Hindawi
Security and Communication Networks
Volume 2020, Article ID 4195852, 11 pages
https://doi.org/10.1155/2020/4195852

Research Article

Efficient Privacy-Preserving Fingerprint-Based Authentication System Using Fully Homomorphic Encryption

Taeyun Kim,[1] Yongwoo Oh,[1] and Hyoungshick Kim [1,2]

[1]*Sungkyunkwan University, Suwon, Republic of Korea*
[2]*CSIRO Data61, Marsfield, NSW, Australia*

Correspondence should be addressed to Hyoungshick Kim; hyoung@skku.edu

Received 19 September 2019; Accepted 17 January 2020; Published 13 February 2020

Academic Editor: José María de Fuentes

To help smartphone users protect their phone, fingerprint-based authentication systems (e.g., Apple's Touch ID) have increasingly become popular in smartphones. In web applications, however, fingerprint-based authentication is still rarely used. One of the most serious concerns is the lack of technology for securely storing fingerprint data used for authentication. Because scanned fingerprint data are not exactly the same each time, the use of a traditional cryptographic hash function (e.g., SHA-256) is infeasible to protect raw fingerprint data. In this paper, we present an efficient privacy-preserving fingerprint authentication system using a fully homomorphic encryption scheme in which fingerprint data are always stored and processed in an encrypted form. We implement a fully working fingerprint authentication system with a fingerprint database (containing 4,000 samples) using the Fast Fully Homomorphic Encryption over the Torus (TFHE) library. The proposed system can perform the fingerprint matching process within about 166 seconds (±0.564 seconds) on average.

1. Introduction

To enhance the usability of smartphone locking, smartphone manufacturers started providing fingerprint-based authentication (e.g., Apple's Touch ID) as an option. With the advances in fingerprint sensing technology, fingerprint-based authentication provides incredibly low false acceptance rate (FAR) and reasonably low false rejection rate (FRR) (e.g., FAR = 0.01% and FRR = 0.1% in Fingerprint Vendor Technology Evaluation [1]) and has finally become the most preferred authentication method on smartphones [2, 3].

However, in web applications, fingerprint-based authentication is still rarely used despite its usability advantages over password-based authentication. One of the most serious concerns is the lack of technology for securely storing fingerprint data used for authentication. Unlike password-based authentication, a traditional cryptographic hash function (e.g., SHA-256) cannot be used to protect raw fingerprint data because only a partial fingerprint can be typically obtained with noise from a fingerprint sensor on smartphone. Because it is very challenging to extract exactly same biometric features each time from the fingerprint sensor even when a physically same fingerprint is used, a fingerprint matching algorithm is generally used to compute the degree of similarity between two fingerprints with their partially matched features [4]. For this reason, the user's biometric information (e.g., raw fingerprint image or fingerprint template) is typically stored in plaintext in the fingerprint database [5].

In this paper, we propose an efficient privacy-preserving fingerprint authentication system using a fully homomorphic encryption scheme in which fingerprint data are always stored and processed in an encrypted form. We aim to minimize the time needed for completing the tasks in fingerprint verification without compromising the accuracy of user identification. To show the feasibility of the proposed system, we implement a fully working fingerprint authentication system with a fingerprint database using the Fast Fully Homomorphic Encryption over the Torus (TFHE) library (https://tfhe.github.io/tfhe/). Our key contributions are summarized as follows:

(1) We propose a privacy-preserving fingerprint-based authentication method using a fully homomorphic encryption scheme and implement the first *fully working* system integrated with a fingerprint database implemented using MySQL. Our source code is available at https://github.com/taeyun1010/HomFingerPrintAuth. We extend the TFHE library to calculate Euclidean distance between two encrypted vectors of double-precision floating-point numbers for filterbank-based fingerprint-based authentication scheme without loss of accuracy [6]. This is a significant advancement from previous studies that merely used a partially homomorphic encryption scheme [7] and lacked a real implementation of working system [8].

(2) With "NIST Special Database 9" containing 4,000 fingerprint samples, we implement the privacy-preserving fingerprint-based authentication system with an integrated MySQL database supporting SQL queries on encrypted data using a MySQL proxy server and evaluate the performance of the proposed system. Our implementation is capable of performing the fingerprint matching process within about 166 seconds (±0.564 seconds) on average.

2. Background

2.1. Homomorphic Encryption. We use homomorphic encryption to encrypt fingerprint data. Homomorphic encryption is an encryption scheme that allows computations to be held on encrypted data. For example, we can use homomorphic encryption to calculate the distance between fingerprint vectors even in encrypted form. When we decrypt the result of computations that were held on homomorphically encrypted data, the result is guaranteed to be the same as the result of performing the same computations on plaintext data. Figure 1 explains the relationship between plaintext and ciphertext under homomorphic encryption.

There are two types of homomorphic encryption schemes: partially homomorphic encryption and fully homomorphic encryption. Partially homomorphic encryption only allows certain types of computations to be performed, whereas fully homomorphic encryption allows any arbitrary computations to be performed. In this paper, we use fully homomorphic encryption since it allows us to perform filterbank-based fingerprint verification algorithm on encrypted fingerprint vectors without any loss of accuracy.

