Somewhat Homomorphic Encryption Scheme for Secure Range Query Process in a Cloud Environment

BY

Shaobo Wei

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science (M.Sc.) in Computational Sciences

The Faculty of Graduate Studies

Laurentian University

Sudbury, Ontario, Canada

Abstract

With the development of the cloud computing, recently, many service models have appeared which are based on the cloud computing, such as infrastructure as a service (IaaS), platform as a service (PaaS), software as a service (SaaS), and database as a service (DaaS). For DaaS, there exist many security issues. Especially, the database as a service cannot be fully secured because of some security problems. This research area of cloud computing is called as cloud security. One of the problems is that it is difficult to execute queries on encrypted data in cloud database without any information leakage. This thesis proposes a secure range query process which is based on a somewhat homomorphic encryption scheme to improve secure database functionalities. There is no sensitive information leakage in the secure range query process. The data that are stored in the cloud database are the integers which are encrypted with their binary forms by bits. A homomorphic “greater-than” algorithm is used in the process to compare two integers. Efficiency, security, and the maximum noise that can be controlled in the process are covered in the security and efficiency analysis. Parameter setting analysis of the process will also be discussed. Results of the proposed method have been analyzed through some experiments to test the secure range query process for its practicability with some relatively practical parameter settings.

Keywords

DaaS, Cloud Security, Somewhat Homomorphic Encryption, Secure Range query

Acknowledgements

I would like to acknowledge my supervisor Dr. Kalpdrum Passi. He worked together with me on my thesis. I found the research area, topic, and problem with his suggestions. He guided me with my study, and supplied me many research papers and academic resources in this area. He arranged a lot of meetings with me to discuss about my thesis and give me much helpful advice. When I wanted to execute some experiments, he also supplied me a SHARCNET account to let me use the computing ability of that website to finish my experiments. I cannot finish this thesis without his help.

Table of Contents

List of Tables

List of Figures

Abbreviations

DaaS	Database as a Service
FHE	Fully Homomorphic Encryption
SHE	Somewhat Homomorphic Encryption
SRQ	Secure Range Queries
PPRQ	Privacy-Preserving Range Queries
SBD	Secure Bit Decomposition
SC	Secure Comparison

Chapter 1

Introduction

1 Introduction

With the development of the times, human beings' activities are much more relying on computing ability. These days, the word "Cloud Computing" is frequently being mentioned in the area of computational science. Cloud computing is based on distributed computing technology which can supply much more powerful computing ability. There are many advantages of cloud computing that are mentioned in [1, 2]. A very important advantage is flexibility of cloud computing. The cloud service provider can establish a cloud with one or more kinds of services in the local domain and supply different, customizable, and flexible services to the users at the far-end. Except for computing ability, cloud also can supply many other kinds of resources as services, such as storage, software, platforms, and even information technology infrastructures. From this, many cloud service models are summarized, such as Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS). The users can use directly these services from different functional clouds which are supplied by the cloud providers. These service models which are based on cloud computing can reduce drastically the costs of information technology for the users. Also, due to flexibility of cloud computing, the waste of computing resources is much more reduced, too. The high utilization rate of information technology resources is enabled through cloud computing, that is mentioned in [1, 2, 3].

According to different positions that cloud is located in, clouds can be classified into three categories, which are public cloud, private cloud, and hybrid cloud. Private cloud is a kind of cloud which is established by the organization itself in its local domain, and used by the members inside the organization. Since private cloud is established inside the organization, this kind of cloud has relatively high security. However the costs of this kind of cloud are relatively high to the organization. In this thesis, the clouds, which is discussed, means public clouds. The features which are mentioned above are also all about public cloud which have more details in [4]. Public cloud is provided by the cloud provider which contains many kinds of functional services. The IT infrastructures of public cloud are set up on the side of the cloud provider, so the users just need to buy the usage rights of those services. This means the users' organizations do not need to establish any IT infrastructures locally to use those IT resources, and reduce the IT costs. There are some advantages are pointed in [1, 4], such like that public cloud has great elasticity, and high IT resource utilization rate. The last kind of cloud is called as hybrid cloud which combines the features of private cloud and public cloud.

The applications of cloud computing are also widely used in business world. The prospect of cloud computing in business is also anticipated. Many companies and enterprises all have the demands to use cloud services. However, if they want to apply for cloud services veritably and comprehensively into the aspects of business system, there are still many practical problems that need to be solved [1, 2, 3, 5]. There are three different business models of cloud computing from [1, 3, 5], which can be summarized as infrastructures in the cloud, platforms in the cloud, and applications in the cloud. Different service models and different users' requirements should work with different business models. This still is a problem which needs to be studied. Cloud computing is also playing a very important role in academic world. The probability of using cloud computing for science is discussed in [6]. Through IaaS, researchers can execute the experiments which need a lot of computing ability. The cloud can provide this computing ability by providing the function of its infrastructures. Also, through SaaS, researchers can use some large software from the cloud without installing it on their local machines.

1.1 Cloud Database

This thesis will focus on one kind of service model, which is the cloud database service that is provided by the cloud service provider that is also called as Database as a Service (DaaS). There are two main models of the cloud database services which are provided by the cloud provider. The first one is that the cloud service provider supplies the platforms and virtual machines to the users to run their own databases. The second one is that the users can directly use the databases which are provided by the cloud service provider. Therefore, the users can store, manage, and access their data on the side of the cloud to enable the functionalities of a database system without installing a database locally. There are some features of cloud database service from [7, 8], such as multi-tenancy, scalability, and encrypted data. Generally speaking, the users can set their databases with a web-based console, such as setting security provisions and configuring the databases. For enabling the management process of the cloud database for the user, the cloud provider need to design an application programming interface (API) to enable the users to manage and control their data in the cloud database. The cloud provider need to ensure all the software which is used by the cloud database is visible to the user, including the operating system, the database software, and the third party software for database accessing and management. Therefore, the user can ensure there is no security and privacy threat. The cloud database provider should also do the maintenance and upgrades to the software to ensure proper running and performance of the database. The scalability and high availability of the database are also the features which the cloud provider should ensure. There are three main problems that are unsolved [7]. The first one is the security and privacy problem. The second one is the performance problem. Since the communication of the information between the user and the cloud database, and the use of the database are all based on the network, the efficiency of a cloud database service depends on the efficiency of the network data transmission between the user and the cloud provider. The third is the operating interface of the cloud database. The cloud provider should design an easy-to-use and powerful operating interface for the user to use. Although DaaS has many advantages, there still is a very important functionality of normal databases that it cannot enable completely, which is query process. It is known that enabling complete query process in a cloud database is still a challenging problem for the researchers in this area [9]. The difficult part is how to enable query process without any information leakage, in other words, how to run queries on completely encrypted data.

Nowadays, there exit many cloud service providers in this industry such as Amazon EC2, Microsoft Azure, Google Cloud, IBM DB2, and so on. The cloud database services that are provided by these cloud providers have their own features [7, 10]. Users need to choose suitable services from a suitable cloud provider. There are three main levels for the basic architecture of the cloud services [7, 10, 11]. They are central services, service management, and user accessing interface. Central services level is to treat infrastructures, platforms, and software as services. Central services should have reliability, high availability, and flexibility to satisfy different requirements of the users. Service management level is to support central services level to ensure the abilities that central services should have and the security. User accessing interface is to ensure the user can access the database on the cloud. Performance is also an indicator to evaluate cloud database services of the cloud providers. In [12], the researchers evaluate the performance of a cloud database service which is based on Amazon EC2. They find that the performance of EC2 cloud computing services are still not efficient for scientific computing with large data, but they still can provide temporary computing resources for the scientists.

1.2 Cloud Security

This thesis will focus on a very important sub domain of cloud computing area which is called as cloud security. These days, this cloud security area has become a very hot research area. Since cloud security problems may affect the scope and level of the applications of cloud computing, the research in this area can benefit the development of practical applications of cloud services.

According to the different objectives of the cloud security issues, these issues can be classified into two categories. They are the issues which are faced by the cloud service providers, and the issues which are faced by the cloud service users. Some cloud security problems faced by the cloud service providers and cloud service users are as follows [13, 14, 15, 16]:

1. User Privilege Control – In the process of the cloud services, the cloud service provider needs to control and authenticate the user's privileges for users from different organizations as well as the users in the same organization but with different privileges;
2. Information Leakage Prevention – In a cloud service process, the information leakage is mainly to three parties. The first one is that the information is leaked to a third party except for the cloud service provider and the cloud service user. The

second one is that the information is leaked to the staff inside the cloud service provider but who do not have the privilege to access this information. The third one is that the information is leaked to the user who is in the authorized organization but without the privilege to access this information;

3. Data Loss Prevention and Recovery – The cloud service provider should ensure that the data of the users will not be damaged due to some physical or technical reasons, and cannot be accessed any more. When the data loss happens for real, the cloud service provider should implement some methods to recover it;
4. Known Attacks Protection – In the process of the cloud services, the cloud provider should have safeguard procedures to deal with the known attacks from the outside to ensure the security of the users' data;
5. Long-Term Availability – A cloud service, especially the cloud database service, should have long-term availability, because the time of the first-time data outsourcing could be long, and the process needs to involve some professionals;
6. Complete Database Functionalities – Since most companies and enterprises encrypt their data before outsourcing to the cloud due to some requirements of privacy and confidentiality, some database functionalities cannot be realized on encrypted data without any information leakage;
7. Building Reputation of Trust for Cloud Providers – This is a relatively new problem. The cloud service provider should make the users to believe that their

cloud services are reliable because of their security and technology, and then the user will choose the cloud services from this cloud provider.

This thesis addresses how to realize secure query process with encrypted data. Query functionalities are very important for a database. They can directly affect the performance of a database. As mentioned above, the data which is stored in the cloud is encrypted, so normal data query process cannot be realized on encrypted data. Only secure data query process can be realized. However, there could be information leakage in the process of secure queries [17, 18]. It is very important to distinguish what information is usable, and can threaten the security of the cloud database and the privacy of the user, and what information is worthless. There should be no usable information leakage in the applications. In [17], the researchers proposed a method that let the users with different authentication levels access the data with different sensitive levels without violating the security of the database. In [18], the researchers proposed a secure data query framework which is based on an information dispersal algorithm to ensure the security and efficiency in the process of secure data queries. This thesis addresses a secure range query process which is used a lot in practical applications.

