1 Brute force

One of the simplest attacks is the brute force attack. The idea is to go through all possible values and find the correct ones by comparing them against a specific target. To make the calculation easier, we first strip some details in the generation of the hash and only focus on the modulo operation:

$$hash(f) := \begin{cases} n_i - divisor * \lfloor \frac{n_i}{divisor} \rfloor, \text{if } n_i \geq 0 \\ n_i - divisor * \lceil \frac{n_i}{divisor} \rceil, \text{if } n_i < 0 \end{cases}$$

To get all possible values of a single element from the hash we need to know the divisor and the hash value h for this element. Given the value space E we can calculate all the possible values R which has the original value from the embedding in it as well.

$$N := \{n \in \mathbb{N} | \lfloor \frac{2 * max(E)}{divisor} \rfloor\}$$
$$R := \{\forall x \in N | n * divisor + h\}$$

However, because we choose a very small number as the divisor, we get a lot of possible values just for a single element from the hash. Considering all the possible combinations from the 512 elements from the embedding, we do get quite a big probability value P.

$$P(hash) := \frac{1}{R^{512}}$$
$$P(hash) \approx \frac{1}{2.59 * 10^{8087}}$$

Even if all possible values are compared against a specific embedding it is not certain that the hash really belongs to it as, after all, it is a hash which do have a limited value space and could produce false positives. Another factor we did not considered here is the noise we are adding in the generation process which would prevent an exact reverse and only give approximated embeddings anyways.

4.2 Recalculation

There exists a more sophisticated method to recalculate the original embedding in comparison to the brute force attack. Lets assume that an attacker gets two hashes which definitely belongs to the same individual. In order to find the correct values, it is necessary to go through the value space of E in respect to the divisors. For each value x in the embedding we need to find the value of the correlating values for $hash_1$ and $hash_2$.

This is not an easy task either, but in contrast to the brute force attack, it is possible to do this calculation for each element in the embedding individually. But there is a pitfall in recovering the embedding this way as each of the possible fitting values we find could also be a multiple of itself. Although the amount of possible embeddings are smaller, this results in multiple possible values for the embedding again.

However, this method is not very stable. The script we are using to make this attack does not factor in the noise we add to the hash. Although it is possible to add this factor in if the exact *gamma* is known which was used to generate the noise, it will result in a basically brute force attack.

There is another issue with this approach. We assumed that an attacker knows that two hashes are definitely from the same individual. This is rather unlikely to be the case as it is not possible to find any correlation between two generated hashes (as shown in the next section). Assuming that the registration process is anonymized, the attacker do not have an attack surface.

4.3 Correlation analysis

Because we are using the same embeddings to create multiple hashes we need to ensure that those hashes do not correlate. This is done by using correlation analysis.

Although there are multiple correlation analysis methods, the following two are well known in the embedding community [5, 7, 2, 13, 14].

- *PCA* principal correlation analysis
- *T-SNE* t-distributed stochastic neighbor embedding

(a) 100 hashes without salt (b) 100 hashes with salt

Figure 5: Correlation analysis with PCA

Tensorflow does provide a tool called *Projector*[3] which can perform those two methods online. Both methods will reduce the 512 dimensions down to 3 or 2 respectively and visualize those either in a 3d or 2d. In our case we choose the 2d version for this report which means that an embedding will be rendered as a single point on a 2d surface.

4.3.1 PCA

We created two samples which we can compare. A sample holds 11 individuals with 100 hashes each. The first sample includes hashes without using salt and the PCA clearly shows how those individuals can be distinguished (figure5a). As a color stands for an individual we can see that hashes from a single individual can be grouped. However, it is notable that there are two individuals colored black and brown in this figure which are overlapping quite a bit.

The second sample is holding the same 11 individuals but using salt. This time we can see that the different colors which representing an individual are distributed evenly throughout the figure5b. It can be concluded that the hashes do not correlate with each other which is exactly what we want to see.

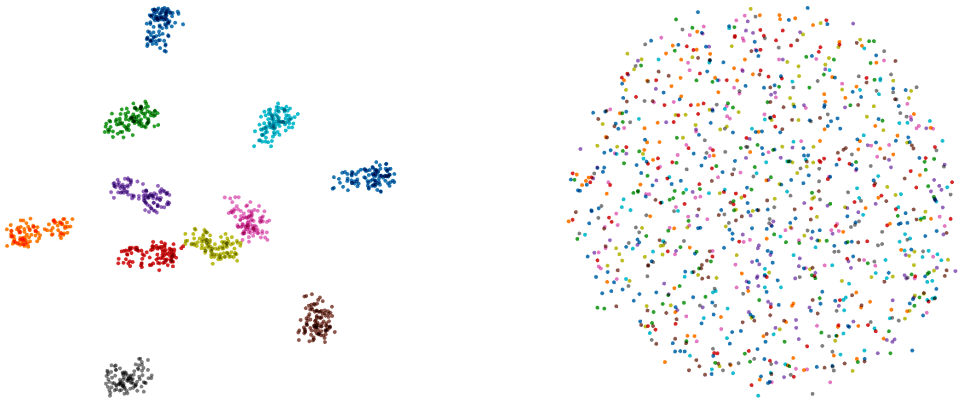
4.3.2 T-SNE

In contrast to the *PCA* analysis, the *T-SNE* is iterative which means that the outcome will take a bit more time. There are also multiple parameters which we can adjust to get better results (described in more detail on the *Projector* homepage).