Fully homomorphic encryption scheme was first proposed by Gentry [9]. Several libraries were introduced to support implementation of fully homomorphic encryption. The HElib library (https://github.com/shaih/HElib) [10] implements the BGV cryptosystem with the GHS optimizations, the FHEW library (https://github.com/lducas/FHEW) [11] implements a combination of Regev's LWE cryptosystem with the bootstrapping techniques of Alperin-Sheriff and Peikert [12], and the TFHE library [13] proposes a faster variant over the Torus with an intuitive API to evaluate Boolean circuits. In this paper, we use the TFHE library that is freely available on GitHub since it provides

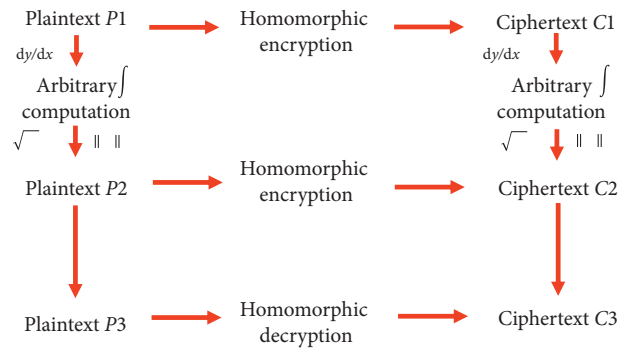

Figure 1: Relationship between plaintext and ciphertext under homomorphic encryption.

relatively more efficient bootstrapping than other libraries. However, our system design is not specifically limited to a certain type of homomophic encryption library but can be flexibly implemented with any homomorphic encryption library.

TFHE (written in C++) can theoretically support any computations to be performed on encrypted data. However, the library only allows computations to be performed on a bit-level. It also only provides some basic gate operations (NOT, NAND, OR, AND, XOR, etc.), a CONSTANT operation, and a MUX operation that can be performed on encrypted data. The library uses a secret key to encrypt and decrypt a bit. A secret key generated by the TFHE library typically has a size of about 82 MB. When a bit is encrypted using a secret key, a C++ struct called `LweSample` which contains an array of 500 integers (`a,` array), one integer (`b`), and one double-data type (`variance`) is formed. The encryption is not deterministic, that is, everytime the same bit is encrypted, the encrypted result is not guaranteed to be the same. Furthermore, the library uses a so-called cloud key to perform computations on encrypted bits. Similar to a secret key, a cloud key also has a size of about 82 MB. The separation of a secret key from a cloud key allows anyone with a cloud key to perform calculations on encrypted data but cannot decrypt the encrypted data or encrypt the plaintext data since they does not have a secret key. The CONSTANT operation provided by the library allows us to create a `LweSample` that encrypts one bit using only a could key. The MUX operation allows us to perform a multiplexer operation on encrypted bits and return a different encrypted value depending on whether the encrypted bit corresponds to the encrypted bit of 0 or to the encrypted bit of 1.

We also test the performance of our fingerprint-based authentication using FHEW, which is another fully homomorphic encryption library. The FHEW library operates in a similar fashion as the TFHE library. It only provides a few basic gate operations on encrypted bits and uses a secret key and an evaluation key to perform these operations. A secret key typically takes 2 KB, whereas an evaluation key requires about 2.5 GB. To show the effectiveness of our extension of the TFHE library, we compare the implementation of our fingerprint-based authentication using TFHE with the implementation using FHEW. For our implementation, we used filterbank-based fingerprint

verification [6] that is popularly used for fingerprint-based authentication.

2.2. Filterbank-Based Fingerprint Verification. We use filterbank-based fingerprint verification [6] as our underlying fingerprint verification algorithm to devise privacy-preserving biometric authentication scheme. The algorithm uses FingerCodes consisting of feature vectors extracted from the given fingerprint. Since the algorithm is based on the Euclidean distance between the two corresponding FingerCodes, it is relatively faster than the minutia-based fingerprint verification [6]. The algorithm requires Euclidean distance to be computed between extracted fingerprint vectors, meaning that we have to be able to compute Euclidean distance between encrypted fingerprint vectors if we were to devise privacy-preserving fingerprint-based authentication.

The algorithm works as follows. First, a reference point of a given fingerprint is located. Here, a reference point is defined as a point of maximum curvature of the concave ridges in the fingerprint image. Next, fingerprint image pixels around the reference point are tessellated, and this is done by grouping pixels that are close together into sectors. Gabor-filtering which removes noise and preserves the ridge and valley structures of fingerprint image is applied to all sectors afterwards. Finally, the average absolute deviation of gray values from the mean are calculated for all sectors, and these values become the FingerCode vector for given fingerprint image. Typically, multiple FingerCodes having the values from rotated fingerprints are stored for each fingerprint to get rid of any rotation-variant given fingerprint might have. These FingerCodes can be used to determine if two fingerprints originated from the same person.

3. Related Work

The use of privacy-preserving biometric authentication has been widely studied. Barni et al. [8] also deployed homomorphic encryption to filterbank-based fingerprint verification. Their work makes use of two partially, additively homomorphic encryption schemes: the Paillier's encryption scheme [7] and a known-variant of the ElGamal encryption scheme [14] but ported on Elliptic Curves. The two encryption schemes are used to compute the square of Euclidean distances between encrypted fingerprint vectors. However, the work requires a template quantization step, where fingerprint vectors of doubles are rounded to vectors of integers to calculate the distance between the vectors using the two encryption schemes. This rounding process can result in the loss of accuracy of the fingerprint recognition using the FingerCode template. However, our proposed system does not degrade the accuracy of fingerprint verification algorithm because our scheme makes use of fully homomorphic encryption to compute distances between encrypted fingerprint vectors without a quantization step. Unlike existing studies, we implement a fully working fingerprint-based authentication system, achieving a higher accuracy and faster time performance to show the feasibility of fingerprint-based authentication system using homomorphic encryption.

