View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Waseda University Repository

2019 Master Thesis

A Privacy-preserving Query System using Fully Homomorphic Encryption with Real-world Implementation for Medicine-Side Effect Search

A Thesis Submitted to the Department of Computer Science and Communications Engineering, the Graduate School of Fundamental Science and Engineering of Waseda University in Partial Fulfillment of the Requirements for the Degree of Master of Engineering

Submission Date: July 22nd, 2019

Advisor: Prof. Hayato Yamana
Research guidance: Research on Parallel and Distributed Architecture

Department of Computer Science and Communications Engineering,
the Graduate School of Fundamental Science and Engineering,
Waseda University
Student ID: 5117FG19-2
Yusheng Jiang

Abstract

Privacy preservation during a search has become a serious problem in recent years. There is a need to make sure that user queries that contain sensitive private information are not abused or misused by a third party, including the search provider. One way to conduct a privacy-preserving search is to encrypt the user queries. However, traditional encryption methods are only capable of protecting user privacy during a data transfer because the query itself is decrypted at the query server. A query server is usually provided by an untrustworthy cloud provider, and the exposure of data may therefore lead to the risk of a data breach.

Fully homomorphic encryption (FHE), being capable of conducting addition and multiplication over a ciphertext, naturally provides a solution to this problem. Using FHE, the privacy of both the user queries and the database of the search provider can be protected. In this paper, we propose a privacy-preserving query system model. We implemented the proposed model on a real-world medicine side-effect query system. We applied a filtering prior to the query to reduce the size of the database and used multi-threading to accelerate the search. The system was tested 10,000 times with a random query over a database of 40,000 records of simulation data and completed 99.84% of the queries within 60 s, proving the real-world application of our system.

Contents

1 Introduction **1**

2 Preliminaries **3**
2.1 Fully homomorphic encryption (FHE) 3
2.2 FHE scheme 3
2.3 SV packing and SIMD-style calculation 4

3 Related work **5**
3.1 Private information retrieval (PIR) 5
3.2 PIR with homomorphic encryption 6

4 Proposed method **7**
4.1 System 7
4.1.1 Overview 7
4.1.2 Privacy 9
4.1.3 Procedure 9
4.2 Real-world implementation 10
4.2.1 Description 10
4.2.2 Specification 12
4.2.3 Optimization 13
4.3 Algorithm 13
4.3.1 Preparation of content-encrypted query 13
4.3.2 Filtration over query's plaintext part 14
4.3.3 Usage of SV packing 17
4.3.4 FHE calculation 18
4.3.5 Decryption and postprocessing 19
4.4 Analysis of privacy during FHE calculation 21

5 Experimental evaluation **22**
5.1 Experiment setup 22
5.2 Simulation dataset 22
5.3 Result 23
5.3.1 Overall performance 23
5.3.2 Comparison between using different number of threads 25

6 Discussion **29**
6.1 Evaluation 29
6.2 Further optimization 29

7 Conclusion **32**

A Sampled running time data in Figure 5.1 **38**

B Sampled running time data in Figure 5.4 **40**

Chapter 1

Introduction

As a quick and easy way to extract certain information from the Internet, a search process is an important Internet service. As improvements to search algorithms [9] and an upscaling of hardware features reach a turning point [8], search service providers are now turning their attention to improving the search experience of users. One important aspect of user satisfaction regarding a search is whether the private information of the users is carefully protected and only used for query-related purposes. Security issues in many global companies including Facebook [22] and Google [23] are raising awareness regarding the importance of personal information security.

Because users want to hide their query content from leaking to a third party or prevent misuse, they prefer to conduct a "search in local ," namely, they would like to access an entire database of a search provider, and choose the information they need. However, search providers hold their databases as important assets, and do not want users to have full access. Instead, they only want to show users a small portion[16]. A dilemma occurs between protecting the privacy of users and "database's privacy." Thus, as a compromise, search providers allow users to encrypt their query contents. However, this method can only protect user data from leakage during a transfer. Search providers, however, hold a secret key that still can decrypt the contents of a user query. The contents of user queries remain under the threat of leakage or misuse. A third party having access to the database of a search provider, e.g., a cloud server provider, can access the contents of a user query in decrypted form. In addition, untrustworthy search providers may use such contents in ways other than the intended query, e.g., selling the query history of the users to a third party. Currently, there exists several theoretical researches covering the area [24, 4]. However, no real-world implementation is ever carried out.

In Japan, which is a quickly aging society [11], medical care is an important topic. According to official data [26], over 3.5 million elderly people over 65 years in age receive outpatient treatment each year, and more than

0.9 million received hospitalization treatment in 2014. When patients suffer from side effects from a particular medicine, they usually ask pharmacists for help. Although referring to the attached documentation of a medicine is a way to find the potential ingredient causing a side effect, situations exist in which the attached documentation is insufficiently comprehensive. As a supplementary empirical method, pharmacists may refer to the medicinal history of other patients and find similar situations, i.e., patients with the same gender, a similar age, and who are taking at least one similar medicine and are suffering from at least one similar symptom. It is the responsibility of pharmacies to protect the medicinal history of their customers from a third party because such information is private and sensitive.

In the particular scenario related to medicine-side effect search, our goal is to develop a scheme to ensure that the queried data and full medicinal history of the patient remain safe during the entire query process. The contribution of our work is that we construct a privacy-preserving query system and we implement and evaluate its performance in consideration with the real-world scenario of medicine-side effect search. The novelty of our research lies in two points. First, we constructed the privacy-preserving query system, and we are applying the system to a real-world situation. Second, we are bringing in the research of privacy-preserving query system to the new area of medicine-side effect search.

In this paper, we first introduce basic knowledge related with fully homomorphic encryption (FHE) in Chapter 2, followed by a brief review of related studies in Chapter 3. We then describe our system using detailed algorithms in Chapter 4, and show the experiment results in Chapter 5, followed with a discussion including the evaluation and further optimization in Chapter 6.

Chapter 2

Preliminaries

2.1 Fully homomorphic encryption (FHE)

Under traditional encryption methods, decryption must be performed prior to the calculations. Direct calculations over a ciphertext are not supported. However, fully homomorphic encryption (FHE) supports an arbitrary number of additions and multiplications directly over a ciphertext.

$$\begin{aligned} FHE.Dec(FHE.Enc(a) \oplus FHE.Enc(b)) = a + b \\ FHE.Dec(FHE.Enc(a) \otimes FHE.Enc(b)) = a \times b \end{aligned} \tag{2.1}$$

In Equation (2.1), $a, b \in \mathbb{Z}$, $FHE.Dec$ and $FHE.Enc$ are methods used in FHE, and $\oplus$ and $\otimes$ are additions and multiplications over a ciphertext in FHE.

The idea of homomorphic encryption (HE) was first introduced in 1978 by Rivest et al. [15]. In 2009, Gentry introduced a practical scheme for FHE implementation on arbitrary functions and a number of operations using an ideal lattice [5]. In 2012, Brakerski, Gentry, and Vaikuntanathan introduced the BGV scheme using ring-learning with errors (RLWE) [1]. In 2010, Smart and Vercauteren introduced their SV packing technique to FHE and enabled a single instruction/multiple data (SIMD) style operation to realize a speed-up and compression [19]. IBM Research released an open-source library of HElib based on the BGV scheme, which is one of the most popular FHE libraries available [7]. Herein, we briefly introduce the FHE scheme and SV packing.

2.2 FHE scheme

In the FHE scheme, the following algorithms are included[1]:

- $FHE.SetUp(1^\lambda)$: Given a security parameter λ, output a set of parameters for encryption, $params$.

- $FHE.KeyGen(params)$: Generate a public/secret key pair, pk/sk, and evaluation key, ek.
- $FHE.Enc(pk, m)$: Given a public key pk and plaintext message m, output the corresponding ciphertext, c.
- $FHE.Dec(sk, c)$: Given a secret key sk and ciphertext c, output the corresponding plaintext message, m.
- $FHE.Eval(ek, f, (c_1, ..., c_t))$: Given evaluation key ek, an arithmetic circuit f that accepts t parameters, and t ciphertext c_1 to c_t, output a ciphertext as an encrypted calculation result, c_f.

2.3 SV packing and SIMD-style calculation

In [20], Smart and Vercauteren introduced a packing method that allows encrypting a vector of plaintext into a single ciphertext. These multiple plaintexts are stored separately in spaces called *slot*s. The authors introduced a SIMD-style operation on the packed ciphertext. In a SIMD-style FHE calculation on ciphertexts, operations such as additions and multiplications are conducted in a *slot*-wise manner. In our system, all operations over a ciphertext are applied in a SIMD manner.

Figure 2.1 illustrate an example of SIMD operation in SV packing model with FHE. In this figure, four integers $1, 2, 3, 4$ is packed into *Ciphertext 1*, and four other integers $4, 2, 3, 1$ is packed into *Ciphertext 2*, both of them are encrypted with FHE. *Ciphertext 1* and *Ciphertext 2* are both FHE ciphertexts with four *slot*s. When doing SIMD-style FHE addition and multiplication, the operation is conducted in a *slot*-wise manner, and the result is also a ciphertext with four *slot*s, respectively.

Figure 2.1: Illustration of SIMD operation in SV packing model

Chapter 3

Related work

3.1 Private information retrieval (PIR)

A private information retrieval (PIR) scheme was first introduced by Chor et al. in 1995 [2]. Kushilevitz and Ostrovsky constructed the first single-database PIR in 1997 [10]. Since then, PIR has become one of the most important cryptography schemes available [13].

Chor's PIR scheme focused on protecting the query content as well as the information inside the database. As described in Figure 3.1 an example of PIR scheme is shown. The whole process of a query is under a certain kind of encryption method, in which the query contents and the database are both encrypted. In this example, we have item indices as $1, 2, 3, 4$ and data as A, B, C, D, and we want to find out the corresponding data for index 3, which is C. PIR focused on maintaining the privacy of both sides, so the database side cannot gain knowledge of the query content, the index 3, thus 3 cannot be directly decrypted. Also, we don't want to show any other information concerning A, B, D, thus we cannot also decrypt the database. Thus, we first do SIMD subtraction of query content index with the list of items indices, which results in an encrypted list of $-2, -1, 0, 1$. An operation called "flip" is used here to change the content of this list. Any non-*zero* result will be changed into *zero*, and any *zero* will be changed into *one*. This operation can be done by decryption and pick out the *zero* part, then recalculate the result list. In some specific cases, this operation can be done without decryption. The resulting list containing only *zero*s and *one*s will conduct a SIMD multiplication with the data list. The resulting list will contain the query result information, here as C.