1.3 Problem Definition

The central content of this thesis is to use a somewhat homomorphic encryption scheme to enable a secure range query process in a cloud environment. As shown in Figure 1, a cloud service environment consists of three parties, which are the data owner, the users, and the cloud service provider. The data owner can be an organization, company, or enterprise. First, the data owner encrypts the data that he wants to outsource to the cloud, and then sends the data and some related files to the cloud. These related files can be the public key, and the authorizing files for the users. This process is called as "data outsourcing". When a user in the organization wants to access some data in the cloud database, first he needs to send a request message to the cloud, and then, the cloud has to verify the authority of this user to see if he has the right to access the data that he is requesting. If the user is authorized, the request will be executed. Otherwise, it will be sent back with a rejecting message.

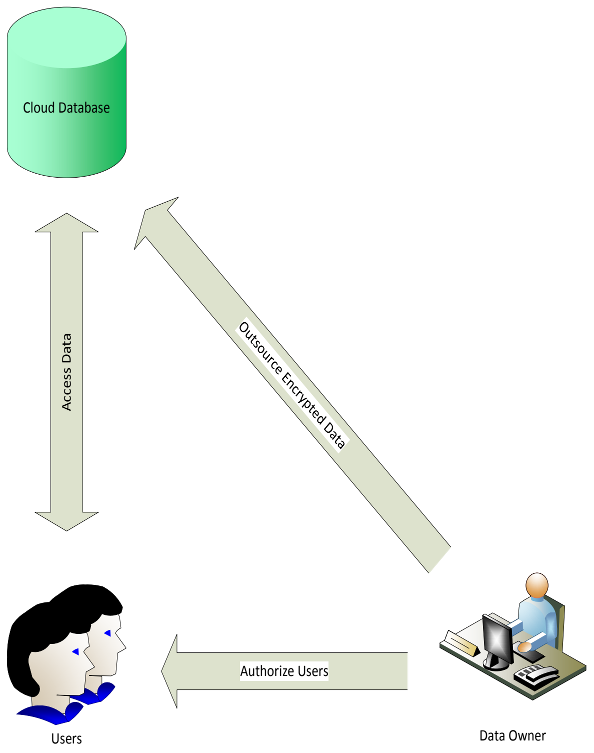

Figure 1: Cloud Environment Model

The request message contains the range query request and the user's authorization information. The range query request in the request message should have two parts. The first part is a set which contains the indexes of the attributes in the database that the user wants to query. Denote this set as H. The second part is the lower and upper boundaries of the range in the query. Denote the lower boundary as L, and upper boundary as U. If after the verification by the cloud, the user has the authority to access the data in the query, the cloud should send all the data in the range (L, U) with the attributes in H as the results of the query. The reason that this process is called “secure” range query is that there is no useful information leakage in the process.

1.4 Contributions

The main contribution of this thesis is that a somewhat homomorphic encryption is used to enable a secure range query process, since there is a previous study that realizes a secure range query process by involving another cloud service provider into the cloud environment. This may lead some unknown security problems because that secure range query process is under an assumption that the collusion between the two cloud service providers is negligible. In the secure range query process in this thesis, there is only one cloud that is involved in the process, since the process is based on a somewhat homomorphic encryption scheme, and the other process is based on some multi-party computation method.

Another contribution of this thesis is that some parameters are analyzed that are involved in the secure range query process. These parameters are the security parameter, the default range for generating the secret key, and the number of the fake bits that are appended behind the encrypted binary bits of the integers. These parameters can affect the efficiency, security, and the maximum noise that can be controlled in the process. The analysis is about how these parameters exactly affect the secure range query process. Two parameter settings are found that can handle the 2^{20} and 2^{64} integers to fit the needs of the practical applications. A way is proposed that changing the default range for generating the secret key to increase the maximum noise that can be controlled in the process for handling larger integers in the process.

1.5 Outline

The rest of the thesis contains the following contents:

Chapter 2 shows some previous works about homomorphic encryption schemes, and some secure database query processes. Both of them are closely related to the secure range query process.

Chapter 3 introduces some methods that are used in the secure range query process. It includes the somewhat homomorphic encryption scheme that is used in the process, some basic homomorphic operations algorithms, and the homomorphic greater-than algorithm that is built based on those basic operators.

Chapter 4 presents the whole secure range query process, the security analysis, and the parameter setting analysis.

Chapter 5 compares the secure range query process to the privacy-preserving range query process from [35], and shows the results of the experiments and their analysis.

Chapter 6 is about the conclusion of the thesis and the future work.

Chapter 2

Literature Review

2 Related Work

This section introduces some research achievements that are in cloud security. They are all requisite foundational methods for the secure range query processing problem. Homomorphic encryption schemes are the most important part for the realization of a secure range query process. There also are some methods and problems that are associated with the realization of the secure database functionalities, such as the famous millionaires' problem, and the secure comparison algorithms with encrypted data.

2.1 Homomorphic Encryption Schemes

First of all, the homomorphic encryption schemes will be introduced. Secure range query (SRQ) process is used to execute the range query on encrypted data, and there is no information leakage in the process. Since the process is implemented on encrypted data, the features of the encryption scheme that is used, at the first place, to encrypt the data has a strong influence on what methods should be used to realize the secure range query process, and the performance of the secure range query process. Another important requirement is that the encryption scheme which is used at the first place must have homomorphic properties.

Homomorphic properties mean that if an operation is applied on several encrypted values (ciphertext), and the decryption process is applied on the result, the plaintext that is obtained should equal the result of the same operation that is applied on the corresponding values without encryption (plaintext). Through the homomorphic properties, some operations can be applied on the encrypted values as can be done on the corresponding values without encryption to obtain the same results. This kind of encryption schemes has a great possibility to complete the cloud database functionalities. Therefore, the importance of this kind of encryption system to cloud computing and its services is obvious.

There are two basic homomorphic operators – addition and multiplication. Let a, b be two integers. Addition of these integers in an encrypted form is the homomorphic addition. Similarly, multiplication of these integers in encrypted form is the homomorphic multiplication.

Homomorphic Addition - $\oplus$:

$$\mathrm{Decrypt}\left(\mathrm{sk}, \left(\mathrm{Encrypt}(\mathrm{pk}, \mathrm{a}) \oplus \mathrm{Encrypt}(\mathrm{pk}, \mathrm{b})\right)\right) = \mathrm{a} + \mathrm{b};$$

Homomorphic Multiplication - $\otimes$:

$$\mathrm{Decrypt}\left(\mathrm{sk}, \left(\mathrm{Encrypt}(\mathrm{pk}, \mathrm{a}) \otimes \mathrm{Encrypt}(\mathrm{pk}, \mathrm{b})\right)\right) = \mathrm{a} \times \mathrm{b},$$

where pk is the public key, sk is the secret key, Encrypt() is a function to encrypt values, and Decrypt() is a function to decrypt values.

This kind of encryption schemes can be classified into two categories according to their different encrypted objects, which are encrypting the integers directly and encrypting the bits in the binary forms of the integers. The one which encrypts the integers directly is called as the encryption schemes which have the homomorphic properties. The one which encrypts the bits in the binary forms of the integers is called as the homomorphic encryption schemes. In this thesis, a homomorphic encryption scheme is used, since the secure range query process is realized mainly relying on the homomorphic operations.

Normally, homomorphic encryption schemes are asymmetric encryption systems, so there is a public key (pk), and a secret key (sk). The public key is generated based on the secret key, so these two keys are arithmetically related. The secret key is used to execute the decryption process and the public key is mainly used to execute the encryption process. Specially, for homomorphic encryption schemes, the public key is also used in the process of executing operations on the encrypted bits. The reason that homomorphic encryption schemes have homomorphic properties is the odd-even properties of integers, and the homomorphic properties are realized by that the modulus operation. Therefore, although in [21, 22, 23, 24, 25] the key generating processes and the encryption algorithms are different, but the decryption algorithms are the same. The decryption of an encrypted binary digit c is given by the following relation:

$$\mathrm{m} = (\mathrm{c} \bmod \mathrm{sk}) \bmod 2,$$

where $\mathrm{m} \in \{0,1\}$, c is the encrypted form of m, sk is the secret key, and mod is the modulus operator.

The one that is put great hope on in the homomorphic encryption schemes is the fully homomorphic encryption scheme. The first fully homomorphic encryption scheme was proposed by Craig Gentry in his PhD thesis [21]. Craig Gentry presented the first true fully homomorphic encryption scheme which theoretically can execute any operations on encrypted values. At the first, he proposed a somewhat homomorphic encryption scheme, and then, through the recursive self-embedding method, and the bootstrappable encryption method, he converted the somewhat homomorphic encryption scheme to a fully homomorphic encryption scheme. However, the full homomorphic encryption scheme is quite complicated, so in [22], the researchers use Gentry's method to convert a somewhat homomorphic encryption scheme to a fully homomorphic encryption scheme, but without working with ideal lattices over a polynomial ring, and only involving the homomorphic addition and the homomorphic multiplication. This work is much simpler than Gentry's. In [23], Dianhua Tang et al. presented another somewhat homomorphic encryption scheme which has a smaller public key and is more efficient. All these three works are to encrypt the bits in the binary forms of the integers.

However, there are still some unsolved problems for using fully homomorphic encryption scheme in the practical applications. Two main problems are the noise control problem, and the efficiency problem. In fully and somewhat homomorphic encryption schemes, the operations on encrypted bits are executed through some circuits [21, 22, 23]. One kind of operations is corresponding to a circuit. Let C be a set which contains all the circuits that a homomorphic encryption scheme $\mathcal{E}$ can handle. If for any existing circuit, it can be found in the set C, then the homomorphic encryption scheme $\mathcal{E}$ is fully homomorphic. A method called $\mathsf{Evaluate()}$ can be used to execute the homomorphic operations on encrypted bits which needs to use the public key in the process. If the homomorphic encryption scheme $\mathcal{E}$ can handle a circuit, then this homomorphic encryption scheme $\mathcal{E}$ can execute the homomorphic operation that is contained in this circuit, and the correctness of the decrypted result of this operation is also guaranteed. The homomorphic encryption schemes encrypt the bits in the binary forms of the integers, and in most of the time, the homomorphic operations are also executed on the encrypted bits. However, the values of the integers will be changed at the end of the homomorphic operations, since the bits in the binary forms of these integers are changed.