To develop a fully working system, we extend the TFHE library to support vector operations on encrypted data and integrated our system with MySQL database. Executing database queries on encrypted data has been introduced in [15]. The paper presents a system called CryptDB which provides database queries on encrypted data. The key idea is the use of multiple encryption layers for each column and maintaining these multiple layers. CryptDB provides several layers including RND, DET, HOM, etc. In particular, HOM layer is a homomorphic encryption layer which allows addition to be performed in queries, thereby allowing SUM aggregate queries on encrypted database. Such homomorphic operation is implemented by using user-defined functions (UDFs) of MySQL. Paillier's cryptosystem [7] is also used in CryptDB. Paillier's cryptosystem only allows queries that can be defined using addition operation on encrypted database. In this paper, we extend their work to fully encrypt a given column homomorphically to allow the users perform any computation, including the Euclidean distance calculation between encrypted fingerprint data, on queries that will be executed on encrypted database.

4. Extension of TFHE Library

To implement filterbank-based fingerprint verification [6] on encrypted domain, we should calculate the Euclidean distance between two encrypted vectors. As described in Section 2, the TFHE library only provides two gate-level logical operations: CONSTANT and multiplexer operations. Therefore, we need to extend the TFHE library to support the operations that are needed to calculate one-norm distance and square of Euclidean distance between two encrypted vectors.

4.1. Implementation of the Basic Arithmetic Operations. In this section, we describe how the operations such as addition and multiplication can be implemented using gate operations. Such implementation is necessary for 1-norm distance calculations and Euclidean distance calculations used in our authentication scheme since the TFHE library only provides two gate operations: CONSTANT and multiplexer operations. Implementations for other operations such as subtraction, absolute value calculation, and comparison of two encrypted vectors of positive numbers are omitted in this paper in the interest of space but are available in our GitHub project (https://github.com/taeyun1010/tfheExtension). We also mention that some of the implementations, namely, addition and comparison, do not originate from us and are from a GitHub page that we found (https://github.com/tfhe/tfhe/tree/master/src/test). The algorithms displayed in this section closely follow the TFHE library syntax, so readers who are interested in implementing other operations on top of our operations can easily do so. Most of the functions' purpose should be clear from the names except for `new_LweSample_array(i)` function and `bootsCONSTANT()`

function. The `new_LweSample_array(i)` function allocates `LweSample*` of i bits for use in the future. Here, `LweSample` is a struct representing an encrypted integer, explained in detail in Section 2.1. The `bootsCONSTANT()` function initializes the given `LweSample*` parameter to either 0 or 1 using the `evalkey`. The `evalkey` parameter that appears in all algorithms is a type of `CloudKeySet`, which is a TFHE equivalent of evaluation key. Lastly, the `bootsMUX(a, b, c)` function is responsible for a multiplexer operation, and it returns a?b:c, i.e., it returns b if a is an encrypted result of bit 1 and c otherwise.

4.1.1. Addition. Homomorphic Addition operation between two encrypted integers can be implemented using gate operations and a multiplexer operation, and the pseudocode is shown in Algorithm 1.

The algorithm is based on the simple logic equations $S = X \oplus Y \oplus C_{\text{in}}$ and $C_{\text{out}} = \text{MUX}_{X \oplus Y}(C_{in}, X \cdot Y)$ that can be performed on bits. Here, X and Y are bits that are being added, C_{in} is a bit that is carried in from the one less significant digit, and C_{out} is a bit that is to be carried to the next significant digit. In the multiplexer operation $\text{MUX}_{X \oplus Y}(C_{\text{in}}, X \cdot Y)$, the operation outputs C_{in} if $X \oplus Y$ is equal to 1 and outputs $X \cdot Y$ otherwise. Using this bit-level addition, we can expand it to the addition between two integers.

The algorithm implements addition between two integers using the TFHE library, and it first initializes two carries, C_{in} and C_{out}, by allocating a variable carry which has enough space for two carries. Then, the algorithm iterates each bit one by one and performs the two forementioned equations on each bit. After the algorithm is complete, the variable sum contains an encrypted sum and may have one more bit than the two inputs do, depending on whether the addition between the two most significant digits resulted in a carry or not.

4.1.2. Multiplication. Multiplication between encrypted integers can be implemented by using combination of addition and shift of bits. The pseudocode for multiplication is shown in the following Algorithm 2.

Multiplication can be seen as additions of shifted ANDed values. During multiplication process, twice the number of bits is allocated for intermediate sums, but to match the number of bits that is input to the algorithm, the first number of bits bits is discarded. Note that by increasing the number of bits, we can arbitrarily reduce the error caused by such loss of information.

4.2. Extension to Double-Precision Floating-Point Format. To calculate 1-norm distance and square of Euclidean distance between encrypted integers, we implemented addition, subtraction, multiplication, absolute value calculation, and comparison of two encrypted vectors of positive numbers. These algorithms can be used to perform calculations on encrypted double-precision floating-point format data types. To represent an encrypted double-data type, we allocate a new struct called Double which has two `LweSample*` fields: `integerpart` and `fractionpart`. This is depicted in the following.

```
struct Double{
  LweSample* integerpart;
  LweSample* fractionpart;
};
```

The `integerpart` holds the encrypted integer part of the given double data type, and the `fractionpart` holds the encrypted fractional part. In addition, we also maintain the variables named integerbitsize and fractionbitsize which tell us how many bits are used to represent the integer part and fractional part. To perform the forementioned calculations, we first append the two fields to form one `LweSample*` by allocating a new `LweSample*` and copy `integerpart` followed by `fractionpart` to allocated `LweSample*`. Then, we perform the calculations on the newly formed `LweSample*`. This yields the calculation result in `LweSample*` struct which also has an `integerpart` followed by `fractionpart`, and to form the resulting Double struct, we divide the resulting `LweSample*` again to `integerpart` and `fractionpart`. Note that when multiplication is done, we are left with twice the number of bits than we started. When copying the resulting `LweSample*` to Double struct, we therefore discard the first fractionbitsize and the last integerbitsize of bits to maintain the bitsize. However, the error caused by such loss of information can be arbitrarily reduced by simply increasing the integerbitsize and fractionbitsize as much as the user wants. In particular, we can calculate the Euclidean distance between two different fingerprint double vectors without rounding them to integer vectors on encrypted domain. This allows us to perform filterbank-based fingerprint matching algorithm on encrypted domain without any degradation in matching performance by using the same number of bits that is used in the original algorithm that uses plaintext fingerprint data.