In addition, if we want the order of data not shown in the result, i.e., that the result C is not in position 3 in our example, we can use shuffling, rotation or FHE *totalsums* to make a change to the result.

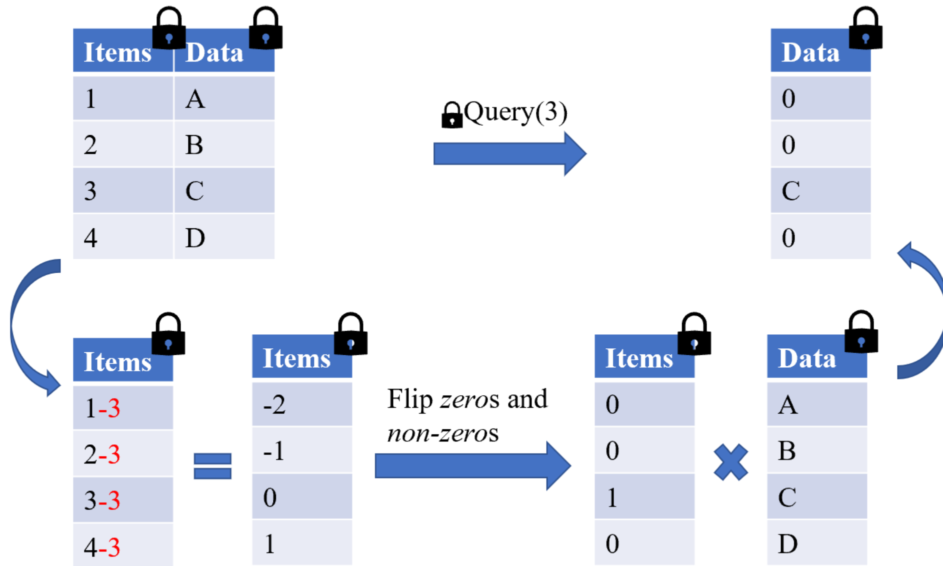


Figure 3.1: An example of the PIR scheme (exact match)

3.2 PIR with homomorphic encryption

The original work by Kushilevitz and Ostrovsky [10] has already resulted in the introduction of a PIR protocol upon homomorphic encryption. Based on Gentry's improvement of homomorphic encryption [5], the implementation of PIR based on the HE scheme has become a reality [21, 24]. A set of implementations of PIR using the HE scheme proved its usefulness in various fields including cloud computing [6], chemical compound management [16], data aggregation [14], and e-voting [17]. However, the schemes applied in these studies are not suitable for use in our system. [14] and [16] use additional homomorphic encryption (AHE), whereas[6] uses BGN homomorphic encryption and [17] uses ElGamal homomorphic encryption, which differ from FHE. These methods are effective for specific types of arithmetic circuits, but they are not suitable for our situation, in which a ciphertext and plaintext are contained inside query content at the same time. Based on[24], because FHE is theoretically suitable for any arbitrary arithmetic circuit, we consider it to be a more suitable way to solve our problem. In [4], Dong et al. introduced a general PIR scheme using FHE, although there is still space for optimization for our specific problem.

Chapter 4

Proposed method

Currently, no real-world implementation of a privacy-preserving query system on PIR using FHE scheme exists. In this research, we construct a privacy-preserving query system and implement it in a real-world scenario of medicine-side effect search. In our proposed method, the usage of filtration with inverted indexes and the usage of multithreading accelerate our system, which are novel to the PIR with FHE scheme.

In this Chapter, we will first describe the architecture of our system in Section 4.1, then we specify our real-world problem of medicine-side effect search in Section 4.2. After that, we will introduce the implementation with detailed algorithms in Section 4.3, then we will analyze the privacy during FHE calculation in Section 4.4.

4.1 System

4.1.1 Overview

For a simple description, we assume that we have only three parties in our model: the *cloud server*, the *client-side server*, and the *terminal device*.

- The *cloud server*: an untrustworthy cloud server conducting FHE calculations, provided by third-party cloud service providers, which holds the encrypted database. Its job is only used to do calculation over ciphertext, so it does not hold secret key *sk*.

- The *client-side server*: an intermediate party existing between the *terminal device* which represents an end user and the *cloud server*, which communicates with both ends. It is considered trustworthy, which connects with the *terminal device* via local network channels (e.g. WLAN). Its role is to conduct FHE encryption and decryption at local, so it holds a somewhat stronger computation power, which does not need to be higher than an ordinary desktop PC. It is considered

trustworthy with users' point of view, so it can do encryption and decryption job for query content and result. But it should not be able to gain any access to an encrypted or decrypted database.

- The *terminal device*: a party representing trustworthy user devices that send a query, e.g., tablet PCs, which is assumed to be used only to send and receive. It is assumed to have no computational power, so no calculation or encryption/decryption over FHE is conducted on this device. This party can be viewed as the front-end part of our system.

Relations between these parties are shown in Figure 4.1.

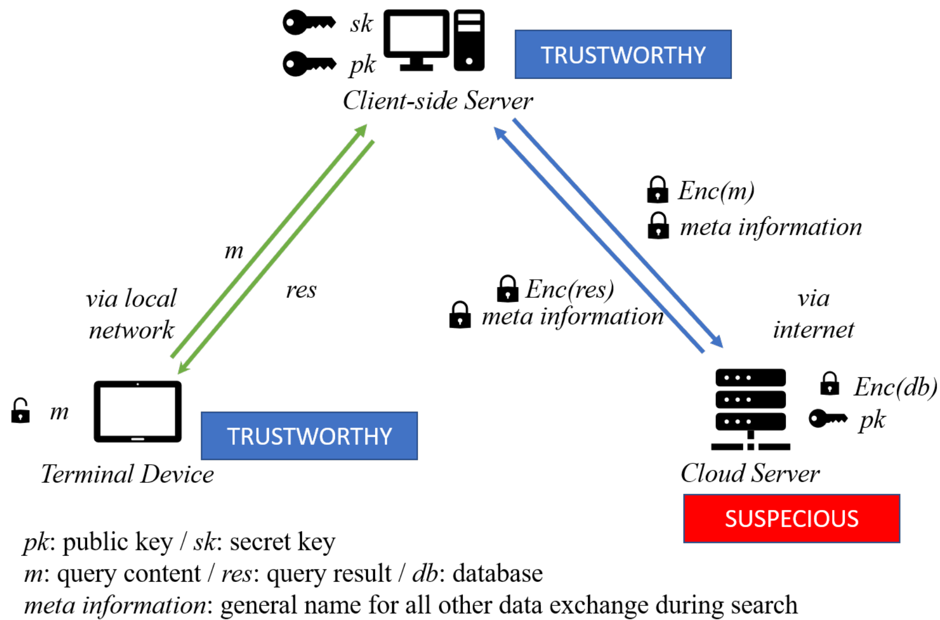


Figure 4.1: Relation between parties

As shown in figure 4.1, data transfer relations are given as follows. A user input the query content m from the *terminal device* and the content m is sent to the *client-side server*, via local network. Then m was encrypted using public key on the *client-side server*, and the encrypted *Enc(m)* is sent to the *cloud server* to perform a query. During the FHE calculation, non-privacy related data, which are represented as *meta information*, are exchanged between the *cloud server* and the *client-side server*. After the calculation is over, the encrypted result *Enc(res)* is sent to the *client-side server* for decryption. The decrypted *res* will be shown to the end user on the *terminal device*.

4.1.2 Privacy

With our system, we plan to use PIR on FHE over a privacy-related database. A key pair was previously generated and distributed to each participant, as shown in Figure 4.1. The key pair pk/sk is used for a certain time period (e.g., a month) and renewed periodically. If there are multiple client-side servers simultaneously exist, they will share the same sk.

Each party only has access to certain content. With our system, the *client-side server* has access to the key pair pk/sk and evaluation key ek; the *terminal device* has access to plaintext query content m; and *cloud server* has access to the ciphertext database $Enc(db)$, public key pk, and evaluation key ek.

Plaintext content m only exists inside the *terminal device* and the *client-side server*, and the plaintext database db is not accessible to any parties, ensuring the privacy of the user's query and database. The $Enc(db)$ will be prepared by the owner of the system or the holder of the database (aka. an authority party, who does not participate in the search scheme shown in Figure 4.1) previously and directly inputted or updated as a ciphertext.

4.1.3 Procedure

As the set-up of the system, the following preparation steps are conducted. In step 1, the parameters for the encryption environment are prepared using the *client-side server*. In step 2, the database holder prepares a fully encrypted database $Enc(db)$ and sends it to the *cloud server*.

1. step 1: The *client-side server* generates a public/secret key pair pk/sk, and evaluation key ek, and share pk and ek with the *cloud server*.

2. step 2: The *cloud server* receives the entire encrypted database $Enc(db)$ offered by the database holder using pk for encryption.

During the search process, the following steps are conducted. In step 1, the *client-side server* receives the query content from the *terminal device*. Then the *client-side server* prepares the content-encrypted query, which is also shown in Algorithm 1. In step 2, a comparison of m and db is conducted using SIMD subtraction, and *slot*s with zero as the subtraction result points to the query result. To make sure the contents of the non-zero *slot*s is not shown to the *client-side server*, the subtraction result is multiplied with a non-zero random integer r. In this way, the zero-containing *slot*s will stay in zero, while the contents of non-zero *slot*s will be changed. This process is described in Algorithm 4. In steps 3 to 5, the *client-side server* decrypts the result, finds the zeros, and reports their *index*es to the *cloud server*, whereas the *cloud server* extracts these records accordingly, as shown in Algorithm 5.

1. step 1: The *terminal device* sends the query content m to the *client-side server*. The *client-side server* encrypts the query content m using pk, and sends the content-encrypted query $Enc(m)$ to the *cloud server*.