Another important problem of homomorphic encryption schemes is to control the noise. In the execution of the homomorphic operations, there is some noise will be generated because of the execution of the basic homomorphic addition or homomorphic multiplication. Basically, one time of the execution of the homomorphic addition will double the volume of the noise, and one time of the execution of the homomorphic multiplication will square the volume of the noise. With the increase of the times that the two basic homomorphic operations are executed, the volume of the noise of the result also increases. When the volume of the noise hits some boundary which could be the value of the secret key sk, or some value that is arithmetically related to sk, and the encrypted result is decrypted, the true result may not be obtained. Therefore, the correctness of this homomorphic encryption scheme will not be guaranteed. This is a basic reason that homomorphic encryption schemes cannot be fully used in practical applications. Researchers seem to have a dilemma here. On one hand, when the size of the secret key is small, the encryption scheme could lose correctness. On the other hand, when the size of the secret key is large, the encryption scheme could lose efficiency. In Gentry's work [21], and also in [22, 23], they use a method called bootstrappable encryption method to control the noise.

Except for the homomorphic encryption schemes, there are also some other encryption schemes with homomorphic properties, which encrypt the integers directly instead of encrypting the bits in the binary forms of the integers. This is the first category that is mentioned before. A relatively early encryption scheme with homomorphic properties is RSA encryption scheme. This encryption scheme has already been used for a long time in practical applications. This encryption scheme was proposed for the first time by Ron Rivest, Adi Shamir, and Lenonard Adleman in 1977 [24]. The letters "RSA" are the first letters in their surnames. In their encryption scheme, the key for encryption is public, and the key for decryption is private. The private key holders can sign a piece of message, and anyone who is holding the corresponding public key can verify this signature. The signer cannot forge, or deny the validity of his signature. At the first, this encryption scheme was used for electronic email and electronic fund transfer systems. Another famous encryption scheme with homomorphic properties, which also encrypts the integers directly, is Paillier cryptosystem [25]. Through Composite Residuosity Class Problem, Paillier proposed a public-key based asymmetric encryption scheme. They proposed a new trapdoor mechanism, and derived three encryption schemes from this: one trapdoor permutation, and two homomorphic probabilistic encryption schemes. They only used the basic modulus operation in their scheme, and proved their scheme is secure under some appropriate assumptions. Paillier cryptosystem has two homomorphic properties which are homomorphic addition and homomorphic multiplication. Another encryption scheme with homomorphic properties is shown in [26]. They present a different trapdoor technique comparing to RSA, and proposed a new probabilistic encryption scheme. This encryption scheme has homomorphic addition properties. Since these encryption schemes do not encrypt the bits in the binary forms of the integers, but encrypt the integers directly, for efficiency and simplicity, they have advantages over the fully and somewhat homomorphic encryption schemes. However, because of the limitation of their homomorphic properties, they can only execute some homomorphic operations on encrypted data, only homomorphic addition, or homomorphic multiplication, or both to be exact. Theoretically, fully homomorphic encryption schemes can execute all the homomorphic operations on encrypted bits, so fully homomorphic encryption scheme is still a research area which is worth to study.

In the academic circles, there exit some papers about trying to use fully or somewhat homomorphic encryption schemes to realize some practical applications. However, there are still some serious limitations in all these papers, such as the maximum value of the integers that these applications can handle is too small, or the efficiency of these applications is too low. A possible reason for these limitations probably is the limitation of security parameter choices. In [27], the author discusses the possibility of using homomorphic encryption schemes in practical applications. If somewhat homomorphic encryption schemes are enough, or the value of the security parameter is not too large, or a true fully homomorphic encryption scheme appears, then the homomorphic encryption schemes can truly be used in practical applications. There is some research in this area. In [28], the researchers proposed an application on a private video streaming service with a somewhat homomorphic encryption scheme. In [29], the researchers presented an application which is receipt-free voting based on a homomorphic encryption scheme. This thesis is also about using homomorphic encryption scheme to enable an application, which is a secure range query process.

2.2 Secure Database Queries

Although some cloud service providers provide some secure cloud databases as services, these cloud database services do not have complete database functionalities compare to general databases. Therefore, how to enable database functionalities fully in cloud database services is still an open problem for researchers. An important one of the database functionalities is querying the database in the cloud. Since the data that is stored in the cloud database is encrypted, this is quite a challenging problem to solve. In [30], the researchers proposed a distributed architecture that can let the organization outsource his data to two untrusted servers, but in the process, the organization can still keep his privacy. In their work, they show how these two untrusted parties work together efficiently and securely, and there is no information leakage to these two parties, or outsiders. The researchers also show how to enable secure query processing in this distributed architecture. However, there is a problem that the architecture cannot handle the collusion between the two untrusted parties. This work is realized under an assumption that the collusion between the two servers is quite impossible. In [31], the researchers present an algebraic framework which can split the query jobs to reduce the computational costs of the client side.

If fully or somewhat homomorphic encryption schemes are not used to encrypt the data at the first, and some encryption schemes with homomorphic properties are used to encrypt the data at the first, multi-party computation problem (MPC) should be used to enable the secure data query processing. This means the user and the cloud need to communicate with each other without any information leakage to each other to work out the results that they both want together. The multi-party computation problem was presented firstly by

Andrew C. Yao [32] in 1982 through a famous question that is called as the millionaires' problem. The millionaires' problem is that there are two millionaires, and they both want to know who is richer between them, but they do not want each other to know how much money they have. Therefore, they communicate with each other in a way that will not expose their own secrets, which is how much money they have, to work out a result together they both want to know, which is who is richer between them. A formal expression is given as:

$$\mathrm{f}(s_1, \dots, s_n) = < r_1, \dots, r_b >,$$

where r_i is the output result that the party P_i wants, and s_i is the secret that party P_i wants to keep.

Multi-party computation (MPC) is very important to the realization of the secure query processing, if fully or somewhat homomorphic encryption schemes are not used to encrypt the data at the first [33, 35].

Except for the original cloud service provider, involving one or more cloud service providers in the environment seems to be an available way to solve the secure query processing problem. In [33], the researchers proposed a federation of cloud computing that can enable secure query processing, and have fine-grained access control of the users' authorities to the encrypted data in the cloud. As mentioned in [32], the researchers let two clouds communicate with each other securely to work together for the results of the queries. The primary cloud randomizes the encrypted data that the user requests a query on, and then sends the randomized encrypted data to the secondary cloud. The secondary cloud has the secret key, and uses it to decrypt the randomized data. After the decryption, the secondary cloud executes the query and gets the results, but these results still have the randomization. The secondary cloud encrypts the results and sends these back to the primary cloud. The primary cloud then removes the randomization of the encrypted results, and sends them to the user. At the end, the user can decrypt the results to gain the true results of his query. In this process, both these clouds cannot know each other's, the data owner's, or the user's secrets. The primary cloud uses randomization to keep its secrets, and the secondary cloud uses the secret key to keep its secrets. This research is also under the assumption that the collusion between the two clouds is negligible. If there is collusion between the two clouds, the data will be totally exposed to them, because they have the encrypted data, and the secret key for decryption.

A key technique to realize secure range query processing is secure comparison (SC) algorithm [34]. Generally speaking, there are two main ways to realize a secure comparison process. The first one is to use fully or somewhat homomorphic encryption schemes to encrypt the bits in the binary forms of the integers, and then compare two encrypted arrays which are the binary encrypted forms of two integers to gain the final result of the comparison between the two integers. The second one is to use the encryption schemes with homomorphic properties to encrypt the decimal integers directly instead of using fully or somewhat homomorphic encryption schemes to encrypt the bits in the binary forms of the integers. Then, use secure bit decomposition (SBD) to convert the encrypted integers to the encrypted binary forms of them. Again, use secure comparison to compare the two arrays which are the encrypted binary forms of two integers bit by bit to gain the final result of the comparison between two integers [35].

In [35], the researchers proposed a framework using secure bit decomposition, secure comparison, and multi-party computation to realize a secure range query processing called Privacy-Preserving Range Queries (PPRQ). They use Paillier cryptosystem to encrypt the integers directly, and then use secure bit decomposition to convert the encrypted integers to the encrypted binary forms so that an encrypted integer is an array of encrypted bits. Then, they use the secure comparison algorithm from [34] to execute the secure comparison process. Two cloud service providers are involved as it needs two parties to communicate with each other securely to finish the secure comparison process in a multi-party computation way.

Since the privacy-preserving range queries (PPRQ) method from [35] involves two clouds, there probably are some unexpected security problems, like the collusion between the two clouds. Therefore, this thesis uses a somewhat homomorphic encryption scheme to solve the secure range query problem involving just one cloud service provider in the cloud environment. Information leakage has to be prevented on two fronts. The first one is to prevent the information leakage from the outsiders. Basically, the cloud service provider will guarantee the security of this. The second one is to protect the sensitive information from the insiders of the cloud service provider. This depends on the somewhat homomorphic encryption scheme that is used, and the way that the secure range query process is realized.

Chapter 3

Preliminaries

3 Preliminaries

This chapter introduces some existing techniques that are used in the thesis. First, a somewhat homomorphic encryption scheme is presented that is used in the thesis. Since fully homomorphic encryption scheme cannot be used in practical applications, a somewhat homomorphic encryption scheme is used to realize secure range queries instead of using a fully homomorphic encryption scheme. Second, four basic homomorphic operators are introduced which will be used to build the greater-than algorithm. All these operators are executed on the encrypted bits in the binary forms of the integers. Two original ones are homomorphic addition and homomorphic multiplication from the nature of the somewhat homomorphic encryption scheme. Other two are built from the two original ones. Then, it shows how to use these basic homomorphic operators to build a homomorphic greater-than algorithm that is used in the secure range query process. At the end of the chapter, a Java class is introduced from the Java library that is called BigInteger that is used to program the experiments.

3.1 Somewhat Homomorphic Encryption Scheme

The somewhat homomorphic encryption scheme is to encrypt the bits in the binary forms of the integers [22]. The data that is involved in the secure range query process is integer data. First, these integers in the dataset will be converted to binary form. Then, these binary forms of the integers are encrypted.