4.3. Optimization. To boost the performance of each operation used in filterbank-based fingerprint verification, we designed more efficient arithmetic operations to be executed in parallel.

For instance, in our addition operation, we can compute the sum bit and the carry bit at the same time, without having to wait for the sum bit to be calculated before computing the carry bit. As another example, in multiplication operation, additions of shifted values can be done in parallel since none of the additions depend on the output of the other additions.

5. Privacy-Preserving Filterbank-Based Fingerprint Verification

The general flow of our privacy-preserving filterbank-based fingerprint verification scheme is shown in Figure 2.

In this scheme, a secret key is only given to each user, and an honest-but-curious server holds an evaluation key alone

```
program HomAddition (LweSample *x, LweSample *y, int numberofbits, CloudKeySet *evalkey)
  //two carries
  LweSample *carry = new_LweSample_array(2);
  //the first carry is intialized to 1
  bootsCONSTANT(carry, 1, evalkey);
  //two temps
  LweSample *temp = new_LweSample_array(2);
  For int i: 0 to (numberofbits - 1)
    temp = bootsXOR(x + i, y + i, evalkey);
    //sum[i] = x[i] XOR y[i] XOR carryin
    sum + i = boots XOR(temp, carry, evalkey);
    temp + 1 = boots AND(x + i, y + i, evalkey);
    //carry out
    carry + 1 = boots MUX(temp, carry, temp + 1, evalkey);
    carry = carry + 1;
  end
  sum + numberofbits = carry;
  return sum;
end
```

Algorithm 1: Homomorphic addition.

```
program HomMultiplication (LweSample *x, LweSample *y, int numberofbits, CloudKeySet *evalkey)
  LweSample *partialsum = new_LweSample_array(numberofbits * 2 + 1);
  bootsCONSTANT(partialsum, 0, evalkey);
  LweSample *temp = new_LweSample_array(numberofbits);
  LweSample *temp2 = new_LweSample_array(numberofbits * 2);
  LweSample *temp3 = new_LweSample_array(numberofbits * 2);
  For int i: 0 to (number of bits - 1)
    LweSample *ybit = y + i;
    //ANDing, temp stores ANDed value
    For int j: 0 to (number of bits - 1)
    bootsAND(temp + j, x + j, ybit, evalkey);
    end
    //shifting, temp2 stores shifted ANDed value,
    //filled with zeros in remaining bits
    For int j: 0 to (i - 1)
      bootsCONSTANT(temp2 + j, 0, evalkey);
    end
    For int j: i to (i + number of bits-1)
      bootsCOPY (temp2 + j, temp + j - i, evalkey);
    end
    For int j: i + number of bits to (number of bits * 2 - 1)
      bootsCONSTANT(temp2 + j, 0, evalkey);
    end
    //copy partialsum to temp3
    For int j: 0 to (number of bits*2 - 1)
      bootsCOPY(temp3 + j, partialsum + j, evalkey);
  end
  partialsum = HomAddition(temp2, temp3, number of bits * 2, evalkey);
  end
  For int i: 0 to (number of bits * 2 - 1)
    bootsCOPY(product + i, partialsum + i, bk);
  end
  return product;
end
```

Algorithm 2: Homomorphic multiplication.

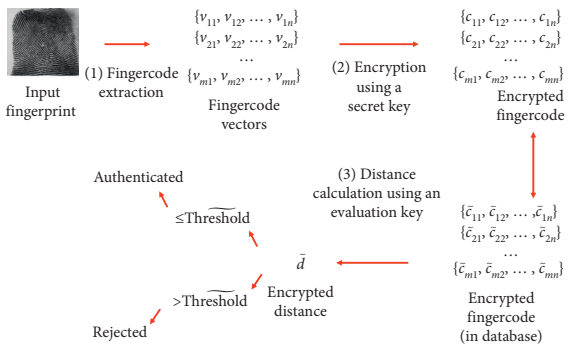

Figure 2: Proposed authentication procedure for privacy-preserving fingerprint verification.

to allow the server to perform Euclidean distance calculation and make authentication decision without uncovering the user's fingerprint data. We assume that the user can be entrusted with a secret key that can be used to encrypt the user's fingerprint data.

The fingerprint data, which is first input by the user who would like to be authenticated, is provided as an image format such as bmp or png format. Then, using the image file, grayscale value for every pixel of the image file is computed. Using these values, FingerCode vectors can be extracted by using the filterbank-based fingerprint matching algorithm. The extracted FingerCode vectors can be encrypted using a secret key, returning encrypted FingerCode vectors. These steps are depicted in Figure 2 as steps 1 and 2, and they must be performed on the client side where a secret key is entrusted. The resulting encrypted FingerCode can now be sent to the honest-but-curious server since the server has no way of figuring out the user's fingerprint data because it does not have a secret key. However, with an access to an evaluation key, the server can still compute the square of Euclidean distance between two encrypted FingerCode vectors, namely, the encrypted vector that has been just calculated and another encrypted vector that is stored in the database. We calculate the square of Euclidean distance rather than Euclidean distance itself to save the computation time while maintaining the accuracy. The calculation results in an encrypted distance, and the server must be able to tell if the distance is less than or greater than the encrypted threshold without decrypting the distance to make the authentication decision. This step is shown as step 3 in Figure 2. Finally, after the encrypted decision result is made, the server can decide whether to authenticate the user or not and take an appropriate action by using the MUX operation with the encrypted decision bit.