2. step 2: The *cloud server* performs a SIMD-style subtraction between $Enc(m)$ and $Enc(db)$, and then multiply the subtraction result $Enc(m) \ominus Enc(db)$ with a non-zero random integer r. Then, the *cloud server* sends the resulting ciphertext $(Enc(m) \ominus Enc(db)) \otimes r$ to the *client-side server*.

3. step 3: The *client-side server* uses sk to decrypt the resulting ciphertext $(Enc(m) \ominus Enc(db)) \otimes r$ to gain the plaintext $(m - db) \times r$. Then, the *client-side server* searches for *slot*s with zeros, and gathers the *index*es of these slots. The *client-side server* then sends these *index*es to the *cloud server*.

4. step 4: The *cloud server* refers to $Enc(db)$ according to the *index*es, access these parts and pack them as $Enc(db')$, then, send them to the *client-side server*.

5. step 5: The *client-side server* uses sk to decrypt $Enc(db')$ to gain plaintext db', and shares them with the *terminal device*.

4.2 Real-world implementation

4.2.1 Description

Our goal is to construct and implement a privacy-preserving query system using PIR on FHE. As an example of real-world implementation of the query system, we choose the medicine-side effect query system as our experiment situation. We focus on the situation that a patient drinks several medicine(s) and suffers from several side effect(s). The patient goes to a pharmacy for consultation over how to adjust the medicine usage in order to avoid the side effect(s). In order to identify the cause of side effect(s), the pharmacist needs to browse past patients' experience for reference. The pharmacist needs to query inside a full medicinal history database, which is serious privacy-containing information. The pharmacist is especially interested in finding "similar" situations inside the database, i.e., same gender and close age with common medicine(s) and side effect(s), to help him/her make a decision. Thus, we need to use FHE here to protect the data privacy of the patient as well as other patients.

Figure 4.2 shows a flowchart that illustrates how the pharmacist judges whether or not to refer to one specific record or not. The pharmacist will first judge by comparing the gender and age difference. In Figure 4.2, we

arbitrarily set the age difference by 5. Thus, if age difference between the record and the query content is smaller or equal to 5, and the gender is the same, then the next judgement is applied, or else this record is ignored. The next judgement is by comparing the medicine list and the side effect list. If there is not at least one same medicine between the record and the query content, the record is also ignored. This also goes for the side effect's comparison. After all these judgements are performed, the record will be referred by the pharmacist.

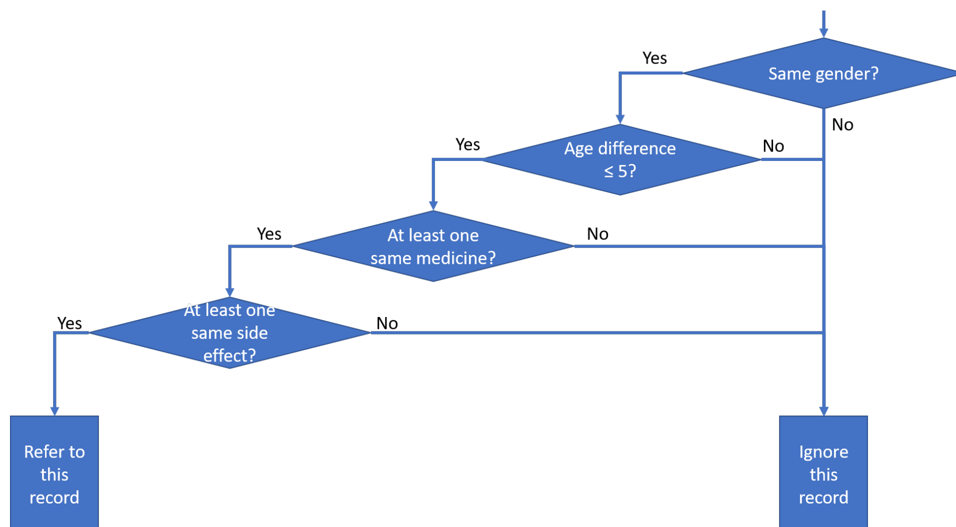


Figure 4.2: A flowchart showing the logic to decide whether to refer to a certain record or not when the allowed age difference is set as 5

Figure 4.3 shows an example dataset of the medicine-side effect query system. As shown in Figure 4.3(a), in the database, we have private information of *Alice*, *Bob*, *Cindy* and *David*, in which their age and gender are recorded, alongside with their medicinal history: the medicine ID list and the side effect ID list, and the pharmacist instructions. Note that the column "Alias" is not included in the real database, it is only used for better depiction in this paper. Figure 4.3(b) shows that the query content contains four parts: age, gender, list of medicine IDs and list of side effect IDs.

Our goal is to search "similar" cases to the query content from the database. The search criteria are the same as described in Figure 4.2. Here, we explain with an example query as shown in Figure 4.3(b): 1) Male, 2) Age 71, 3) Medicine 1 and Medicine 2, and 4) Side effect β, γ, δ. In this case, Bob' s case becomes one of the similar cases with the given query: The recorded age is within 5 years from the query content, and the recording objective is of the same gender along with the query content; The record's medicine list and side effect list both overlap with the query content's counterpart. Thus, Record ID 2 in Figure 4.3(a) will be removed of Bob's personal

information, i.e., Alias, age and gender, then shown as a query result, shown to the pharmacists as figure 4.3(c).

Record ID	Alias	Age	Gender	Medicine ID	Side Effect ID	Pharmacy Treatment Instruction
1	Alice	105	Female	[1, 2, 3, 4]	[α, β, γ, δ]	Stop 1
2	Bob	74	Male	[1]	[δ]	Drink 3
3	Cindy	6	Female	[2, 3]	[α, β]	Drink 4, Stop 2
4	David	21	Male	[4]	[γ]	Double 4

(a) Database

Age	Gender	Medicine ID	Side Effect ID
71	Male	[1, 2]	[β, γ, δ]

(b) Query content

Medicine ID	Side Effect ID	Pharmacy Treatment Instruction
[1]	[δ]	Drink 3

(c) Expected result

Figure 4.3: An example dataset of the medicine-side effect query system

The list of results will contain multiple records that meet the search criteria. This brings up a question of whether the order of the results is significant or not. If the order is important, then we need to consider on how to sort the results. As a conclusive answer to this question, the order is not significant because if the pharmacist is to make a comprehensive decision based only on the query results, then he/she will need to read all of them. Thus, we do not need to sort the list of results. In addition, because each attribute has different criteria in judging the "similarity," such as the age difference, medicine list difference and side effect list difference, it is also difficult to make a concrete judgement on how to sort. Especially, as the hit record's age information is not shown inside the query result, the ranking decision is hard to make. The comparison between results is not in the range of this thesis, and will not be discussed any further here.

4.2.2 Specification

Our goal is to provide a privacy-preserving scheme to extract information that can support pharmacists in finding the reason for the occurrence of side effects based on the medicinal history of the patient. To be specific, we need records of patients with the following conditions for a comparison of the query content:

- Gender: same
- Age: R years younger – R years older, $R \in \mathbb{Z}$
- Medicine: At least one the same
- Side Effect: At least one the same

Sensitive information includes the gender and age information of the patients, as well as other private information. Such information must be kept encrypted during the entire query process.

4.2.3 Optimization

We can perform optimization in FHE calculation for our real-world case. By using optimization, we can speed up as well as reducing memory space consumption.

Query content m contains the following four types of data: age, gender, list of medicines, and list of side effects. Here, we encrypt only age and gender to avoid privacy leakage. The lists of medicines and side effects are handled in plaintext format. Note that all data, mainly, the encrypted age, encrypted gender, and the other two plaintexts, are transferred over encrypted channels, such as AES, to keep them secured during their transfer. We separate m into two parts:

- FHE-encrypted part: age and gender
- non-FHE-encrypted part: lists of medicines and side effects

Because the lists of medicines and side effects are not encrypted by the FHE, the *cloud server* contains them in plaintext. Thus, we can apply a filtering at the *cloud server* before the FHE calculation, reducing the size of the database for calculation purposes. We apply the inverted index method commonly used in search engines to conduct the filtering step [25], as shown in Algorithm 2.

Because the *cloud server* is equipped with multiple CPU cores, we use an NTL threading pool [18] in the *cloud server* side when conducting the FHE calculations. In addition, as shown in Algorithm 3, SV packing is used.

4.3 Algorithm

In this section, we will describe our implementation in detail with five algorithms, described in five subsections. Section 4.3.1 and Algorithm 1 describes how content-encrypted query is prepared on the *client-side server*. Section 4.3.2 and Algorithm 2 describes how the encrypted database is filtered over medicine ID list and side effect ID list on the *cloud server*. Section 4.3.3 and Algorithm 3 describes how we pack multiple ciphertextsinto one ciphertext using SV packing on the *cloud server*. Section 4.3.4 and Algorithm 4 describes how we do the FHE calculation on the *cloud server*. Section 4.3.5 and Algorithm 5 describes the decryption and postprocessing steps after receiving the FHE calculation results on the *client-side server*.

4.3.1 Preparation of content-encrypted query

The *cloud server* will receive a query content from the *client-side server* prepared using Algorithm 1. Since Algorithm 1 is conducted on the *client-side server*, we already have pk which is not included in the input. This

algorithm takes in 4 plaintexts as inputs: *age*, *gender*, a list of $Med_i, i \in \{1, 2, ..n\}$ which represents medicine IDs, a list of $Side_i, i \in \{1, 2, ..m\}$ which represents side effect IDs. This algorithm outputs a tuple *query*, which contains three parts. A ciphertext c which packs up the information of *age* and *gender*, the list of medicine IDs, and the list of side effect IDs. Figure 4.4 shows an example on how we prepare the content-encrypted query.

Herein, we use m' as the grouped parameter from *gender* and *age*, combining the FHE-related parameters into only a single parameter. This is beneficial to reducing FHE calculation time complexity, as the "if"-statement is not allowed in FHE arithmetic circuit. By reducing the number of parameters, we can reduce the numbers of FHE operations, especially, the numbers of FHE multiplications in the calculation. So, it is important that we use a grouped parameter to represent *gender* and *age*.