Parameters Definition:

λ: security parameter; the value of this parameter can affect the security of the somewhat homomorphic encryption scheme and the controllable noise boundary;

η: the bit-length of the secret key; the value of this parameter decides the size of the secret key and furthermore, affects the security of the somewhat homomorphic encryption scheme and the controllable noise boundary;

γ: the bit-length of the public key; the value of this parameter decides the size of the public key and furthermore, affects the security of the somewhat homomorphic encryption scheme.

Let $\eta = \lambda^2$ and $\gamma = \lambda^5$. Yet the relationship between the values of these two parameters and the value of security parameter λ is indeterminate, since this relationship need to be changed sometimes to balance the security and the efficiency of the application.

Then the somewhat homomorphic encryption scheme is defined as follows.

skGen(): the method to generate the secret key p. p can be used to generate the public key, and decrypt the encrypted bits to obtain the integers again. In this method, choose an odd integer p from the range $[2^{\eta-1}, 2^{\eta})$ as the secret key;

pkGen(): the method to generate the public key N. The public key is usually associated arithmetically with the secret key, and can be used to encrypt the bits in the binary forms of the integers. In this method, choose an integer q from the range $[2^{\gamma-1}, 2^{\gamma})$, and then, let $N = pq$. N is the value of the public key;

Encrypt(N, m): the method Encrypt(N, m) takes the public key N and a bit m in the binary form of an integer as the inputs, and returns an encrypted form of the bit m as the output. For $m \in \{0, 1\}$, let c be the encrypted form of m. Then c can be computed as:

$$c = m + 2r + N,$$

where r is a random integer which is chosen from the range $[2^{\lambda-1}, 2^{\lambda})$;

Decrypt(p, c): the method Decrypt(p, c) takes the secret key p and an encrypted value c as the inputs, and returns the decrypted value of c which is a true bit m as the output. Then m can be computed as:

$$m = (c \bmod p) \bmod 2,$$

where mod is the modulus operator.

Since this somewhat homomorphic encryption scheme is to encrypt the bits of the binary forms of the integers in the dataset, one single bit after the encryption process is a single encrypted value. For each encrypted integer in some column of the dataset, let $c_i \in \{c_1, \dots, c_n\}$, where $c_1, \dots, c_n$ are all the encrypted integers in the column, and n is the total number of the integers in the column. Since the integers are encrypted by encrypting individual bits of their binary forms, each encrypted integer is an array of encrypted bits. An encrypted integer c_i will consist of an array of encrypted bits $\{c_{i1}, \dots, c_{ib}\}$, where b is the bit-length of the binary form of the integer c_i.

In the somewhat homomorphic encryption scheme, the 2r generated in the process of the encryption is the noise. The volume of the noise will increase with the increase in the number of times the operations are executed, since in a certain sense, every time an operation is executed the result is encrypted one more time. Only when the volume of the noise is less than the value of the secret key p, the correctness of the decryption process can be guaranteed. The somewhat homomorphic encryption scheme has four basic operators, the homomorphic addition, homomorphic multiplication, NOT and OR operations. The NOT and OR homomorphic operations can be constructed from the homomorphic addition and homomorphic multiplication operators.

3.2 Basic Operator Algorithms

In [36], researchers build more complicated homomorphic operators from the homomorphic addition and homomorphic multiplication operators. The four basic homomorphic operators are defined on encrypted bits, that is, encrypted 0 and encrypted 1 in their execution. Since these four operators are executed on individual encrypted bits, the decrypted results of these homomorphic operators do not have any carry bits. These operators are not exactly like the binary arithmetic operators but like the logical operators on 0 and 1. Therefore, these four basic homomorphic operators can be defined as EXCLUSIVE OR, AND, NOT, and OR where EXCLUSIVE OR represents the homomorphic addition, AND represents the homomorphic multiplication.

Basic homomorphic operators:

EXCLUSIVE OR: is the homomorphic addition operator on encrypted bits c_1 and c_2 and is denoted as XOR. It can be expressed as follows:

$$c_1 \oplus c_2 = c,$$

where c is the result of the addition of the two encrypted bits.

The homomorphic property of this operator is given by:

$$\text{Decrypt}(p, c) = m = m_1 + m_2,$$

where m_1 and m_2 are the corresponding true bits of the encrypted bits c_1 and c_2, m is the addition of the two true bits, p is the secret key, and Decrypt() is the decryption function.

AND: is the homomorphic multiplication operator on encrypted bits c_1 and c_2. It can be expressed as follows:

$$c_1 \otimes c_2 = c,$$

where c is the result of the homomorphic multiplication of the two encrypted bits.

The homomorphic property of this operator is given by:

$$\mathrm{Decrypt}(\mathrm{p}, \mathrm{c}) = \mathrm{m} = \mathrm{m}_1 \times \mathrm{m}_2,$$

where m_1 and m_2 are the corresponding true bits of the encrypted bits c_1 and c_2, m is the multiplication of the two true bits, p is the secret key, and Decrypt() is the decryption function.

NOT: is the homomorphic addition of an encrypted bit c_1 and an encrypted 1. It can be expressed as follows:

$$NOT(c_1) = c_1 \oplus Encrypt(N, 1) = c,$$

where c is the result of the homomorphic addition of the encrypted bit c_1 and an encrypted 1, N is the public key, and Encrypt() is the encryption function.

The homomorphic property of this operator is given by:

$$\mathrm{Decrypt}(\mathrm{p}, \mathrm{c}) = \mathrm{m} = \mathrm{NOT}(\mathrm{m}_1),$$

where m_1 is the corresponding true bit of the encrypted bit c_1, m is the decrypted form of c, p is the secret key, and Decrypt() is the decryption function. NOT(m_1) can be computed by adding m_1 and 1 in binary arithmetic without bit carrying.

OR: is derived from the three previous homomorphic operators:

$$c_1 \text{ OR } c_2 = (c_1 \otimes c_2) \oplus \Big((\text{NOT}(c_1) \otimes c_2) \oplus \big(c_1 \otimes \text{NOT}(c_2)\big)\Big) = c,$$

where c_1 and c_2 are two encrypted bits, and c is the result of the OR operation on the two encrypted bits.

The homomorphic property of this operator is given by:

$$\text{Decrypt}(p, c) = m = m_1 \text{ OR } m_2,$$

where m_1 and m_2 are the corresponding true bits of the encrypted bits c_1 and c_2, m is the decrypted form of c, p is the secret key, and Decrypt() is the decryption function. $m_1 \text{ OR } m_2 = (m_1 \times m_2) + ((\text{NOT}(m_1) \times m_2) + (m_1 \times \text{NOT}(m_2)))$

Table 1 gives all the possible results of the four basic homomorphic operators. Let an encrypted bit m be denoted in a form as ENC(m).

Table 1: Four Basic Homomorphic Bit Operators and Their Results

c_1	c_2	XOR $\oplus$	AND $\otimes$	NOT	OR
ENC(0)	ENC(0)	ENC(0)	ENC(0)	ENC(1)	ENC(0)
ENC(0)	ENC(1)	ENC(1)	ENC(0)	ENC(1)	ENC(1)
ENC(1)	ENC(0)	ENC(1)	ENC(0)	ENC(0)	ENC(1)
ENC(1)	ENC(1)	ENC(0)	ENC(1)	ENC(0)	ENC(1)

3.3 Greater-Than Algorithm

The greater-than algorithm [36] is required for secure range query processing. The greater-than algorithm is defined by the function GreaterThan(). This function takes two encrypted integers which are two arrays of encrypted bits as the inputs. By execution of a combination of the four basic homomorphic operators on the encrypted bits of the two encrypted integers, an encrypted result of the comparison of the two encrypted integers to find the greater of the two integers can be generated. The decryption of the encrypted result will give the comparison of the true values of the two integers.

Let c and c' be two encrypted integers, and m and m' be the corresponding true values of the two integers.

If m > m', then

GreaterThan(c, c') = Encrypted 1 and

Decrypt(p, GreaterThan(c, c')) = 1, which means m > m' is true.

If m < m', then

GreaterThan(c, c') = Encrypted 0 and

Decrypt(p, GreaterThan(c, c')) = 0, which means m > m' is false.

It should be noted that Encrypted 1 and Encrypted 0 include the noise accumulated through multiple encryption operations. Since the greater-than algorithm is derived from the four basic homomorphic operators, if the volume of noise is under a controllable boundary, the greater-than algorithm should also be homomorphic. The greater-than algorithm is given in Figure 2.

```
GreaterThan(c, c')
    Input: two encrypted integers c and c', where c = {$c_1$, …, $c_n$} and c' = {$c'_1$, …, $c'_n$}
    Output: an encrypted 0 or 1
    Initialize
        result = Encrypt(N, 0);
        done = Encrypt(N, 0);
    For i = 1, …, n (n is the bit length)
        $t_1 = c_i \otimes$ NOT($c'_i$);
        $t_2 = c'_i \otimes$ NOT($c_i$);
        result = (done $\otimes$ result) $\oplus$ (NOT(done) $\otimes$ $t_1$);
        done = done $\oplus$ (NOT(done) $\otimes$ ($t_1$ OR $t_2$));
return result
```

Figure 2: Greater-Than Algorithm

3.4 Java Class - BigInteger

To perform some experiments in this thesis to test the practicability of the secure range query framework, a test program is required to realize the secure range query process. In this thesis, Java programming language is used to realize the secure range query process. Since there are some large integers involved in the somewhat homomorphic encryption scheme, such as in pkGen(), q is chosen from the range $[2^{\gamma-1}, 2^{\gamma})$. There are some integers which are beyond the boundaries of the primitive integer data types in Java. Therefore, a Java class which is called as BigInteger [37] is used from the Java library to handle the large integers in the process.

This class is designed specifically to handle the problems that involve large integers. The BigInteger class can provide all the operations of the primitive integer data types in Java, and all the operations from java.lang.Math. Additionally, this class also provides modulus related operations, GCD calculation, primality testing, prime generation, and other complicated methods.

It is worth to mention the methods to generate big random integers in the BigInteger class. Since in the somewhat homomorphic encryption scheme, some random large integers need to be chosen from some default ranges, the methods for generating random big integers are given next. The methods are as follows (assume that rndSrc provides an equitable random source):

BigInteger(int, int, Random):

returns a random large integer (probably be prime) with the given bit-length;

BigInteger(int, Random):

returns a random integer from the range [0, $2^{\text{numBits - 1}}$].