In our implementation, we used the FHEW library and the TFHE library, both available on GitHub, to show the generality of the proposed scheme. To recommend the best implementation solution, we also compare the time performance when using these two different fully homomorphic encryption libraries. For filterbank-based fingerprint verification algorithm, we used a MATLAB implementation that was available on GitHub (https://github.com/hbhdytf/FingerCode). For the fingerprint dataset, we used "NIST Special Database 9," which contains fingerprint images of 2,000 different individuals. The image size is 832 * 768 pixels, and the images are scanned at 500 dpi. The dataset contains 2 different fingerprint instances of the same finger per individual, resulting in a total of 4,000 fingerprints.

6. SQL Queries on Encrypted Fingerprint Data

In this section, we explain how we can store the values that are encrypted using the TFHE library to relational databases and how we can query the encrypted databases. The general flow of our storage procedure is depicted in Figure 3.

Similar to a technique in CryptDB [15], we also use a database proxy to intercept queries provided by a user and modify them to queries that can be performed on encrypted databases. During the modification process, the proxy server may require a secret key. However, this secret key is only given to the proxy that can be integrated at the client side and is not given to the honest-but-curious DBMS server. As in CryptDB [15], we assume that the attacker is passive; the attacker wants to learn confidential data but does not change queries. The MySQL proxy can be implemented by using a Lua script to import C++ functions from the TFHE library, and the DBMS server can make use of MySQL's UDFs and .so files to define new functions using the TFHE library so that any arbitrary queries can be performed on the encrypted database. Therefore, we built a fully working implementation to support database queries on encrypted databases. Our source code is available at https://github.com/taeyun1010/mysqlproxy_product.

6.1. Database Schema. To store the encrypted fingerprint vector feature values, we create a column for each integer value in a `LweSample` struct and also define the fields of `userid` and `integerorder` to represent the information about the user of the fingerprint vector and its specific feature, respectively. We note that these `userid` and `integerorder` values can be stored in plaintext in the database if we assume that attackers do not obtain any private information from those values; otherwise, we can also securely encrypt these values because the equality function of the two values can be securely evaluated even when they are encrypted. An example database storing one encrypted integer is represented in Table 1.

We create a separate table for each bit of an integer. When a user wants to insert an integer into the encrypted database, the user simply executes a query as if we execute that query on a plaintext database. The MySQL proxy intercepts a query and modifies it to execute the query on an encrypted database. Finally, the MySQL proxy sends the modified query to the DBMS server. This process is also shown in Figure 3. In the figure, the user wants to insert a plaintext value of 50 and thus executes an insert query with a value of 50 as shown in step 1 of the figure. After receiving the query from the user, MySQL proxy encrypts 50 using a secret key and executes multiple insert queries containing encrypted values on tables stored in DBMS server. These steps are shown as steps 2 and 3 in Figure 3.

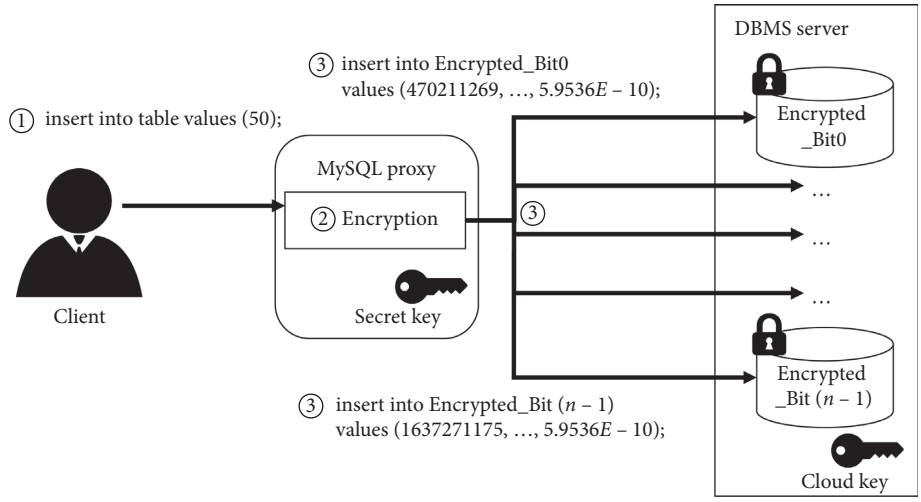

Figure 3: Inserting encrypted data into databases using a MySQL proxy.

Table 1: Example database storing encrypted integers.

Table	Userid	Integerorder	0th_a	…	499th_a	b	Variance
Bit0	1	1	470211269	…	−1455671213	1110023459	5.9536E – 10
…	…	…	…	…	…	…	…
Bit$n$ – 1	1	1	1637271175	…	−463281669	−602048671	5.9536 – 10

6.2. Executing Queries on Encrypted Data in a Database. After encrypting and storing values into encrypted database, the DBMS server can perform any arbitrary queries using an evaluation key and UDFs. For example, queries as simple as select queries that are used to retrieve values in a plaintext format from the encrypted database and queries to calculate Euclidean distance can also be done on the server side while securely storing the user's fingerprint vectors. An example execution of select query is depicted in Figure 4.

In our implementation, the proxy can process multiple queries to different tables in parallel by modifying the query into different ones. After the proxy receives the query results from the DBMS server, it can gather the results together, reconstruct the `LweSample` struct, and decrypt the resulting struct by using a secret key. Finally, the proxy can return the decrypted result to the user. All of these processes are not transparent to users because the encryption process is too complex for the average user who is not familiar with homomorphic encryption. In the proposed system, we can simply create SQL queries with the assistance of the proxy as if the queries are executed on a plaintext database.