This grouping is supported according to [27], provided that each possible pair of (*age*, *gender*) for humans will map to a unique m' within $[0, 255]$, when the upper limit of the age difference is $R \leqslant 5$. Males are mapped to $[0, 127]$, and females are mapped to $[128, 255]$. Then m' is encrypted using the *pk* which is originally stored in the *client-side server* through FHE, which results in the ciphertext c. The tuple *query* consisting of c, medicine ID list and side effect ID list is the output of this Algorithm 1.

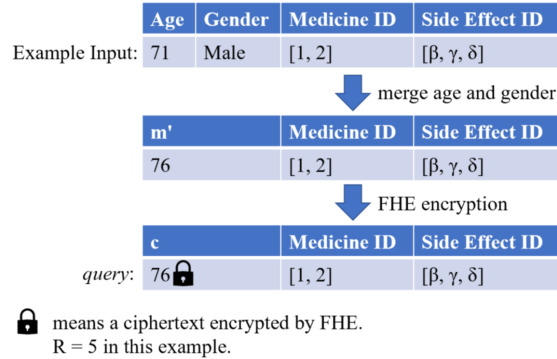


Figure 4.4: An illustration showing how we prepare the content-encrypted query

As shown in figure 4.5, the age and gender part of database *db* is also encrypted by FHE, processed by a similar rule as the query. This is mentioned in Section 4.1.3.

4.3.2 Filtration over query's plaintext part

After receiving the query contents c, $(Med_1, ..., Med_n)$, $(Side_1, ..., Side_m)$ from the *client-side server*, the *cloud server* initiates to apply filtering with an inverted index of the medicine $InvMed$ and side effect $InvSide$, and a

Algorithm 1 *QueryGen*

Input $age, gender, (Med_1, ..., Med_n), (Side_1, ..., Side_m)$
Output $query$

1: **if** $gender$ is male **then**
2: $\quad m' \leftarrow age + R$
3: **else**
4: $\quad m' \leftarrow age + 128 + R$
5: **end if**
6: $c \leftarrow FHE.Enc(pk, m')$
7: $query \leftarrow c, (Med_1, ..., Med_n), (Side_1, ..., Side_m)$

Record ID	Alias	Age	Gender	Medicine ID	Side Effect ID	Pharmacy Treatment Instruction
1	Alice	105	Female	[1, 2, 3, 4]	[α, β, γ, δ]	Stop 1
2	Bob	74	Male	[1]	[δ]	Drink 3
3	Cindy	6	Female	[2, 3]	[α, β]	Drink 4, Stop 2
4	David	21	Male	[4]	[γ]	Double 4

(a) Database *db*

Record ID	Alias	Age & gender	Medicine ID	Side Effect ID	Pharmacy Treatment Instruction
1	Alice	238	[1, 2, 3, 4]	[α, β, γ, δ]	Stop 1
2	Bob	79	[1]	[δ]	Drink 3
3	Cindy	139	[2, 3]	[α, β]	Drink 4, Stop 2
4	David	154	[4]	[γ]	Double 4

(b) Encrypted Database *Enc(db)*

Figure 4.5: An illustration showing how we prepare the encrypted database *Enc(db)*

smaller part is extracted from db for the FHE calculation. The classical pyramidal merge algorithm used in mergesort [3] will be operated over inverted indexes, making a smaller encrypted database $Enc(db')$ of size s for the FHE calculation. The filtering is shown in Algorithm 2. This algorithm is computed at the *cloud server*. This algorithm 2 receives the medicine ID list $(Med_1, ..., Med_n)$ and side effect ID list $(Side_1, ..., Side_m)$ as input, and generates a filtered encrypted database $Enc(db')$ as an output. Within this algorithm, $Intersect(List)$ takes the intersection of several sets, and $Union(List)$ takes the union of several sets.

Figure 4.6 illustrates an example of filtration with inverted indexes. In this example, we are still using the same database as Figure 4.3(a) and the same input $query$ as Figure 4.4. The inverted index by medicine ID $InvMed$ is shown in Figure 4.6(b), and the inverted index by side effect ID $InvSide$ is shown in Figure 4.6(c). As shown in Figure 4.6(b), the inverted index by medicine ID is generated previously by the *cloud server* which records the

mapping relationships from the content medicine IDs to their locations in the database. For example, medicine 1 is shown in record 1 and record 2, its corresponding inverted index will be a list $[1, 2]$.

Record ID	Alias	Age & gender	Medicine ID	Side Effect ID	Pharmacy Treatment Instruction
1	Alice	238 🔒	[1, 2, 3, 4]	[α, β, γ, δ]	Stop 1
2	Bob	79 🔒	[1]	[δ]	Drink 3
3	Cindy	139 🔒	[2, 3]	[α, β]	Drink 4, Stop 2
4	David	154 🔒	[4]	[γ]	Double 4

(a) Encrypted Database *Enc(db)*

Medicine ID	Record ID
1	[1, 2]
2	[1, 3]
3	[1, 3]
4	[4]

(b) Inverted Index by Medicine ID *InvMed*

Side Effect ID	Record ID
α	[1, 3]
β	[1, 3]
γ	[1, 4]
δ	[1, 2]

(c) Inverted Index by Side Effect ID *InvSide*

c	Medicine ID	Side Effect ID
76🔒	[1, 2]	[β, γ, δ]

(d) Content-encrypted query

ResMed	*ResSide*	*Res*
[1, 2, 3]	[1, 2, 3, 4]	[1, 2, 3]

(e) Value of parameters in Algorithm 2

Record ID	Alias	Age & gender	Medicine ID	Side Effect ID	Pharmacy Treatment Instruction
1	Alice	238 🔒	[1, 2, 3, 4]	[α, β, γ, δ]	Stop 1
2	Bob	79 🔒	[1]	[δ]	Drink 3
3	Cindy	139 🔒	[2, 3]	[α, β]	Drink 4, Stop 2

(f) filtered Encrypted Database *Enc(db')*

Figure 4.6: Illustration of an example of filtration with inverted indexes

As given by Figure 4.6(d), the content-encrypted query, contains ciphertext c, the medicine ID list $[1, 2]$ and the side effect ID list $[\beta, \gamma, \delta]$. In algorithm 2, we will first filter by medicine IDs and generate a restricted list of records $ResMed$, then filter by side effect IDs and generate a restricted list of records $ResSide$, then generate the combined list of records Res. In the end, we will refer to encrypted database $Enc(db)$ by record ID list Res, and generate the filtered database $Enc(db')$.

According to Figure 4.2, a record in the query result needs to have at least one same medicine in its medicine list as that of the query content. Thus, we take the union of the inverted indexes corresponding to the input medicine ID list $[1, 2]$ (marked by red in Figure 4.6(b)). Thus, in this example, $ResMed$ is the union of $[1, 2]$ and $[1, 3]$, which gives out $[1, 2] \cup [1, 3] = [1, 2, 3]$. The same rule applies to side effect, which results in $ResSide = [1, 3] \cup [1, 4] \cup [1, 2] = [1, 2, 3, 4]$.

As mentioned in Section 4.2.1, a record' s medicine list and side effect list should both overlap with the query content' s counterpart to be put into the query result. Thus, we take the intersection of $ResMed$ and $ResSide$,

which results in the combined list of records $Res = ResMed \cap ResSide = [1, 2, 3] \cap [1, 2, 3, 4] = [1, 2, 3]$. The values of $ResMed$, $ResSide$ and Res are shown in Figure 4.6(e).

Algorithm 2 *Filter*

Input $(Med_1, ..., Med_n), (Side_1, ..., Side_m)$
Output $Enc(db')$

1: $MergeListMed \leftarrow []$
2: **for** $i \leftarrow 1$ **to** n **do**
3: $MergeListMed.Append(InvMed[Med_i])$
4: **end for** ▷ $MergeListMed$ stores inverted indexes for inputted medicines
5: $ResMed \leftarrow Intersect(MergeListMed)$
6: $MergeListSide \leftarrow []$
7: **for** $i \leftarrow 1$ **to** m **do**
8: $MergeListSide.Append(InvSide[Side_i])$
9: **end for** ▷ $MergeListMed$ stores inverted indexes for inputted side effects
10: $ResSide \leftarrow Intersect(MergeListSide)$
11: $Res \leftarrow Union([ResMed, ResSide])$ ▷ Res is a list of record IDs
12: $Enc(db') \leftarrow []$
13: **for** i in Res **do**
14: $Enc(db').Append(Enc(db)[i])$
15: **end for**

4.3.3 Usage of SV packing

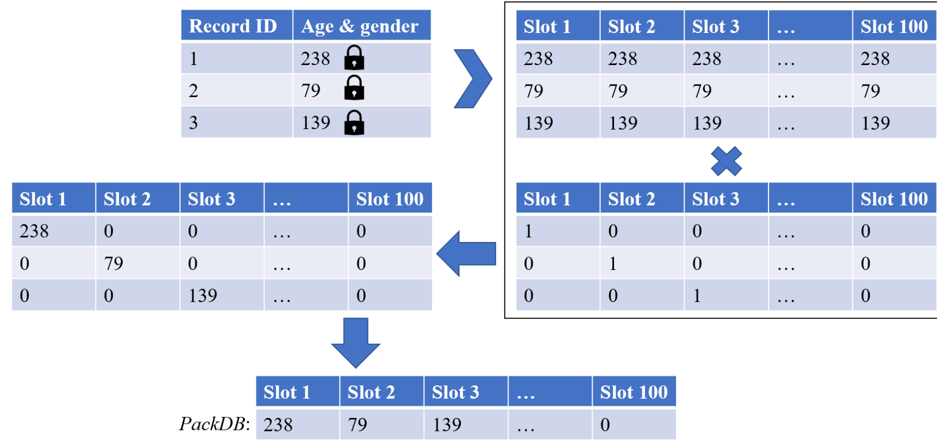
We use the SIMD calculation to speed up the FHE calculation, thus we introduce SV packing [19] to our method. An illustration showing how SV packing works for our scenario is shown in Figure 4.7 and Algorithm 3.

As shown in Figure 4.7(a), we are still using the same example database previously used in Figure 4.3. As shown in Figure 4.7(b), an FHE ciphertext contains a certain number of slots. The number of slots is predefined during the FHE parameter setup. In our example, we predefine the number of slots at 100. The result of Algorithm 2, $Enc(db')$, contains a list of ciphertexts. SV packing's goal is to pack these separated ciphertexts into a fewer number of ciphertexts.