Chapter 4

Secure Range Query Using Somewhat Homomorphic Encryption

4 Secure Range Query

The query objects of the secure range query are the encrypted integer data by the somewhat homomorphic encryption scheme. Since this range query process is secure, the data and every result of every execution of greater-than algorithm is encrypted. Additionally, there is no sensitive or useful information leakage to any party that does not have the authority to access this information except for the user who requests the query. This section first introduces the framework of the secure range query, and then, for some security reasons, shows how to hide the real bit-length of each encrypted integer in the data set. In this way, the attackers or other untrusted parties can be prevented to conjecture the approximate value range of the data set. Next part shows how to realize the user access control, since there probably are some users who have no privileges to access some data. Then, the correctness and the security of the secure range query process are analyzed. At the end, the influence and the setting of some important parameters is illustrated which are involved in the secure range query process.

4.1 Secure Range Query Application

The form of the data set used in the process is in row-column form. Let the lower-case letter s to represent the number of the columns in the data set. As mentioned earlier, if the user wants to execute a range query process, he needs to send a request message to the cloud service provider. The message includes two encrypted integers and one array. The two integers are encrypted by the somewhat homomorphic encryption scheme which are the lower boundary and the upper boundary of the range in the range query. Let U represent the upper boundary of the range and L represent the lower boundary of the range. Let $\mathcal{u}$ denote the encrypted value of U and let ℓ be the encrypted value of L. Since ℓ and $\mathcal{u}$ are also encrypted integers, they are two arrays which contain some encrypted bits. The array contains the indexes of the columns of the data set that the user wants to query on, denoted as H. Then, the cloud service provider can execute the secure range query on some columns. Then a request message to the cloud will be represented as:

Query Request Message: $\{H, \mathcal{u}, \ell\}$.

Choose an encrypted integer c_i from the set $\{c_1, ..., c_n\}$, where $c_1, ..., c_n$ are n encrypted integers of the integers $m_1, ..., m_n$ in some column. Further, c_i consists of an array $c_{i1}, \ldots, c_{ib}$ of encrypted bits of the integer with b as the bit-length of the integer.

Let $m_i, m_j \in \{m_1, ..., m_n\}$.

If $m_i > m_j$, then GreaterThan(c_i, c_j) = Encrypted 1;

If $m_i < m_j$, then GreaterThan(c_i, c_j) = Encrypted 0.

For each $c_i \in \{c_1, \ldots, c_n\}$, we compute GreaterThan(c_i, ℓ) and GreaterThan(c_i, u).

If $m_i > U$, we will get

GreaterThan(c_i, ℓ) = Encrypted 1 and

GreaterThan(c_i, u) = Encrypted 1;

If $m_i < L$, we will get

GreaterThan(c_i, ℓ) = Encrypted 0 and

GreaterThan(c_i, u) = Encrypted 0;

If $L < m_i < U$, we will have

GreaterThan(c_i, ℓ) = Encrypted 1 and

GreaterThan(c_i, u) = Encrypted 0.

If we do the homomorphic addition (XOR) operator on the two results in each condition from the above, only the condition that m_i is in the range (L, U) will give us the result which is an encrypted 1, and other two will give us the results which are all encrypted 0s.

Then, we compute

```
for i = 1 to n
    for j = 1 to b
        $c_{ij} = c_{ij} \otimes (\text{GreaterThan}(c_i, \ell) \oplus \text{GreaterThan}(c_i, u))$.
```

When m_i is in the range (L, U), the result of $\text{GreaterThan}(c_i, \ell) \oplus \text{GreaterThan}(c_i, u)$ will be an encrypted 1 and in this case the true value of c_{ij} in the above computation will remain unchanged as its true value is multiplied by 1. Consequently, the true value of c_i will not change if m_i is in the range (L, U).

When m_i is not in the range (L, U), the result of $\text{GreaterThan}(c_i, \ell) \oplus \text{GreaterThan}(c_i, u)$ will be an encrypted 0 and in this case the true value of c_{ij} in the above computation will be changed to 0 as its true value is multiplied by 0. Consequently, the true value of c_i will change to 0 if m_i is not in the range (L, U).

The data owner will encrypt the data and in the encryption process, and a public key and a secret key will be generated. Then, the data owner will send the encrypted data to the cloud service provider for storage, and also send the public key to the cloud service provider for using in the future. The data owner will send the secret key to the users inside the organization to let them decrypt results. Figure 3 shows the data outsourcing process.

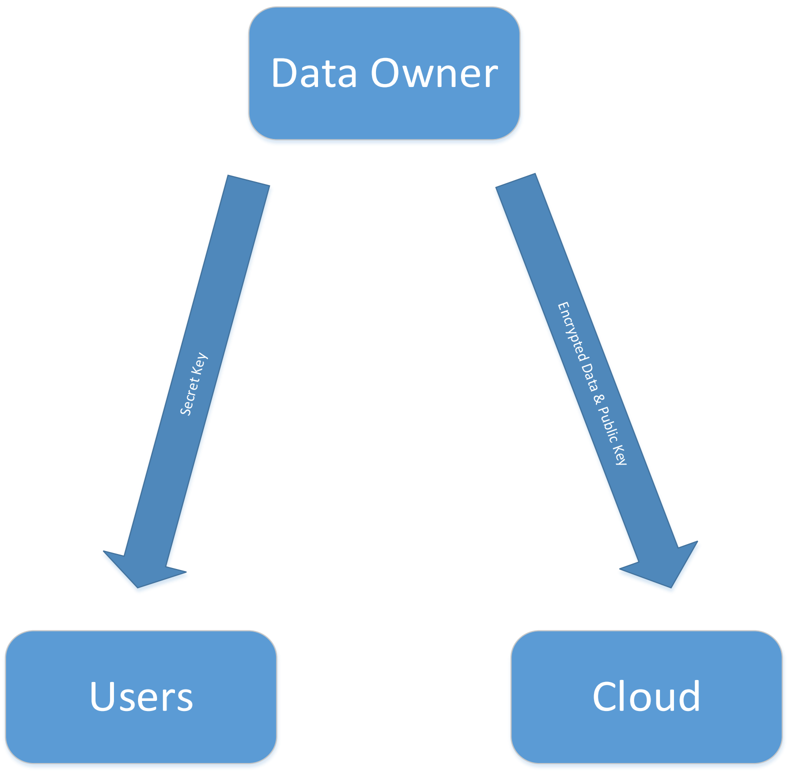

Figure 3: Data Outsourcing Process

After the data outsourcing process, the users inside the organization can request range queries to the cloud service provider. As shown in Figure 4, the user generates the request message and then sends the request message to the cloud service provider. When the cloud service provider receives the request message, after the user authority authentication, the cloud service provider will execute the above secure range query process on all the encrypted integers in the columns that are requested to query on.

Then, the cloud service provider sends all the encrypted results to the user. At the end, the user decrypts all the encrypted results, and removes the useless 0s to obtain the real results of the range query.

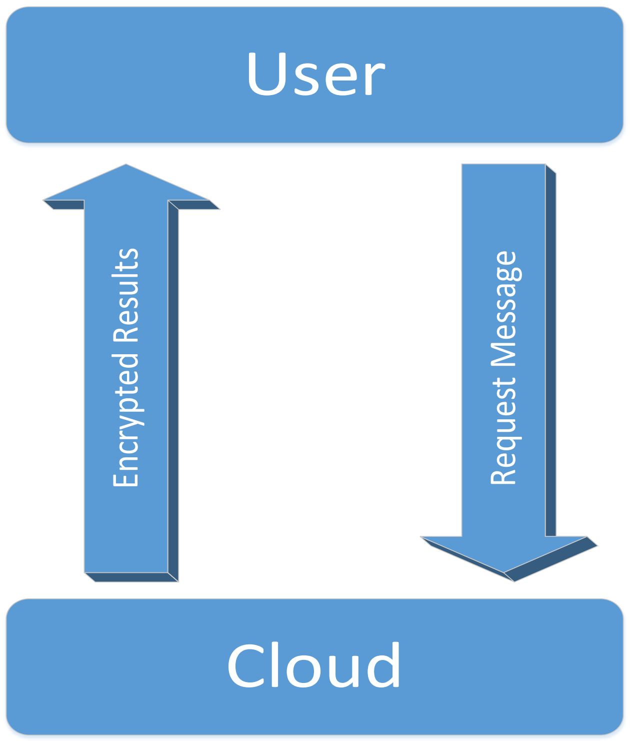

Figure 4: Secure Range Query Process

Figure 5 shows the activities that are done by the three parties, which are the data owner, the cloud, and the user, in the secure range query process.

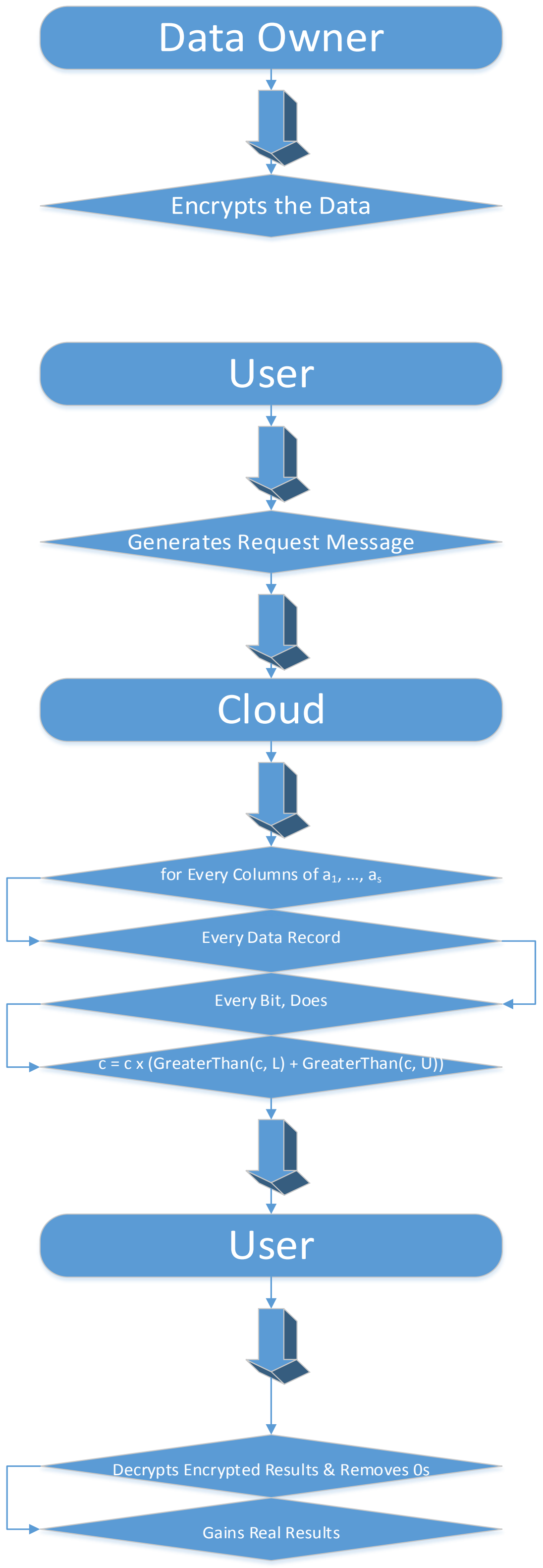
Data Owner
Encrypts the Data
User
Generates Request Message
Cloud
for Every Columns of $a_1, \ldots, a_s$
Every Data Record
Every Bit, Does
c = c x (GreaterThan(c, L) + GreaterThan(c, U))
User
Decrypts Encrypted Results & Removes 0s
Gains Real Results