6.3. Storing Double-Precision Floating-point Format Values. Filterbank-based fingerprint verification makes use of FingerCode vectors consisting of double-precision floating-point numbers. Hence, we must be able to store floating-point numbers. We can store floating-point numbers by maintaining `userid` and `order` fields to identify a user's specific fingerprint vector part in the same manner of storing integer numbers and introducing additional tables for the integer part and fractional part of a floating-point number.

6.4. Implementation Environment. MySQL enables a proxy server to be easily implemented by using a Lua script [16]. We also implement a MySQL proxy using a Lua script to intercept SQL queries from a user and modify them to be executed on encrypted domain. On the DBMS server side, UDFs can be implemented by using a `.so` file generated by compiling a C++ code which imports the TFHE library. In our test setting, we used an Intel Core i5-7500 CPU with 3.40 GHz ∗ 4 and 64-bit Ubuntu 16.04 LTS.

7. Evaluation

7.1. Effects of the Number of Features and Number of Bits. We tested how different number of features impacted the accuracy and time performance of filterbank-based fingerprint verification for "NIST Special Database 9" dataset. The tested configurations are shown in Table 2. The table also shows (Equal Error Rate) differences. Equal Error Rate (EER) is defined as the rate where false acceptance rate and false rejection rate are equal.

The table also shows how much EER increased compared to when 1,280 features are used. We observe that reducing the number of features does not severely impact EER when the number of features is not extremely low. For instance, using 640 number of double-data-type features only increases EER by about 0.0085 compared to when 1,280 doubles are used. This is echoing prior work [8]. Rounding vectors of doubles to vectors of integers did not have significant impact on EER. Increasing the number of features also increases the time it takes to authenticate. When higher number of features is used, the time it takes to calculate the Euclidean distance increases, although not exactly in a linear fashion. The time taken to calculate Euclidean distances for

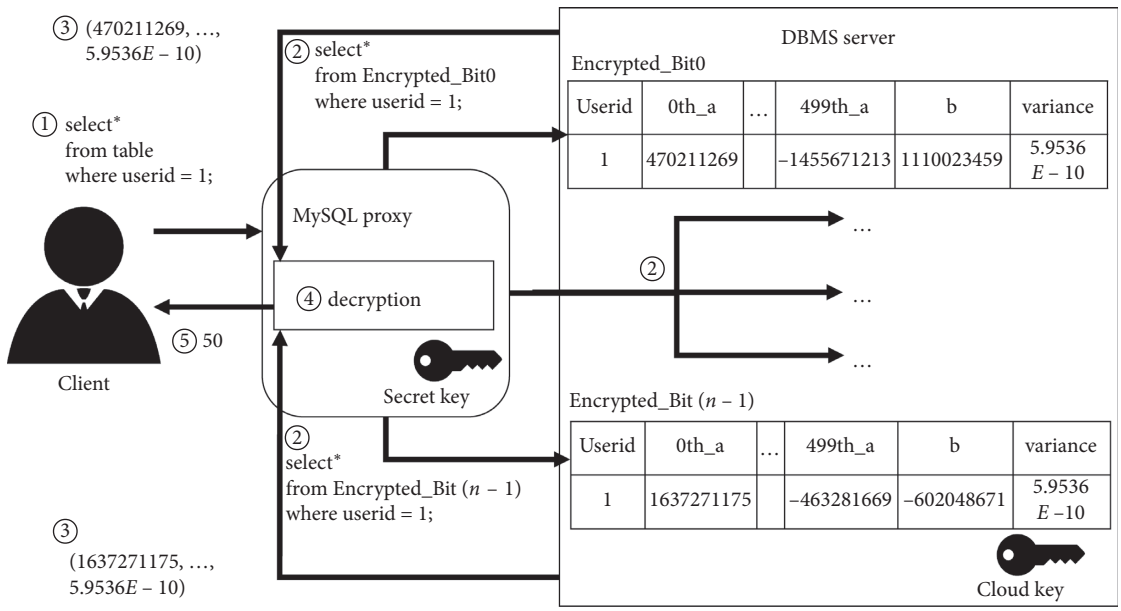

Figure 4: Executing select queries on encrypted data from databases using a MySQL proxy.

Table 2: Tested configurations.

No. of sectors	No. of disks	Total no. of features	EER differences
80	16	1280	0
80	8	640	0.0085
48	8	384	0.0065
8	2	16	0.021
4	2	8	0.033
1	2	2	0.053

Total no. of features = no. of sectors. ∗ no. of disks.

2,000 different pairs of double-data-type fingerprint vectors using different number of features is shown in Table 3.

Table 3 also shows how much time is required to compare one pair of fingerprints, i.e., the total time divided by 2,000. Since the time it takes to authenticate notably grows as the number of features increases, we propose using the minimum number of features while also maintaining an acceptable EER value.

In addition, we also varied the number of bits used to represent the integer and fractional part of double-precision value stored in a fingerprint vector and recorded how these changes impacted the EER values. The results are shown in Figure 5.

The z-axis of the figure stands for EER differences, i.e., how much EER increased with the number of bits. We note that reducing the number of bits used to represent the fractional part of a floating-point number has no significant impact on EER, but reducing the integer bits can increase EER substantially. However, we also note that increasing the number of bits further does not have much impact on EER when a sufficient number of bits is used to represent fingerprint vectors. According to our experiment results, we found that the filterbank-based fingerprint verification algorithm produced a reasonable level of EER when the size of integer bits representing the fingerprint vector is at least 7 bits.