As shown in Figure 4.7(b), in the FHE ciphertext part of a single record in $Enc(db')$, each *slot* is filled in the same value. We can SIMD multiply them each with a corresponding position indicator l, which is a vector of integers whose size equals the number of *slots*, i.e., 100. Position indicator l is a *one-hot vector* in which the position corresponding to the record ID is set to 1 and all other positions are set to 0. We can see in Figure 4.7(b),

Record ID	Alias	Age & gender	Medicine ID	Side Effect ID	Pharmacy Treatment Instruction
1	Alice	238	[1, 2, 3, 4]	[α, β, γ, δ]	Stop 1
2	Bob	79	[1]	[δ]	Drink 3
3	Cindy	139	[2, 3]	[α, β]	Drink 4, Stop 2

(a) filtered Encrypted Database *Enc(db')*



(b) Illustration of SV packing

Figure 4.7: Illustration of an example of using SV packing

after SIMD multiplication, each content is in the slot corresponding to its record ID. Then, SIMD addition is performed on these resulting ciphertexts, which results in the SV-packed encrypted database $PackDB$.

Note that in this example, the number of records does not exceed the number of *slots*. When the number of records is greater than the number of *slots*, $PackDB$ refers to a list of multiple ciphertexts.

Algorithm 3 $SVPack$

Input $Enc(db')$
Output $PackDB$

1: $PackDB \leftarrow Enc(pk, 0)$
2: **for** $i \leftarrow 1$ **to** s **do**
3: $\quad l \leftarrow$ all-zero integer list of size s
4: $\quad l[i] \leftarrow 1$ $\triangleright$ l act as a position indicator showing which *slot* to insert
5: $\quad PackDB \leftarrow PackDB \oplus l \otimes Enc(db')[i]$
6: **end for**

4.3.4 FHE calculation

We can then conduct a calculation over the ciphertext part using FHE to complete the query process in *cloud server*. The calculation is illustrated in Figure 4.8, and described by Algorithm 4.

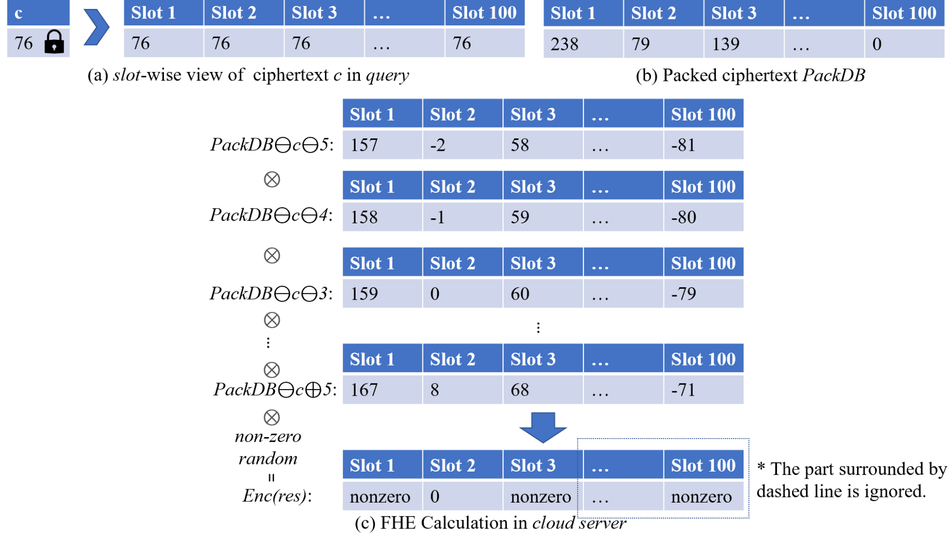


Figure 4.8: Illustration of FHE calculation in server

The matching result should have the same gender as m and at an age difference within R with the age information of m; hence, the target value should be in $[m-R, m+R]$ inside the original partial database db'. After the SIMD subtraction of $PackDB$ and c, we need to add it with $[-R, +R]$ and obtain $2R+1$ ciphertexts $MetaRes$. We can SIMD multiply these $2R+1$ ciphertexts into one ciphertext $Enc(res)$ and send it back to the *client-side server* for decryption, as shown in Algorithm 4. This algorithm takes place inside the *cloud server*. In this algorithm, the outputs of Algorithm 1, c, and Algorithm 3, $PackDB$ are inputs. R which means the allowed difference in age range is also taken as an input. The output is a ciphertext $Enc(res)$ waiting to be decrypted.

In this algorithm, first, we will generate the $MetaRes$, which is the $2R+1$ PIR calculation results for each allowed age and gender set. Because any occurrence of zero is considered as hit, we can avoid sending them all back to the *client-side server* by multiplying them up. Here we use a pyramidal way up method, i.e., multiply the neighboring two ciphertexts up inside the $MetaRes$ list until only one ciphertext remains. This is one of the best routine ways to help to reduce FHE level growth. In the end, the result is multiplied by a non-zero random integer, which turns out to be the output of the whole algorithm, $Enc(res)$.

4.3.5 Decryption and postprocessing

After receiving ciphertext $Enc(res)$, the *client-side server* will apply a decryption and request additional information, e.g., pharmacy instructions, as shown in Algorithm 5. Algorithm 5 is run by the *client-side server*. Algo-

Algorithm 4 *FHECalculate*

Input $PackDB, c, R$
Output $Enc(res)$

1: $MetaRes \leftarrow []$
2: **for** $i \leftarrow -R$ **to** $+R$ **do**
3: $MetaRes.Append(PackDB \ominus c \oplus i)$
4: **end for** ▷ $MetaRes$ stores $2R+1$ ciphertexts
5: **while** $len(MetaRes) > 1$ **do**
6: $i \leftarrow 1, j \leftarrow len(MetaRes)$
7: **while** $i < j$ **do**
8: $MetaRes[i] \leftarrow MetaRes[i] \otimes MetaRes[j]$
9: $MetaRes.Remove(MetaRes[j])$
10: $i+ = 1, j- = 1$
11: **end while**
12: **end while** ▷ Calculate product of sequences
13: $Enc(res) \leftarrow MetaRes[1] \otimes (random() + 1)$

rithm 5 will also require information from the *cloud server*. Algorithm 5 receives the result of Algorithm 4, $Enc(res)$ as input, and outputs the query result.

In Algorithm 5, $Enc(res)$ is decrypted to res. Then each *item* is checked to see whether it is *zero*. If a *zero* is found, it represents that the corresponding record should be listed in the query result. Then, the *client-side server* will request the corresponding "pharmacy treatment instructions" information from the *cloud server*. After last-step decryption, the result will be sent to the *terminal device*.

Algorithm 5 *Decrypt*

Input $Enc(res)$
Output query result

1: $res \leftarrow FHE.Dec(pk, Enc(res))$
2: **for** *item* in *res* **do**
3: **if** *item* is 0 **then**
4: request additional information related with the *index* of *item* from *cloud server*
5: decrypt the responded additional information and send it to the *terminal device*
6: **end if**
7: **end for**

4.4 Analysis of privacy during FHE calculation

We need to confirm that our system is really privacy-preserving, i.e., the privacy of the user's query and database is kept encrypted. From previous description, we have already known that m is never decrypted inside the *cloud server*, so it is safe. Now, we focus on offering proof that the privacy of the database is kept safe.

In the $metainformation$ transferred from *cloud server* to *client-side server*, a ciphertext packed from a list of non-zeros and zeros is included. An index is picked by *client-side server* after decryption. However, this does not mean that the real index or alias is shown to users. Thanks to Algorithm 2 and Algorithm 3, we are eventually making a new smaller database, which differs from situation to situation. Thus, during each time of a search, the query is eventually conducted inside a *unique* database. This procedure makes it impossible to recover the original content of database *db* from multiple times of trial query.

Chapter 5

Experimental evaluation

5.1 Experiment setup

We implemented the scheme in C++ using HElib (version: *1.0.0-beta0-release-Jan19*). We set up our *client-side server* on a desktop PC and the *cloud server* on a cloud computing platform provided by Nifty Cloud. The *client-side server* (desktop PC) was equipped with an 8-core Intel i7-8770 CPU and 16 GB of memory, with the CPU clock frequency at 3.20 GHz. The *cloud server* was equipped with 28 virtual CPUs and 256 GB of memory, which corresponds to a physical server with 28 Intel Xeon E5-2697A v4 CPUs with clock frequency at 2.60 GHz. Both ends are using Ubuntu Linux 18.04 LTS as the operating system. To make full use of the system, the maximum thread for the NTL threading pool was set as 28. The number of *slot*s is set at 100. The FHE parameters used in HElib were set as shown in Table 5.1.

Table 5.1: HElib parameters

m	p	r	L
12,097	257	1	11

5.2 Simulation dataset

We prepared the simulation dataset based on statistical data. We referred to official data offered by the Japan Ministry of Health, Labor, and Welfare [26], and by Statistics Japan [27]. From the statistical data, we gained patient distribution over the different age periods and genders. In addition, we assumed 2,000 types of medicines and 100 types of side effects, distributed in occurrence according to a Pareto distribution [12]. We arbitrarily assumed that each patient takes fewer than 20 different medicines and suffers from

fewer than five side effects. We cannot acquire the distribution data of the number of medicines a patient is actually taking simultaneously, and the distribution of the number of side effects a patient is actually suffering from simultaneously, so we arbitrarily decide to generate the number of medicines and the number of side effects uniformly, i.e., in the dataset $\frac{1}{20}$ of the patients are drinking a single kind of medicine, and $\frac{1}{20}$ of the patients are drinking 20 kinds of medicines. Though lack of support in official data here, according to Algorithm 2, increasing the number of medicines drinking eventually only results in a bigger $Enc(db')$, which sets higher obstacles for our experiment. Thus, this assumption does not make our conclusion any weaker. According to these rules, we created a dataset with a size of 40,000.

5.3 Result

We conducted 10,000 randomly generated queries between the *client-side server* and *cloud server*. The queries are generated based on the same distribution for age, gender, list of medicines, and list of side effects as we generate the simulation dataset. Each query is repeated 10 times then averaged, in order to reduce the error from change of virtual machines during query.