Figure 5: Inside Activities in SRQ

4.2 Bit-Length Hiding

In this subsection, another important technique that is used in the secure range query process to enhance the security and protect the privacy of the data owner is explained. In the above secure range query process, a hidden security problem is that the number of the bits in the binary forms of the integers from the data owner is exposed to the cloud service provider. By knowing the numbers of bits in the binary forms of the integers from the data owner, the cloud service provider can deduce some relatively smaller ranges of the encrypted integers, and, furthermore, can deduce some relatively small ranges for each attribute in the database. Because of this, some sensitive information of the data owner will be exposed to the cloud service provider, which the data owner does not want to be exposed to an untrusted party. Therefore, hiding the real bit-length of the encrypted integers in the database is a very important part of ensuring security and the privacy protection in the secure range query process.

The process of bit-length hiding is as follows.

First, let $c_i = \{c_{i1}, c_{i2}, ..., c_{in}\}$, where c_i is an encrypted integer, and $c_{i1}, c_{i2}, ..., c_{in}$ are c_i's encrypted bits. There are n bits in the binary form of the true value of c_i.

In the keys generation part, we generate another key which is called as the bit-length hiding key, which is sent to the user by the data owner. We defined it as h.

Then, we can build a new value

$$m_0 = \{0, \ldots, 0\} \xrightarrow[\text{encryption}]{} c_0 = \{c_{01}, c_{02}, \ldots, c_{0h}\},$$

where m_0 is a number 0 (zero) but is expressed with h 0s in its binary form, and c_0 is the encrypted form of m_0.

Append c_0 to c_i to gain

$$c_i + c_0 = \{c_{i1}, \ldots, c_{in}, c_{01}, \ldots, c_{0h}\}.$$

Then, we can let $c_i + c_0$ hide the real bit-length of c_i, and send $c_i + c_0$ instead of c_i to the cloud service provider to store.

In the secure range query process, we have

$$\text{GreaterThan}(c_i, x) = \text{GreaterThan}(c_i + c_0, x + c_0),$$

where x is an encrypted integer for comparison.

Since in the greater-than algorithm, we compare two encrypted integers based on their encrypted binary forms through all their encrypted bits, the above equality is tenable. Therefore, this bit-length hiding method will not affect the results of the greater-than algorithm, and consequently will not affect the results of the secure range query process. After the secure range query process, the cloud service provider sends the results back to the user. The user decrypts the results, and removes the extra 0s based on the value of h, that is the bit-length hiding key that is kept by the user, through the logic shift method, or the arithmetic shift method to gain the real results of the range query.

Since the number of the extra 0s is another key which is kept by the user, in order not to increase the costs of the computation and storage on the user's side, the same number of extra 0s is used for all the integers. This means the same number of 0s is appended to each encrypted integer. However, this bit-length hiding key for every encrypted integer must be large enough to prevent the detective attacks. However, if the bit-length hiding key is too large, it will have the risk to lose efficiency. The bit-length hiding key is regarded as a parameter in the secure range query process. Details for setting this parameter are given in Section 4.6.

4.3 User Access Control

Another important function in the secure range query process is to control the user's authority to access the data in the database inside the organization. Since the users in the organization may be from different departments, or at different levels, they may have different access authorizations to the different attributes of the data in the database. For example, only a few users may have the authorities to access some very sensitive data in the organization, and the users from different departments may not have the authorities to access the data from other departments. Therefore, we need to control the authorizations of the users in the secure range query process.

In the user authorization controlling method, each user has a unique user ID to verify their access authorization for the attributes in the database. Each user ID has a corresponding verification file. The file contains the indexes of the attributes that the user has the authorization to access. These verification files are generated by the data owner at the first place, and outsourced to the cloud service provider with the encrypted data and the public key.

In the secure range query process, before the user wants to execute a range query, he needs to generate a request message. In addition to the content that should be covered in the request message, the ID of the user should also be covered in the request message. Once the cloud service provider receives the request message, he first checks the verification file of the user based on the user ID in the request message. The cloud service provider needs to find the intersection between the requested attributes in the request message and the authorized attributes in the verification file of the user. The attributes in the intersection are the attributes that the cloud service provider will execute the range query on. Figure 7 shows the user authorization controlling process.

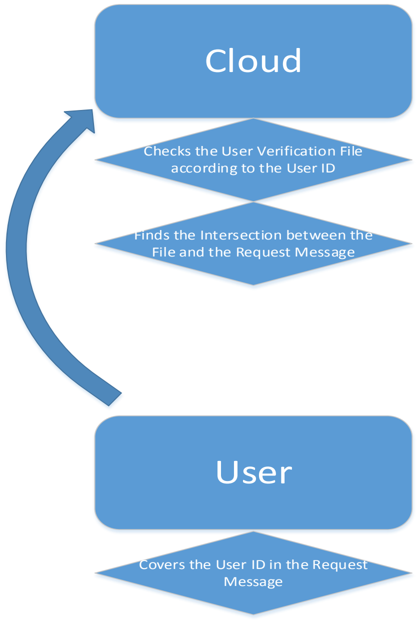
Cloud
Checks the User Verification File according to the User ID
Finds the Intersection between the File and the Request Message
User
Covers the User ID in the Request Message

Figure 6: User Authorization Control Process

4.4 Correctness

This section illustrates how to ensure the correctness of the final true results of the secure range query process after the decryption process. Since the homomorphic encryption scheme that is used in the secure range query process is a somewhat homomorphic encryption scheme, it cannot handle all the homomorphic operations with any depth. When the depth of the executed homomorphic operation is increased, the noise that is generated in this process also increases. When the volume of noise in the final encrypted results is more than the maximum volume of the noise that the somewhat homomorphic encryption scheme can handle, the correctness cannot be ensured. This means that the user cannot obtain the correct final results of the secure range query after the decryption process. Therefore, in this section, the factors that affect the maximum volume of noise that a somewhat homomorphic encryption scheme can handle, the arithmetic relations between these factors and the volume of noise, and how to guarantee the correctness of the decryption process of the somewhat homomorphic encryption scheme when it is used in the secure range query process will be discussed.

First, what the noise that is generated in the process of the homomorphic operations execution in a somewhat homomorphic encryption scheme is will be explained.

The encryption process is as follows:

$$c = m + 2r + N,$$

where m is a true bit, r is a random integer which is chosen from the range $[0, 2^{\lambda})$, N is the public key, and c is the encrypted form of m.

The decryption process is as follows:

$$m = (c \bmod p) \bmod 2,$$

where p is the secret key.

Since $N = pq$, in the decryption process, the decryption expression can be changed to

$$m = ((m + 2r + pq) \bmod p) \bmod 2,$$

that is

$$m = ((m + 2r) \bmod p) \bmod 2 + (pq \bmod p) \bmod 2.$$

Since $pq \bmod p = 0$, and $0 \bmod 2 = 0$, the only part that will affect the results of the decryption process is

$$((m + 2r) \bmod p) \bmod 2.$$

Therefore, the part $m + 2r$ is the noise that is generated in the homomorphic operations process.

Next, the noise that is generated by the four basic homomorphic operators XOR, AND, NOT, and OR, and the upper boundary for the volumes of the noise that is generated by these four basic homomorphic operators will be shown.

Since $r \in [0, 2\lambda)$, and $m \in \{0, 1\}$, there is

$$m + 2r < 2^{\lambda + 1} + 1.$$

For the homomorphic operator XOR, there is

$$c_1 \oplus c_2 = (m_1 + 2r_1 + pq) + (m_2 + 2r_2 + pq)$$

$$= (m_1 + 2r_1) + (m_2 + 2r_2) + 2pq.$$

Since

$$(2pq \bmod p) \bmod 2 = 0,$$

the noise of this basic homomorphic operator is

$$(m_1 + 2r_1) + (m_2 + 2r_2).$$

Since

$$m + 2r < 2^{\lambda + 1} + 1,$$

there is

$$(m_1 + 2r_1) + (m_2 + 2r_2) < 2(2^{\lambda + 1} + 1) = 2^{\lambda + 2} + 2.$$

Therefore, the volume of the noise that is generated the homomorphic operator XOR must be less than $2^{\lambda + 2} + 2$.

For the homomorphic multiplication operator AND, there is

$$\begin{aligned} c_1 \otimes c_2 &= (m_1 + 2r_1 + pq)(m_2 + 2r_2 + pq) \\ &= (m_1 + 2r_1)(m_2 + 2r_2) + (m_1 + 2r_1 + m_2 + 2r_2 + pq)pq. \end{aligned}$$

Since

$$(((m_1 + 2r_1 + m_2 + 2r_2 + pq)pq) \bmod p) \bmod 2 = 0,$$

the noise of this basic homomorphic operator is

$$(m_1 + 2r_1)(m_2 + 2r_2).$$

Since

$$m + 2r < 2^{\lambda + 1} + 1,$$

there is

$$(m_1 + 2r_1)(m_2 + 2r_2) < (2^{\lambda + 1} + 1)^2 = 2^{2\lambda + 2} + 2^{\lambda + 2} + 1.$$

Therefore, the volume of noise that is generated by the homomorphic operator AND must be less than $2^{2\lambda + 2} + 2^{\lambda + 2} + 1$.