7.2. Execution Time. We evaluate the time performance of our implementation to show its feasibility. We implemented the proposed scheme with the TFHE and the FHEW libraries, respectively, to optimize the effectiveness of our implementation.

The implementation using the TFHE library produced better performance results. Based on our optimized implementations, we calculated 1-norm distance between encrypted vectors that have 16 different 10 bit integers within about 49.637 seconds (±0.124 seconds) when the TFHE library was used. This is a significant improvement compared to 209 seconds when using the FHEW library. Thus, we recommend using the TFHE library. We also note that increasing the number of bits used to represent integers in vectors during 1-norm distance calculation linearly increased the 1-norm distance calculation time.

As expected, Euclidean distance calculation between encrypted fingerprint vectors took much longer. Whereas using vectors containing 16 different 10-bit integers required about 106 seconds (±0.159 seconds), using vectors containing 16 different 32-bit integers required about 597 seconds (±20.0 seconds) to calculate 2-norm distances. Using vectors containing 16 different 16-bit integers required about 166 seconds (±0.564 seconds) (see Table 4). These results indicate that our proposed system is capable of

Table 3: Time to calculate Euclidean distances after varying the number of features used in fingerprint verification.

Total # of features	Total time (s)	Time per pair (s)
1280	1236	0.618
640	662	0.331
384	539	0.270
16	217	0.109
8	152	0.076
2	121	0.061

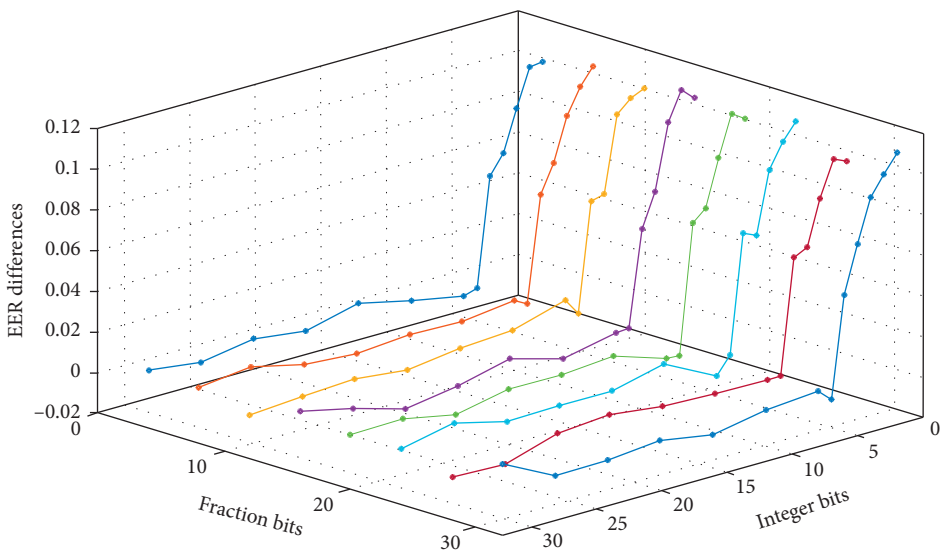

Figure 5: How changing the number of bits impacts EER.

performing the fingerprint matching process within about 166 seconds on average (±0.564 seconds). We note that the time did not increase in a linear fashion as the number of bits increased. As for the calculation of Euclidean distance between vectors of encrypted doubles, it took about 1,564 seconds (±4.69 seconds) when 13 bits were used to represent an integer part and 2 bits were used to represent a fractional part.

As for the storing of encrypted data to MySQL database, encrypting and inserting a 16-bit integer into the database took about 0.27 seconds using the schema and methods described in Section 6. Retrieving data by using a select query and decrypting a ciphertext into a 16-bit integer took about 2.045 seconds. Inserting 32,000 rows (equal to the number of the whole "NIST Special Database 9" dataset) which would result from representing each fingerprint vector with 16 integers (features), assuming they are already encrypted, took 14 hours, whereas inserting 2,000 fingerprint vectors without encryption took 3 minutes. If we include the encryption step, the time grows even more since encrypting and inserting a fingerprint vector having 16 integers along with the `userid` field to the encrypted database took about 27.07 seconds. Thus, the total time to encrypt "NIST Special Database 9" dataset and insert encrypted vectors into MySQL database would take about 15 hours. However, we claim that this time overhead is worthwhile since we can perform privacy-preserving calculations for highly sensitive data such as individuals' fingerprints afterwards once we store the dataset in encrypted form.

Inserting 32,000 rows (equal to the number of the whole "NIST Special Database 9" dataset) using the schema mentioned in Section 6.3 to store already encrypted fingerprint vectors that contain 16 doubles took 29 hours when we used 16 bits to represent an integer part of double and 16 bits to represent fractional part of double. This time is approximately twice the time it took to insert the same number of rows when an integer was used to represent a fingerprint vector. This is expected as fractional bits are stored in exactly the same manner as integer bits are stored, and the time to store only depends on the total number of bits to represent a double or an integer.

The experiment results of our implementation demonstrate that the execution time overheads (about 166 seconds on average) incurred by the proposed system is not ignorable but would be acceptable for some applications (e.g., biometric-based authentication) requiring the high confidentiality of user data.

7.3. Storage Overhead. Everytime a bit is encrypted using the TFHE library, a `LweSample` struct containing 501 integers and a double-precision floating-point number is returned. This may lead to potential growth in size when we store the

Table 4: Time to calculate distances between fingerprint vectors having different number of bits.