5.3.1 Overall performance

The relation between the filtered data size and the query processing time is shown in Figures 5.1, 5.2 and 5.3. A concrete sampled data is recorded in the appendix. The distribution of the filtering percentage, i.e., the percentage of filtered database size based on the original database size is shown in Table 5.2.

Table 5.2: Distribution of Filter Percentage

Filter Percentage	Percentage for a Total of 10,000 Attempts
0%	60.80%
0%-1%	37.45%
1%-2%	0.62%
2%-5%	0.56%
5%-10%	0.28%
10%-20%	0.17%
20%-50%	0.10%
>50%	0.02%

From Figure 5.1 and Table 5.2, we can see that filtering is effective in scaling down the database size. In 98.25% of all cases, the database was scaled down to less than 400 items, which is only 1% of the original database

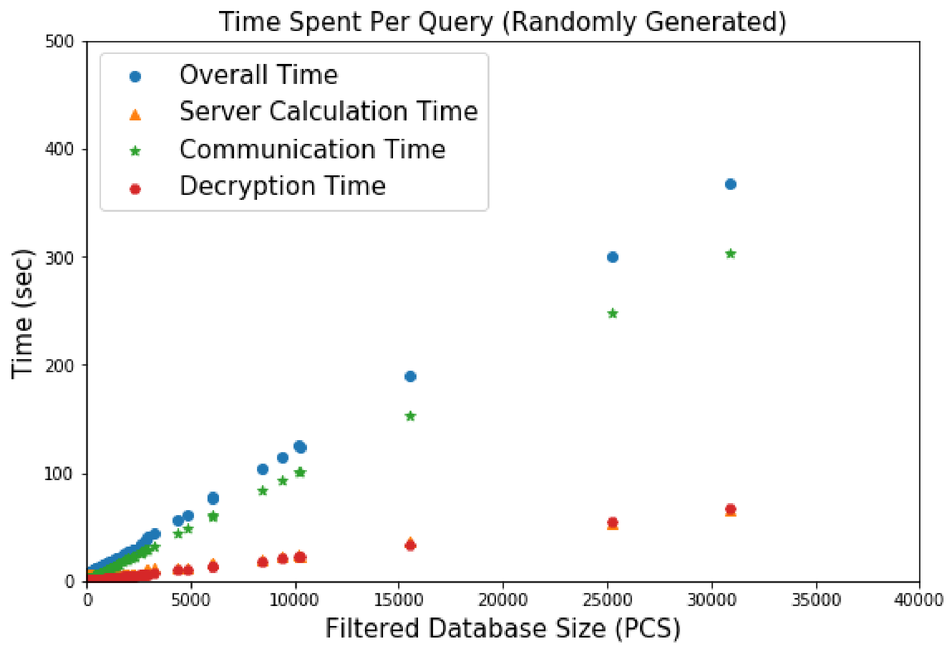


Figure 5.1: Relation between time consumption and filtered database size in 10,000 experiments

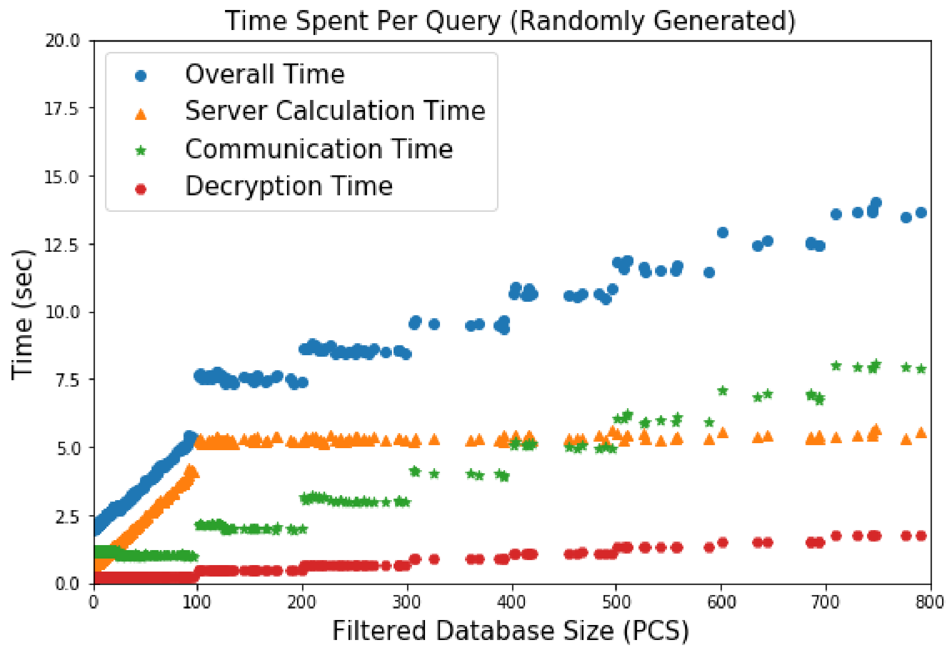


Figure 5.2: A closer view when filtered down to a dataset of 2.00%

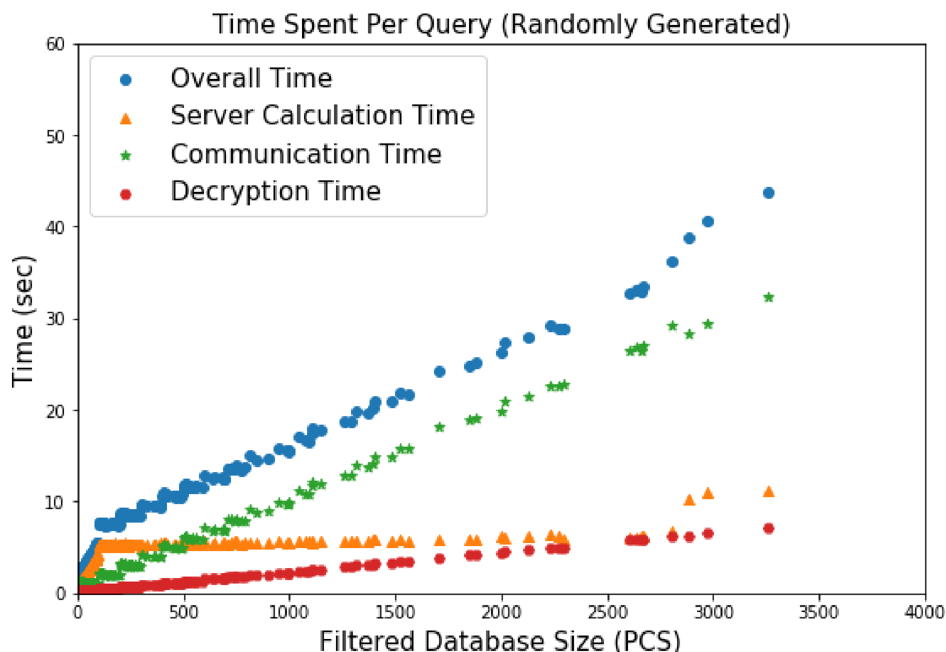


Figure 5.3: A closer view showing the “stair”-like change in FHE execution time

size. Using filtering, FHE calculations over the full database are not required, which saves time and memory.

From Figure 5.2, we can observe an improvement through multithreading. We set 100 as the *slot* number in a single ciphertext in our experiment. Thus, in Figure 5.2, the server calculation time, which was originally considered the most time-consuming, increases much more slowly after the filtered database size s reaches 100.

As a general result, 99.84% of all queries completed the entire process, i.e., from the completed collection of m at the *terminal device* to the complete output, within 60 s.

As an observation by Linux command *htop*, the maximum memory usage is approximately 2.48 GB.

5.3.2 Comparison between using different number of threads

In order to have a clearer picture of how multithreading affects the performance of FHE calculation, we conducted a second experiment. We tested the multithreading performance when setting the maximum threading numbers to 1, 7, 14 and 28, and compared the FHE calculation time. Same as the previous experiment, each query is repeated 10 times then averaged to reduce the error from change of virtual machines during query. This experiment

result is shown in Figure 5.4 and Figure 5.5.

From Figure 5.4, we can see that multithreading largely reduces the FHE calculation time in the *cloud server*. For example, when the filtered database size s is 4911, in terms of no threading in use, overall time is 236.72s with FHE calculation time at 195.82s and communication time of 43.18s; in terms of threading with maximum 28 threads, overall time is 61.74s, with FHE calculation time only at 11.33s and communication time of 50.17s. We can see that the most significant difference in time consumption lies in the FHE calculation time.

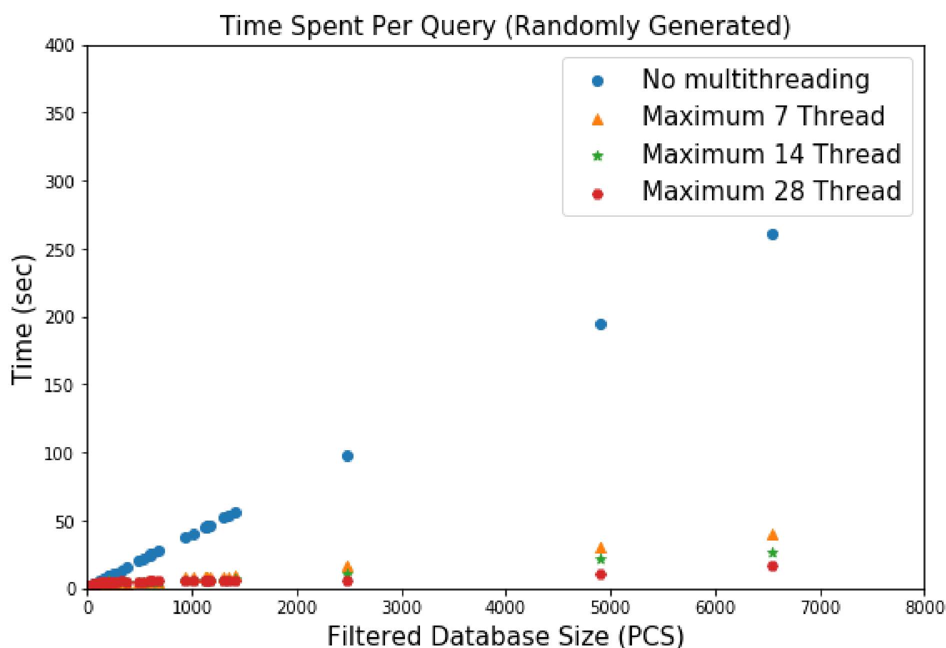


Figure 5.4: Comparison of cloud server calculation time between different maximum threading settings

Figure 5.6 is showing the comparison over acceleration ratio (FHE calculation time in the *cloud server* when single-threaded divided by when multithreaded) in FHE calculation time between different maximum threading settings. This ratio represents how many times multithreading can accelerate the FHE calculation in the *cloud server*. The comparison is between maximum 7, 14 and 28 threads.