For the homomorphic operator NOT, there is

$$\text{NOT}(c) = c \oplus \text{Encrypt}(N, 1)$$

$$= (m + 2r + pq) + (m_1 + 2r_1 + pq)$$

$$= (m + 2r) + (m_1 + 2r_1) + 2pq,$$

where Encrypt() is the encryption function, N is the public key, and Encrypt(N, 1) is an encrypted 1.

Since

$$(2pq \bmod p) \bmod 2 = 0,$$

the noise of this basic homomorphic operator is

$$(m + 2r) + (m_1 + 2r_1).$$

Since

$$m + 2r < 2^{\lambda + 1} + 1,$$

there is

$$(m + 2r) + (m_1 + 2r_1) < 2(2^{\lambda + 1} + 1) = 2^{\lambda + 2} + 2.$$

Therefore, the volume of the noise that is generated by the homomorphic logic operator NOT must be less than $2^{\lambda + 2} + 2$.

For the homomorphic operator OR, there is

$$c_1 \text{ OR } c_2 = (c_1 \otimes c_2) \oplus ((\text{NOT}(c_1) \otimes c_2) \oplus (c_1 \otimes \text{NOT}(c_2))).$$

After simplification, the noise of this basic homomorphic operator is

$$3(m_1 + 2r_1)(m_2 + 2r_2) +(m_1 + 2r_1)(m_{11} + 2r_{11}) + (m_2 + 2r_2)(m_{11} + 2r_{11}),$$

where m_{11} is 1, and r_{11} is the random integer which is generated in the encryption process for this 1.

Since

$$m + 2r < 2^{\lambda + 1} + 1,$$

there is

$$3(m_1 + 2r_1)(m_2 + 2r_2) +(m_1 + 2r_1)(m_{11} + 2r_{11}) + (m_2 + 2r_2)(m_{11} + 2r_{11}) < 5(2^{2\lambda + 2} + 2^{\lambda}$$

$^{+2} + 1)$.

Therefore, the volume of the noise that is generated by the homomorphic operator OR must be less than $5(2^{2\lambda+2} + 2^{\lambda+2} + 1)$.

Since the deepest homomorphic operator in the secure range query is the homomorphic greater-than algorithm, ensuring that the noise that is generated by this operator is important.

In the greater-than algorithm, three values are definite in every loop in the algorithm. They are t_1, t_2, and t_1 OR t_2, since these three variables are assigned with the new encrypted bits.

Then, according to volume of the noise that is generated by the four basic homomorphic operators from the above, there is

$$\mathrm{Noise}(t_1) < 2(2^{\lambda+1} + 1)^2 = 2^{2\lambda+3} + 2^{\lambda+3} + 2,$$

$$\mathrm{Noise}(t_2) < 2(2^{\lambda+1} + 1)^2 = 2^{2\lambda+3} + 2^{\lambda+3} + 2,$$

and

$$\mathrm{Noise}(t_1 \text{ OR } t_2) < 12(2^{\lambda+1} + 1)^4 + 4(2^{\lambda+1} + 1)^3.$$

Since the values of the variables changes in every loop, and the values from the last loop are used in the current loop, it is very difficult to find the relations between the number of the bits of the encrypted integers and the noise of the final result.

Therefore, a Java program is used to calculate the noise values for the bit-length from 1 to b in the greater-than algorithm. In addition, the noise that is generated in the following process should be taken into account,

$$c_{ij} = c_{ij} \otimes (\mathrm{GreaterThan}(c_i, L) \oplus \mathrm{GreaterThan}(c_i, U)).$$

Figure 7 shows the volume of the noise that is generated in the secure range query process for bit-lengths from 1 to b in the greater-than algorithm. Here, the security parameter $\lambda = 3$, and the bit-length of the integers is 10, so that $b = 10$. Since the volume of the noise could be very large, so when $b = 10$, the trend of the curve can be observed, and for different security parameter λ, the curve is shown in Figure 7. As shown in Figure 7, with the increase of b, the volume of the noise in the final result increases drastically from $b = 9$ to $b = 10$. The reason that the noise level in the curve for b from 1 to 9 is not obvious is that when $b = 10$ the volume of the noise is much larger than the volume of noise for b= 1 to 9.

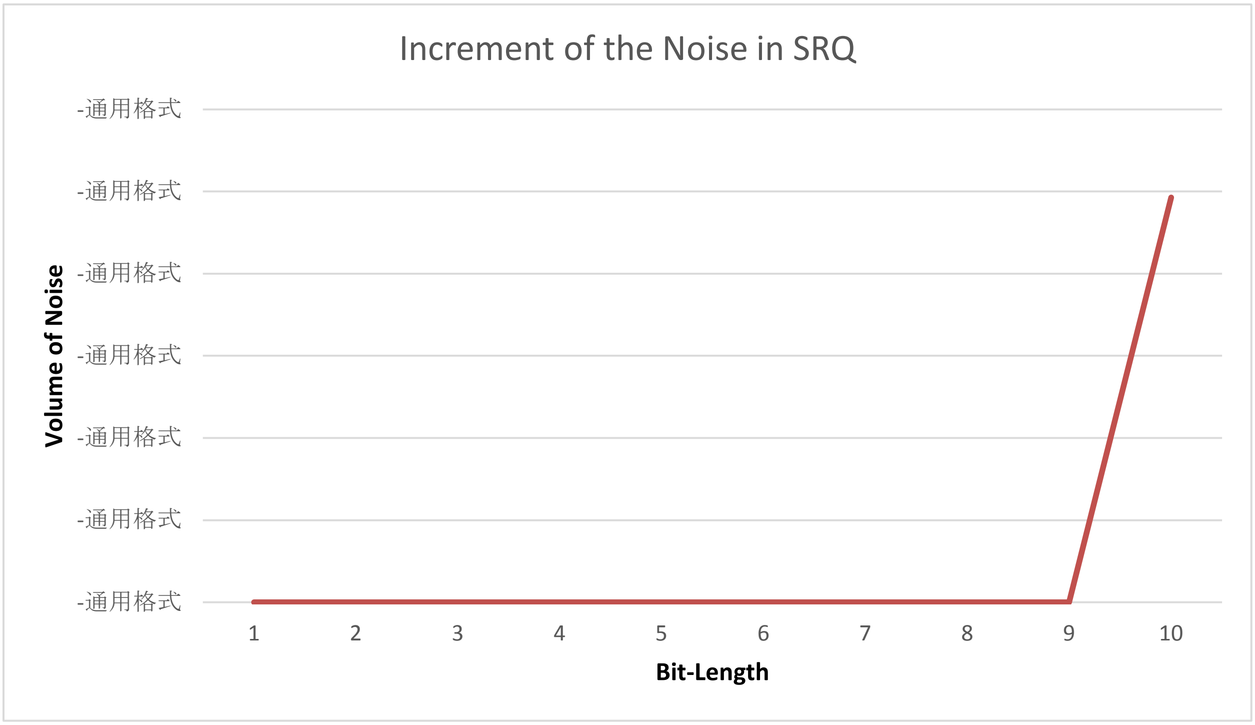

Figure 7: Increment of the Noise in SRQ

4.5 Security Analysis

First of all, the cloud service provider should guarantee the security of the encrypted data which belongs to the data owner that is stored in the cloud. The cloud service provider should not let anyone who does not have the access authorization to the encrypted data to access the encrypted data in any ways. Except for the leakage of the data, the cloud service provider should ensure that the encrypted data that is stored in the cloud from the data owner is not damaged or lost. In the event that the data is damaged or lost, the cloud service provider should have some measures to recover the damaged or lost data. Additionally, there should be some secure channels among the three parties in the cloud environment, which are the cloud service provider, the data owner, and the user, to ensure the communication activities and data exchange among these three parties are secure. However, all the points from above are not the main parts in this security analysis section because the design of the secure range query process hardly can affect the security of these parts. Therefore, the next analysis is done under two assumptions. The first assumption is that there is no physical data leakage, damage or loss of the encrypted data in the cloud. The second assumption is that when the three parties communicate or exchange data with each other, there is a secure channel to ensure their communication activities and data exchanges are secure.

The next part of the security analysis is mainly about the secure range query process. There are two main parts in the security analysis of the secure range query process. First, the decryption attacks from the insiders or the outsiders will be shown. These attacks try to obtain the secret key based on some ciphertext and the public key. Once the true secret key is obtained by the attackers, the data owner will lose all the secrets of the encrypted data that is stored in the cloud. Since defending the attacks from the insiders or outsiders mainly rely on the encryption scheme that is used in the process, the first part of the security analysis is to analyze the security of the somewhat homomorphic encryption scheme that is used in the secure range query process. In the second part, the security of the secure range query process is analyzed. In this part, if there is no useful information leakage to the attackers from inside or outside the cloud service provider and from the users who do not have the authority to access the data will be analyzed.

For the security analysis of the somewhat homomorphic encryption scheme that is used in the secure range query process, the problem is reduced to another problem which is called as the approximate greatest common divisors problem (ACDP) [38]. Before giving the definition of the approximate greatest common divisors problem, the normal greatest common divisors problem will be explained. If there are two given integers a and b, then their greatest common divisor can be found in polynomial time. This is the greatest common divisors problem. A more formal expression is

$$d = \mathrm{gcd}(a, b),$$

where gcd() is a method to find the greatest common divisor of two input integers.

When the value of d is very large, the inputs a and b may have some error. In this case also, their greatest common divisor can be found in polynomial time. This is the approximate greatest common divisors problem. A more formal expression is

$$d = \mathrm{Agcd}(a + a_0, b + b_0),$$

where Agcd() is a method to approximately find the greatest common divisor of two input integers that have some additive error. a_0 and b_0 are the additive error, and there are $|a_0| < A$, and $|b_0| < B$, where A and B are the boundaries of the additive error that this approximate greatest common divisors problem can handle. Their values should be small enough compared to the values of a, b, and d. Generally speaking, the costs and complexity of the approximate greatest common divisors problem should be relatively higher than the regular greatest common divisors problem.