Distance type	No. of bits	Time (s)
Euclidean	10	106
Euclidean	16	166
1-norm	10	49.637

encrypted value in the database. We recorded how much storage would be needed to store "NIST Special Database 9" by inserting the same number of tuples as the dataset has into the database, both when data are unencrypted and encrypted using MySQL as a backend database. Using the table schema described in Section 6.1, after inserting 32,000 rows into each table (equal to the number of 2,000 individual's 16 integers fingerprint vector), each table which represents one bit had a size of about 71.88 MB. Storing the plaintext fingerprint vectors resulted in a database table size of about 0.23 MB. We note that when we store plaintext values, there is no need to keep track of which integer it is or which bit it is since we can store the whole fingerprint vector in one tuple. There was an expansion factor of about $(71.88 * 16)/0.23 \approx 5000$. As expected, storing encrypted double-data type values expanded the storage size proportionally to the total number of bits used to represent integer and fractional parts of double. For instance, using the same number of bits to represent integer and fractional parts, we need about twice the storage than we store the integer values. In other words, if we use two 16 bits to represent the integer part and the fractional part, respectively, each bit requires 71.88 MB. The expansion factor can be calculated as $(71.88 * 32)/0.23 \approx 10000$ in this case. We can see that there is a tradeoff between storage overhead and accuracy of the data.

8. Conclusion

In this paper, we implemented a fully working privacy-preserving fingerprint-based authentication system based on the filterbank-based fingerprint matching algorithm. For our implementation, we extended the TFHE library to calculate the Euclidean distance between vectors of encrypted positive numbers in double-precision floating-point format. We also provide a secure database implementation using a MySQL proxy to store encrypted values to and from a relational database under the honest-but-curious database management server.

To evaluate the performance of the proposed system, we constructed a MySQL database with 4,000 real-world fingerprint samples. Our experiment results demonstrated that an efficient fingerprint-based authentication system using fully homomorphic encryption can be indeed deployed, performing the fingerprint matching process within about 166 seconds (±0.564 seconds) on average without compromising the authentication accuracy.

We surmise that the performance of the proposed system, with further optimization techniques such as parallel computation for Euclidean distance calculation and the adoption of personalized threshold for each individual's fingerprint matching proposed in [8], could be improved. As part of future work, we plan to further optimize the performance of bootstrapping operation which is the most time-consuming step in fully homomorphic encryption schemes.

Data Availability

The code for fingerprint authentication system and MySQL proxy data used to support the findings of this study have been deposited in https://github.com/taeyun1010/tfheExtension, https://github.com/taeyun1010/mysqlproxy, and https://github.com/taeyun1010/HomFingerPrintAuth.

Conflicts of Interest

The authors declare that they have no conflicts of interest regarding the publication of this paper.

Acknowledgments

This work was supported by the ICT R&D Programs (no. 2017-0-00545) and the ITRC Support Program (IITP-2019-2015-0-00403).

References

[1] C. Wilson, A. Hicklin, M. Bone et al., "Fingerprint vendor technology evaluation 2003: summary of results and analysis report," NIST Technical Report NISTIR 7123, National Institute of Standards and Technology, Gaithersburg, Ma, USA, 2003.

[2] A. De Luca, A. Hang, E. von Zezschwitz, and H. Hussmann, "I feel like I'm taking selfies all day!: towards understanding biometric authentication on smartphones," in *Proceedings of the 33rd ACM Conference on Human Factors in Computing Systems*, Seoul, Republic of Korea, April 2015.

[3] S. Mare, M. Baker, and J. Gummeson, "A study of authentication in daily life," in *Proceedings of the 12nd Symposium on Usable Privacy and Security*, Denver, CO, USA, June 2016.

[4] X. Jiang and W.-Y. Yau, "Fingerprint minutiae matching based on the local and global structures," in *Proceedings of the 15th International Conference on Pattern Recognition*, vol. 2, Barcelona, Spain, September 2000.

[5] H. X. Y. Zhang, Z. Chen, and T. Wei, *Fingerprints on Mobile Devices: Abusing and Leaking*, Black Hat, Las Vagas, NV, USA, 2015.

[6] A. K. Jain, S. Prabhakar, L. Hong, and S. Pankanti, "Filterbank-based fingerprint matching," *IEEE Transactions on Image Processing*, vol. 9, no. 5, pp. 846–859, 2000.

[7] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 223–238, Springer, Prague, Czech Republic, May 1999.

[8] M. Barni, T. Bianchi, D. Catalano et al., "Privacy-preserving fingercode authentication," in *Proceedings of the 12th ACM*

Workshop on Multimedia and Security, pp. 231–240, ACM, Roma Italy, September 2010.

[9] C. Gentry, *A Fully homomorphic encryption scheme*, Ph.D. thesis, Stanford University, Stanford, CA, USA, 2009.

[10] S. Halevi and V. Shoup, "Algorithms in helib," in *Advances in Cryptology*, Springer, Berlin, Germany, 2014.

[11] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachne, "Faster fully homomorphic encryption: bootstrapping in less than 0.1 seconds," Cryptology ePrint Archive, Report 2016/870, 2016, https://eprint.iacr.org/2016/870.

[12] J. Alperin-Sheriff and C. Peikert, "Faster bootstrapping with polynomial error," Cryptology ePrint Archive, Report 2014/094, 2014, https://eprint.iacr.org/2014/094.

[13] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachne, "Improving TFHE: faster packed homomorphic operations and efficient circuit bootstrapping," Cryptology ePrint Archive, Report 2017/430, 2017, https://eprint.iacr.org/2017/430.

[14] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE Transactions on Information Theory*, vol. 31, no. 4, pp. 469–472, 1985.

[15] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: protecting confidentiality with encrypted query processing," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pp. 85–100, ACM, Cascais, Portugal, October 2011.

[16] Lua, https://www.lua.org/, 2018.