From Figure 5.6 we can observe the difference between different maximum number of threads in use. The acceleration ratio of maximum 7 threads in use (represented by blue triangles in Figure 5.6) grows almost linear to filtered database size s when s is in 0-700. After $s > 700$, the acceleration ratio starts to stay around in a stable range around an average value of 5.90. As $700 = 7$ (number of maximum threads) $\times 100$ (number of items per chunk), we can

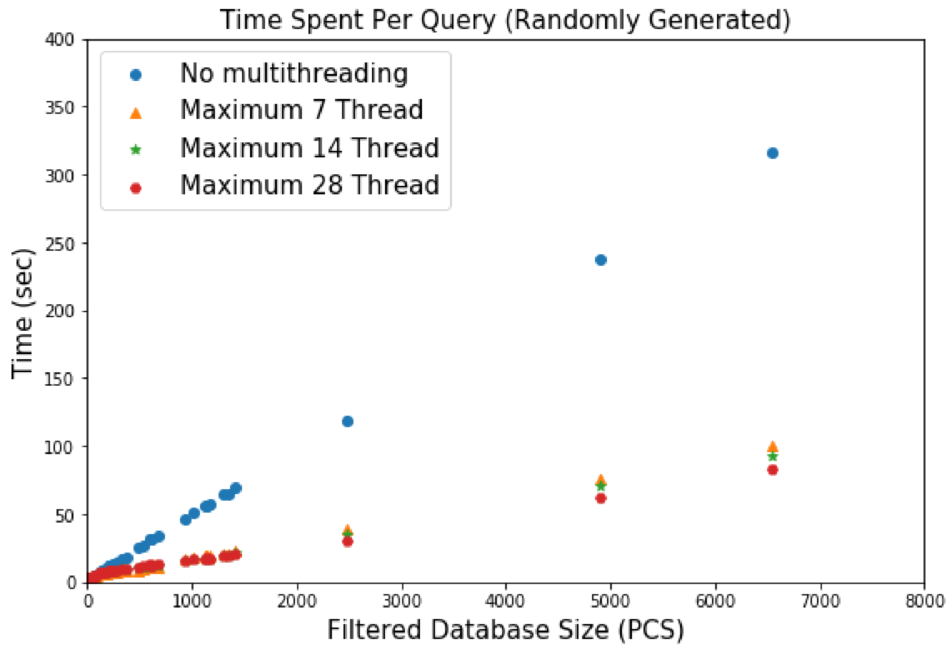


Figure 5.5: Comparison of overall time between different maximum threading settings

infer that the acceleration ratio grows until all the threads are allocated with a chunk of data, then the acceleration ratio fluctuates around a steady value.

A similar phenomenon is observed for maximum thread number at 14 and 28. The average value of acceleration ratio for maximum thread number at 14 after $s > 14 \times 100$ is 8.82. The average value of acceleration ratio for maximum thread number at 28 after $s > 28 \times 100$ is 16.15.

From this observation, we infer that the stable value of acceleration ratio is positive related to the number of maximum threads in use.

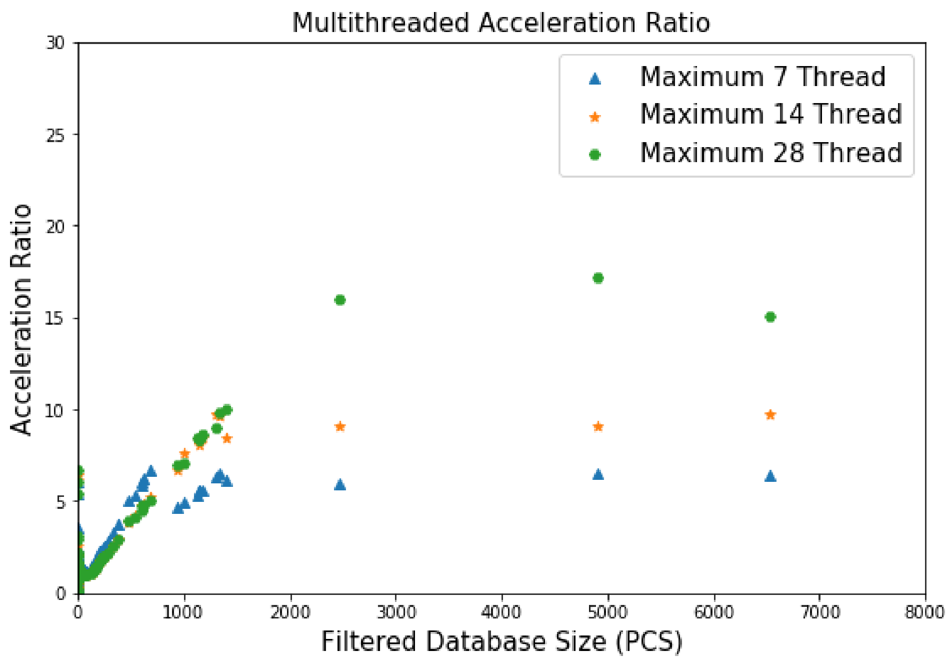


Figure 5.6: Comparison of cloud server calculation time when single-threaded divided by multithreaded time between different maximum threading settings

Chapter 6

Discussion

6.1 Evaluation

The results of our experiment conducted on a simulation dataset show that our privacy-preserving query system model is capable of conducting a search for the side effects of a medicine.

A filtering application using an inverted index and multithreading was successful in decreasing the time and memory use. From the results of Table 5.2, we can see that, in most cases, only less than 1% of the database needed to be used in the calculation. If such filtering is not applied, an FHE calculation must be performed over a full database, resulting in a waste of both time and memory. However, we must note that this emphasizing of importance on filtration is based on our assumed distribution of patients' data. Because we cannot acquire real data, we have to use simulation data instead in our experiment, which uses several arbitrary assumptions. If these assumptions turn out to be very different from real data's distribution, we cannot confirm the reliability of our result. This point needs further exploration when we can gain access to a real dataset.

From Figure 5.2, we can see that the server calculations require the greatest amount of time for a filtered database size s of less than 500. At greater than 500, the communication time, however, becomes the most time-consuming part. We introduced an NTL threading pool in the server calculations. When s is less than 100, the process is the same as with a single thread, and the server calculation time rapidly increases. When s is greater than 100, multithreading is applied, and the increase in the calculation time becomes much slower.

6.2 Further optimization

Observed from Figure 5.1, communication time takes up the largest portion in overall time. When the filtered database size s grows, the size of trans-

ferred data grows in a way linear to s. In our implementation, we focused on reducing the time for FHE calculation, but the reduction of transferred data size is not included. This brings to the problem of how to reduce the communication data size.

When we calculate the time complexity of our system, we can conclude that the following three parts are included. In the following analysis, we use s to represent the filtered database size, the same as defined in Section 4.3.

- FHE execution time. This represents the calculation time inside the *cloud server*. When observing Figure 5.2, we can see that when multithreading is not utilized, i.e., when filtered database size s is between 0-100, it escalates quickly with s goes up, in a linear manner, i.e., $O(s)$. However, after multithreading, i.e., when s is greater than 100, the shape of FHE execution time turns out to become a stair-like manner, with 2,800 items as a stair (28 (max thread number) $\times$ 100 (number of items per chunk)), i.e., $O(\frac{s}{2{,}800})$, shown in Figure 5.3.

- Communication time. In our consideration, the bandwidth and transfer speed will stay unchanged during the process. It depends on the size of $Enc(m)$, *meta information* and $Enc(res)$. The size of $metainformation$ contributes most to the packet amount, which is directly linear to s. Thus, we can also consider that communication time as $O(s)$.

- Decryption time. It depends on the size of $Enc(res)$, which, to some extent grows as s grows larger. This part cannot be properly predicted; however, it is small enough to ignore.

Thus, we can consider that the overall time complexity should be $O(\alpha\frac{s}{2{,}800}+\beta s)$. Herein we use α and β as a representation of coefficients in regard to FHE execution time and communication time, as they differ in time scales. Basically, the overall time complexity is still in a linear relation with s, thus eventually an $O(s)$ linear time complexity.

In our observation, $\beta \gg \frac{\alpha}{2{,}800}$ holds. This means that the time complexity in referring to communication time is taking up a greater portion, and this influence is shown in Figure 5.1 where s grows large enough. As the communication time mainly depends on the transferred packet size and bandwidth, we can consider solutions from two aspects. The first is to reduce the packet size, but this is very difficult. A possible way is to consider using compression like *zip*, however, we still need to consider that the time loss for compression and decompression. Another possible way lies in that we should find another way to express the status, i.e., find another ciphertext expression for gender and age information, in which case we may possibly make an easy "flip" of the result, so that we can avoid transferring big packets of *meta information* data eventually. However, this needs further investigation in its feasibility in terms of mathematics. What is more, to reduce this

heavy time consumption, we can consider packing more slots into a single ciphertext. However, this requires that we adjust the parameters to obtain a larger packing capacity. The second is to solve the problem from physical parts, e.g., expanding the bandwidth.

Chapter 7

Conclusion

In this research, we proposed a scheme for a privacy-preserving query system, and implemented it into a real-world case. The novelty of our research is that we made the first real-world implementation of an FHE-using privacy-preserving query system, which is also the first real-world implementation of PIR with the FHE scheme. This system can be useful in helping pharmacists make a decision during treatment, while protecting the patients' as well as the database's privacy. This system can conduct a query inside a 40,000-items scale FHE encrypted database within 60 seconds for 99.84% of all cases. In 98.85% of all cases, we can even reach within 12.5 seconds of overall running time per query session. From this observation, we can confirm that our FHE-using privacy preserving query system is efficient in time consumption.