We also did the same testing from the *PCA* analysis with *T-SNE*. The first figure shows the result without using salt 6a. We can see that the 100 hashes from each individuals are clearly grouped and distinguishable. There is no overlapping of individuals in this method compared to the *PCA* previously.

In figure 6b we are using salt for the hashing. Here we got the same result again, all hashes from the individuals are well distributed and cannot be correlated.

[3]https://projector.tensorflow.org



(a) 100 hashes with salt (b) 100 hashes with salt

Figure 6: Correlation analysis with T-SNE

4.3.3 Continuity

We did another analysis to show how the divisor is affecting the hash. This time we do not using salt to show that very small changes in the divisor also leads to very small changes in the hash 7. To show the differences, we create a hash from a random divisor and then create 100 more hashes by incrementing the divisor by a step of 10^{-17}. Then we calculate the distance of each of those 100 hashes to the original hash to see how much the distance will change. As the figure shows, if we change the divisor more and more, the distance is also getting bigger. This shows that the distance from a hash will change accordingly to divisor changes which means that the hash itself is not created randomly and thus can be used to compare embeddings.

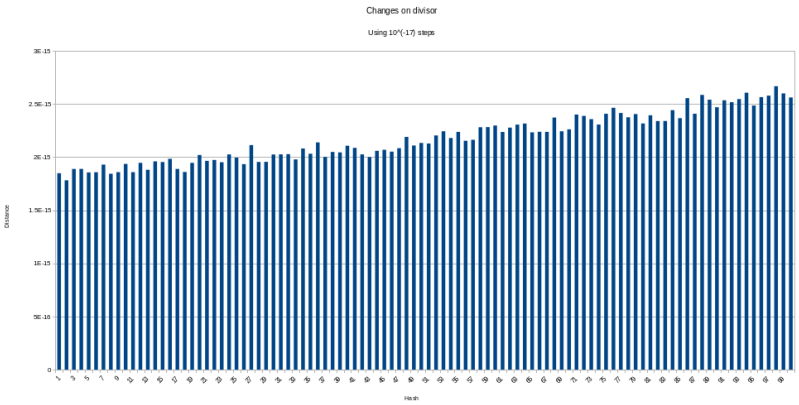


Figure 7: Distance of 100 hashes with small changes in divisor

5. Discussion

Although we proofed that the hash does not correlate to other hashes generated from the same data, it remains unclear why the actual fuzzy hash works. The most compelling explanation is that the generated vector from the hash function provides enough space to place similar embeddings in a nearby location. By using the euclidean distance to determine if two hashes are nearby emphasize

this. It might also explain why some of the generated hashes are doing worse than others in the comparison process, as the divisor might was chosen unlucky and spread similar embeddings more in the hash space. Unfortunately, this is unproven and remains open for future work.

5.1 Input variables

As explained in the section of creation and comparison of a hash, it is possible to change the static input variables to create a new hash. The comparison process does not need any adoption which means that one side can change input values whereas the other side can compare such a changed hash without any issue. However, this also means that the part which creates a hash can change values such that the created hash will fit to every embedding. This could be used to get informed if a sensor interacts with an individual.

Currently, this is subject for future investigation as it might can be solved by using an interactive protocol at the registration process.

5.2 Multiple registrations

To further increase the accuracy on sensors, it would be possible to generate multiple hashes from different embeddings, like frontal, left- and right-side. As they are not traceable, the sensor would not gain more information but the service would potentially get multiple matches back with a confidence value that can be used to make the validation more precise. Multiple registered hashes also increase the overall accuracy as it has shown that drops of accuracy occur depending on the random generated divisor. This can lead to significant drops from 98% down to 85%.

5.3 Bidirectional hashes

To validate if the sensor really interacted with the individual a service represents, it is useful that the sensor sends further information about the individual. Currently, the embedding from the sensor is sent back. However, this could also lead to leaks as such as an embedding might get send to the wrong service. To prevent this, the sensor itself could also make use of the hashes and create a fresh hash from the embedding which is then send to the service (as shown in figure 8). The service again only can compare the hash if it holds the correct embedding. As the service may hold multiple embeddings, it could verify it with a higher accuracy if this hash represents the correct embedding.