Thanks to our collaborative work with *Meiji Pharmaceutical University*, this system is estimated to become a real open-source product and will be experimented in real-world pharmacies with real data. This is the first real-world implementation of such a system using PIR over FHE, and we confirmed the usefulness of the scheme through experiments conducted on simulation datasets. In addition, this first implementation can bring further research into transplanting this scheme in other various fields engaging privacy protection. We can estimate that fully homomorphic encryption will be widely used in many cases around the world, in the future.

We still have problems left to be solved for this area of research. The first problem is we need further proof of the reliability of our result, because we are using a simulation dataset. To solve this problem, we need to gain access to a real-world dataset.

Another problem lies in that we need the cut down on communication cost, as this part cannot be accelerated by multithreading. To solve this problem, we come up with two possible ways. One is to reduce the number of packets necessary for transferring. We can use *zip* or other encoding ways to help reduce this information amount. Another way is to physically enlarge

the number of packets that can be sent at a time, i.e., broaden the bandwidth. However, both ways need further investigation. We will continuously explore other possible ways to solving these problems.

Acknowledgement

This work was supported by JST CREST, Grant Number JPMJCR1503, Japan.

I want to specially acknowledge Prof. Yamana for leading my way to the field of computer science and the field of cryptography. His modest attitude on formatting and referencing improves my academic reading and writing greatly. He also offered the usage of a cloud server from Nifty Cloud so that we can conduct the experiment directly.

I want to grant my thanks to Yu Ishimaki, Hiroki Sato, Yoshiko Yasumura, Takuya Suzuki, Qiuyi Lyu and Arisa Tajima for their help in my research. They helped me in solving many of my puzzles and taught me very much in coding using HElib.

I want to thank Prof. Noguchi and Prof. Kanno from Meiji Pharmaceutical University who offered the chance of collaborative research, so that we can use and refer to their data collected from medicinal field. They also give me instructions on the real-world implementation of this system.

Bibliography

[1] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 309–325. ACM, 2012.

[2] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *Proceedings of the IEEE 36th Annual Foundations of Computer Science*, pages 41–50. IEEE, 1995.

[3] R. Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, 1988.

[4] C. Dong and L. Chen. A fast single server private information retrieval protocol with low communication cost. In *Proceedings of the European Symposium on Research in Computer Security*, pages 380–399. Springer, 2014.

[5] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st annual ACM symposium on Symposium on theory of computing*, pages 169–169. ACM Press, 2009.

[6] Y. HaiBin and Z. Ling. A secure private information retrieval in cloud environment. In *Proceedings of the 2016 International Conference on Intelligent Networking and Collaborative Systems (INCoS)*, pages 388–391. IEEE, 2016.

[7] S. Halevi and V. Shoup. Algorithms in HElib. In *Proceedings of the 34rd Annual International Cryptology Conference (CRYPTO 2014)*, pages 554–571. Springer Verlag, 2014.

[8] H. N. Khan, D. A. Hounshell, and E. R. Fuchs. Science and research policy at the end of moore' s law. *Nature Electronics*, 1(1):14–21, 2018.

[9] L. Kotthoff. Algorithm selection for combinatorial search problems: A survey. In *Data Mining and Constraint Programming*, pages 149–190. Springer, 2016.

[10] E. Kushilevitz and R. Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, pages 364–373. IEEE, 1997.

[11] N. Muramatsu and H. Akiyama. Japan: super-aging society preparing for the future. *The Gerontologist*, 51(4):425–432, 2011.

[12] T. I. Oprea. Property distribution of drug-related chemical databases. *Journal of computer-aided molecular design*, 14(3):251–264, 2000.

[13] R. Ostrovsky and W. E. Skeith. A survey of single-database private information retrieval: Techniques and applications. In *Proceedings of the International Workshop on Public Key Cryptography*, pages 393–411. Springer, 2007.

[14] T. D. Ramotsoela et al. *Data aggregation using homomorphic encryption in wireless sensor networks.* PhD thesis, University of Pretoria, 2015.

[15] R. L. Rivest, L. Adleman, and M. L. Dertouzos. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.

[16] K. Shimizu, K. Nuida, H. Arai, S. Mitsunari, N. Attrapadung, M. Hamada, K. Tsuda, T. Hirokawa, J. Sakuma, G. Hanaoka, et al. Privacy-preserving search for chemical compound databases. *BMC bioinformatics*, 16(18):S6, 2015.

[17] S. S. Shinde, S. Shukla, and D. Chitre. Secure e-voting using homomorphic technology. *International Journal of Emerging Technology and Advanced Engineering*, 3(8):203–206, 2013.

[18] V. Shoup. NTL: A library for doing number theory. *http://www.shoup.net/ntl/*, 2001.

[19] N. P. Smart and F. Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In *Proceedings of the International Workshop on Public Key Cryptography*, pages 420–443. Springer, 2010.

[20] N. P. Smart and F. Vercauteren. Fully homomorphic SIMD operations. *Designs, codes and cryptography*, 71(1):57–81, 2014.

[21] V. Vaikuntanathan. Computing blindfolded: New developments in fully homomorphic encryption. In *Proceedings of the 2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, pages 5–16. IEEE, 2011.

[22] P. Waterfield and T. Revell. Huge new facebook data leak exposed intimate details of 3M users. New Scientist, URL: https://www.newscientist.com/article/mg23831782-100-huge-new-facebook-data-leak-exposed-intimate-details-of-3m-users, May 2018. Accessed: 2019-07-17.

[23] J. C. Wong and O. Solon. Google to shut down Google+ after failing to disclose user data leak. The Guardian, URL: https://www.theguardian.com/technology/2018/oct/08/google-plus-security-breach-wall-street-journal, Oct 2018. Accessed: 2019-07-17.

[24] X. Yi, M. G. Kaosar, R. Paulet, and E. Bertino. Single-database private information retrieval from fully homomorphic encryption. *IEEE Transactions on Knowledge and Data Engineering*, 25(5):1125–1134, 2013.

[25] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems (TODS)*, 23(4):453–490, 1998.

[26] 大臣官房統計情報部. 平成 26 年 (2014) 患者調査の概況. *https://www.mhlw.go.jp/toukei/saikin/hw/kanja/14/*, Dec 2015.

[27] 総務省. *日本の統計: 2018*. 日本統計協会, 2018.

Appendix A

Sampled running time data in Figure 5.1

Table A.1 shows a selected portion from the running time data. All time is marked in seconds, with two digits after decimal point. The "percentage" column states how many queries is surpassed when sorting by corresponding filtered database size s.

Table A.1: Sample of running time data (seconds)

Filtered database size s	Overall time	FHE calculation time	Communication time	Decryption time	Percentage
0	0.30	0.00	0.05	0.00	>0.00%
1	1.99	0.61	1.16	0.25	>60.80%
5	2.17	0.76	1.18	0.24	>81.91%
10	2.35	0.96	1.16	0.25	>86.16%
20	2.74	1.32	1.19	0.24	>90.03%
40	3.27	2.02	1.02	0.24	>93.46%
60	3.99	2.73	1.03	0.24	>95.02%
81	4.83	3.55	1.04	0.24	>95.92%
101	7.62	5.24	2.14	0.46	>96.74%
250	8.44	5.24	2.98	0.67	>97.49%
501	11.80	5.48	6.07	1.32	>98.59%
686	12.57	5.41	6.93	1.54	>98.84%
693	12.43	5.33	6.87	1.53	>98.85%
1000	15.61	5.51	9.87	2.18	>99.11%
1560	21.64	5.68	15.74	3.49	>99.40%
2609	32.76	6.11	26.42	5.84	>99.56%
3578	52.66	12.66	39.78	8.22	>99.74%
4406	55.81	11.61	43.98	9.71	>99.83%
4837	60.56	11.94	48.38	10.60	>99.84%
6049	77.29	16.80	60.26	13.22	>99.86%
8442	103.58	19.31	84.04	18.37	>99.88%
9412	115.12	22.21	92.67	20.53	>99.93%
15513	189.95	36.01	153.71	33.63	>99.97%
25206	300.79	52.69	247.85	54.56	>99.98%
30900	368.29	65.29	302.77	66.62	>99.99%

Appendix B

Sampled running time data in Figure 5.4

Table B.1 shows a selected portion from the comparison between maximum number of threads. All time is marked in seconds, with two digits after decimal point. "Acceleration ratio" is calculated by single-threaded FHE execution time divided by multithreaded FHE execution time.

Table B.1: Sample of running time data (seconds) in comparison of maximum number of threads

Filtered database size s	Single threaded time	7 threads	(Acceleration ratio)	14 threads	(Acceleration ratio)	28 threads	(Acceleration ratio)
1	0.85	0.52	1.63	0.58	1.45	0.60	1.42
10	1.15	0.79	1.45	0.92	1.23	0.94	1.21
85	3.47	2.89	1.20	3.59	0.96	3.61	0.96
183	7.32	4.01	1.82	5.13	1.42	5.15	1.41
207	9.02	4.02	2.24	5.31	1.69	5.17	1.74
376	15.11	4.02	3.75	5.16	2.93	5.20	2.90
484	20.41	4.05	5.04	5.25	3.89	5.20	3.92
540	21.73	4.06	5.35	5.20	4.18	5.25	4.14
688	27.50	4.12	6.67	5.27	5.22	5.50	5.00
937	37.87	8.06	4.70	5.68	6.67	5.45	6.95
1012	40.69	8.17	4.98	5.33	7.63	5.74	7.09
1179	46.70	8.39	5.57	5.53	8.45	5.41	8.64
1345	53.55	8.27	6.47	5.55	9.64	5.46	9.81
1410	56.61	9.27	6.10	6.71	8.44	5.64	10.03
2476	97.88	16.39	5.97	10.78	9.08	6.14	15.94
4911	194.41	29.94	6.49	21.35	9.11	11.33	17.15
6539	260.67	40.73	6.40	26.80	9.73	17.34	15.03