5.4 Performance

The performance of the creation of a hash depends how many embeddings are used. However, it is very fast to create hashes and compare them. On an actual Intel i7-10850H on 2.4GHz single threaded we get the following results:

- *Creation of 1.000 hashes:* 370ms (including parsing 5 embeddings from SSD each time)
- *Comparing of 10.000 hashes:* 760ms (including parsing the embeddings for the comparison from SSD each time)

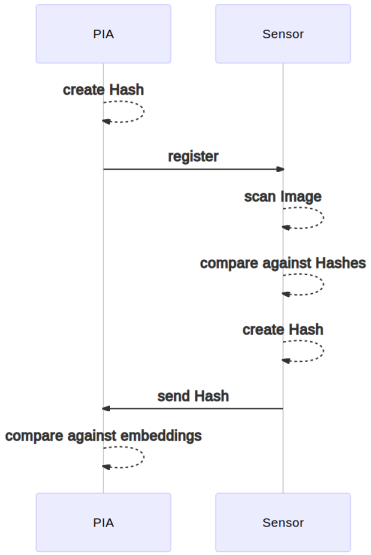


Figure 8: Sending back a hash instead of the embedding

This is without optimization like multi threading which would be easy to implement as the process of generating a and comparing can be split in multiple smaller tasks which can be run parallel.

6. Conclusion

Our fuzzy hash implementation can be used to create multiple unique hashes from the same embeddings which can be used to register to a sensor without reveal the original embeddings. We showed that it is resistant against correlation which prevents linking multiple hashes to a single individual. Furthermore, we tested our hashing method with an open dataset and compared it against our face recognition method directly. The results are in the lower 90 percentage which are good considering that the dataset were not optimal for our method as it gets better with multiple embeddings from an individual.

References

[1] Raymond T. Boute. 1992. The euclidean definition of the functions div and mod. *ACM Trans. Program. Lang. Syst.*, 14, 2, (April 1992), 127–144. ISSN: 0164-0925. DOI: 10.1145/128861.128862. https://doi.org/10.1145/128861.128862.

[2] Lih H Chan, Sh-Hussain Salleh, and Chee M Ting. 2010. Face biometrics based on principal component analysis and linear discriminant analysis. *Journal of Computer Science*, 6, 7, 693.

[3] Tee Connie, Andrew Teoh, Michael Goh, and David Ngo. 2005. Palmhashing: a novel approach for cancelable biometrics. *Information Processing Letters*, 93, 1, 1–5. ISSN: 0020-0190. DOI: https://doi.org/10.1016/j.ipl.2004.09.014. https://www.sciencedirect.com/science/article/pii/S0020019004002741.

[4] G.I. Davida, Y. Frankel, and B.J. Matt. 1998. On enabling secure applications through off-line biometric identification. In *Proceedings. 1998 IEEE Symposium on Security and Privacy (Cat. No.98CB36186)*, 148–157. DOI: 10.1109/SECPRI.1998.674831.

[5] Ian Jolliffe. 2005. Principal component analysis. *Encyclopedia of statistics in behavioral science*.

[6] Cagatay Karabat, Mehmet Sabir Kiraz, Hakan Erdogan, and Erkay Savas. 2015. Thrive: threshold homomorphic encryption based secure and privacy preserving biometric verification system. *EURASIP Journal on Advances in Signal Processing*, 2015, 1, 1–18.

[7] Sasan Karamizadeh, Shahidan M Abdullah, Azizah A Manaf, Mazdak Zamani, and Alireza Hooman. 2013. An overview of principal component analysis. *Journal of Signal and Information Processing*, 4, 3B, 173.

[8] Vishal M. Patel, Nalini K. Ratha, and Rama Chellappa. 2015. Cancelable biometrics: a review. *IEEE Signal Processing Magazine*, 32, 5, 54–65. DOI: 10.1109/MSP.2015.2434151.

[9] Shantanu D. Rane, Wei Sun, and Anthony Vetro. 2009. Secure distortion computation among untrusting parties using homomorphic encryption. In *2009 16th IEEE International Conference on Image Processing (ICIP)*, 1485–1488. DOI: 10.1109/ICIP.2009.5414544.

[10] N. Ratha, J. Connell, R.M. Bolle, and S. Chikkerur. 2006. Cancelable biometrics: a case study in fingerprints. In *18th International Conference on Pattern Recognition (ICPR'06)*. Volume 4, 370–373. DOI: 10.1109/ICPR.2006.353.

[11] A. Stoianov. 2010. Cryptographically secure biometrics. In *Biometric Technology for Human Identification VII*. B. V. K. Vijaya Kumar, Salil Prabhakar, and Arun A. Ross, editors. Volume 7667. International Society for Optics and Photonics. SPIE, 107–118. https://doi.org/10.1117/12.849028.

[12] Wilson Abel Alberto Torres, Nandita Bhattacharjee, and Bala Srinivasan. 2014. Effectiveness of fully homomorphic encryption to preserve the privacy of biometric data. In *Proceedings of the 16th International Conference on Information Integration and Web-Based Applications & Services* (*iiWAS '14*). Association for Computing Machinery, Hanoi, Viet Nam, 152–158. ISBN: 9781450330015. DOI: 10.1145/2684200.2684296. https://doi.org/10.1145/2684200.2684296.

[13] Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-sne. *Journal of machine learning research*, 9, 11.

[14] Jizheng Yi, Xia Mao, Yuli Xue, and Angelo Compare. 2013. Facial expression recognition based on t-sne and adaboostm2. In *2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing*, 1744–1749. DOI: 10.1109/GreenCom-iThings-CPSCom.2013.321.