There is a kind of deformation of the approximate greatest common divisors problem that is called as the partially approximate greatest common divisors problem (PACDP). When only one of the two input values has the additive error, the greatest common divisor d still can be found. This problem is the partially approximate greatest common divisors problem. A more formal expression is

$$d = \mathrm{PAgcd}(a + a_0, b),$$

where $\mathrm{PAgcd}()$ is a method to partially approximately find the greatest common divisor of two input integers where only one of them has some additive error. In the expression above, only the input value a has the additive error a_0, and the input value b is accurate. Theoretically, the costs and complexity of the partially approximate greatest common divisors problem should be between the approximate greatest common divisors problem and the regular greatest common divisors problem.

The reason to reduce the security analysis of the somewhat homomorphic encryption scheme to the approximate greatest common divisors problem is that the process that the attacker uses to recover the secret key of the somewhat homomorphic encryption scheme is the process of the approximate greatest common divisors problem. Assume that the information that is held by the attacker is some ciphertext, then he can execute the following process to recover the secret key of the somewhat homomorphic encryption scheme

$$p = \mathrm{Agcd}(c_1, c_2),$$

where p is the secret key of the somewhat homomorphic encryption scheme, and c_1 and c_2 are two ciphertext. This attack process exactly is the process of the approximate greatest common divisors problem.

Assume that the information that is held by the attacker is some ciphertext and the public key, then he can execute the following process to recover the secret key of the somewhat homomorphic encryption scheme.

$$p = \mathrm{PAgcd}(c, N),$$

where p is the secret key and N is the public key of the somewhat homomorphic encryption scheme, and c is a ciphertext. This attack process exactly is the process of the partially approximate greatest common divisors problem. Once the attacker obtains the secret key p, all the data that belongs to the data owner will be leaked to the attacker.

As shown above, the difficulty of the approximate greatest common divisors problem completely reflects the security of a somewhat homomorphic encryption scheme. If the difficulty of solving the approximate greatest common divisors problem is higher, then the security of the somewhat homomorphic encryption scheme is higher. In the approximate greatest common divisors problem, if the gap between the greatest common divisor d and the two input values a and b is larger, then the difficulty of solving the approximate greatest common divisors problem is higher. Since N = pq, and the value that the attacker wants to recover is the secret key p, q affects the gap from the secret key p to the public key N. Therefore, when the value of q is large enough that means the gap from the secret key p to the public key N is large enough, the somewhat homomorphic encryption scheme that is used in the secure range query process is secure, and it is hard to recover the secret key p from some given ciphertext and the public key N. About the specific parameter setting to ensure that q is large enough to ensure the security of the somewhat homomorphic encryption scheme, more details are given in the section 4.6.

Next, if there is no information leakage to the parties who do not have the authority to access the data of the data owner will be analyzed. First of all, in the data outsourcing process, since the data that the data owner sends to the cloud service provider is encrypted, and the bit-lengths of the integers in the database are hidden, the cloud service provider or the outside attacker cannot gain any useful information that can threaten the security of the data of the data owner in this process.

After the data outsourcing process, if there is information leakage in the secure range query process will be analyzed. First, the user who wants to request a range query on the encrypted data needs to send a request message to the cloud service provider. As mentioned above, this request message contains the lower and upper boundaries of the range in the range query, the attributes of the data on which the query is made, and the user ID. Now, assume that this request message is intercepted by the outside attacker. The lower and upper boundaries of the range in the range query are two integers, and they are encrypted with their binary forms bit by bit, and the bit-length is hidden. Therefore, this information is useless to the outside attacker. The attributes of the data on which the query is made is also useless to the outside attacker. The only possibly useful information is the user ID. If the outside attacker intercepts and captures this request message, and sends it to the cloud service provider camouflaging as the user, then once the cloud service provider finishes the secure range query process, and send the results to the outside attacker, the outside attacker still cannot get any useful information, since the results are encrypted with their binary forms bit by bit. Additionally, the cloud service provider can also examine the IP address of the party who sends the query request message. Similarly, the request message is also useless for the cloud service provider to discover the data owner's secrets.

In the secure range query process that is executed by the cloud service provider, all the operations are executed on the encrypted data, and the result of the greater-than algorithm which is produced every time is also encrypted. Therefore, the cloud service provider just executes the secure range query process, and all the middle results are encrypted. The cloud service provider cannot gain any useful information from this process.

After the cloud service provider finishes the secure range query process, he will send the query results back to the user. If these results are intercepted and captured by the outside attacker, as mentioned above that the outside attacker impersonates the user to send the request message to the cloud service provider, the outside attacker cannot do anything with these encrypted results to threaten the security of the data of the data owner.

The user who wants to query some data from the cloud needs to put his user ID into the request message for the authentication from the cloud service provider. After the authentication, the cloud service provider will not execute the secure range query process on the attributes that the user has no authority to access. Therefore, the user cannot access the data that he has no authority to access.

4.6 Controlling the Parameters

There are some parameter settings involved in the secure range query process. These parameters will directly affect the value of the secret key, the value of the public key, the values of the encrypted bits, the volume of the noise that is generated by the homomorphic operations, and the boundaries of the noise that the secure range query process can handle to ensure the correctness of the decryption process. All these parts will affect the efficiency and security of the secure range query process. This section will discuss how to set these parameters, and how these settings will affect the secure range query process. There are three parameters that affect the efficiency and security of the secure range query. They are the security parameter, the range of the secret key generation, and the number of the fake bits that is appended behind each encrypted integer. At the end, how to set these parameters to balance the efficiency and security of the secure range query process will be discussed.

4.6.1 Security Parameter

The security parameter is a parameter in the somewhat homomorphic encryption scheme, and it is also very important for the secure range query process. It is usually denoted with the Greek letter λ. In the somewhat homomorphic encryption scheme, some randomly generated integers from some default ranges are usually used to execute the cryptographic process, such as the generation of the public key, the generation of the secret key, and the generation of the random integer r in the encryption process. This randomness, in some sense, provides much more security for the encryption scheme. Those default ranges for the randomly generations is decided by the security parameter λ.

In the somewhat homomorphic encryption scheme that is used in the secure range query process, three integers will be generated randomly from some default ranges. The first one is

$$p \in [2^{\eta-1}, 2^{\eta}),$$

where p is the secret key, and $\eta = \lambda^2$. This default range is to generate the secret key p, and it can decide the bit-length of the secret key. From the above, η is the bit-length of the secret key, and $\eta = \lambda^2$. Therefore, the bit-length of the secret key p is λ^2. Therefore, the security parameter λ decides the bit-length of the secret key, which means the value of p.

The second one is

$$q \in [2^{\gamma-1}, 2^{\gamma}),$$

where the public key N is q times of the secret key p, since N = pq, and $\gamma = \lambda^5$. This default range is to generate q which is the times from the secret key p to the public key N, and it can decide the bit-length of q. From the above, γ is the bit-length of q, and $\gamma = \lambda^5$. Therefore, the bit-length of q is λ^5. Therefore, the security parameter λ decides the bit-length of q, which means the times from the secret key p to the public key N. Since N = pq, the value of q will affect the value of the public key N partially.

The third one is

$$r \in [0, 2^{\lambda}),$$

where r is the random integer which is used in the encryption process. This default range is to generate r which is the random integer that is used in the encryption process. From the above, λ is the bit-length of the upper boundary for the value of the random integer r. Therefore, the security parameter λ decides, in some sense, the value of the random integer r. Since m + 2r is defined as the noise for each encrypted bit, 2r is almost the whole part of the noise. Therefore, we can say that the security parameter λ will affect the volume of the noise in the secure range query process.

Therefore, the value of the security parameter λ can affect the value of the secret key p, the value of q which is the times from the secret key p to the public key N, the value of the public key N, the value of the random integer r, and then the volume of the noise in the secure range query process. If the value of the security parameter λ increases, then the values from the above will increase relatively. If the value of q increases, then the times from the secret key p to the public key N will increase. This will increase the difficulty of the decryption attack by the outside attacker as mentioned in the security analysis part. Therefore, the security of the somewhat homomorphic encryption scheme will increase. Additionally, the increase in the value of the random integer r also will increase the security of the somewhat homomorphic encryption scheme. However, if the value of the secret key p, the value of the public key N, and the value of the random integer r increase, the value of each encrypted bit will also increases. Thus, the operations in the encryption process, the decryption process, and the secure range query process will have more costs. Sometimes, these operations cannot be done in an efficient time on most computers. This will lower the efficiency of the secure range query process. There is another effect which is on the noise. When the value of the security parameter λ increases, although the volume of the noise will increase, the value of the secret key p will also increase. Due to the gap between the two ranges for generating the secret key p and the random integer r, to be specific, λ and λ^2, the gap between the secret key p and the random integer r will also increases. Therefore, the increase in the value of the security parameter λ will enable the somewhat homomorphic encryption scheme to handle much more volume of the noise.

4.6.2 Secret Key Generation

Another part which is very important to the secure range query process is the default range for generating the secret key p in the somewhat homomorphic encryption scheme. The default range usually is

$$p \in [2^{\eta-1}, 2^{\eta}).$$

As mentioned before, η is the bit-length of the secret key p which is generated within the range above randomly. In general case, $\eta = \lambda^2$ that means λ^2 is the bit-length of the secret key p.

However, when the bit-length of the secret key p is λ^2, sometimes the noise which is generated in the secure range query process probably cannot be handled. As mentioned before, the volume of the noise usually is decided by two parts. The first one is the number of the homomorphic operations that are executed in the secure range query process. The second one is the default range for generating the random integer r in the encryption process. The range, which is $[0, 2^{\lambda})$ generally, is default and usually not changeable. Therefore, the only part that will affect the volume of the noise is the number of homomorphic operations that are executed in the secure range query process. If, in the secure range query process, there are too many homomorphic operations, probably because the integers in the database are too large which implies that the binary forms of the integers have too many bits, too much noise may be generated in the process and will be beyond the maximum noise that the secure range query process can handle. Due to this, the correctness of the decryption process may be lost. However, the volume of noise is hard to decrease, no matter decreasing the range for generating the random integer r, or decreasing the number of homomorphic operations in the secure range query process. Therefore, there is a way that can increase the maximum volume of noise that can be handled in the secure range query process. Changing the default range for generating the secret key p is a way to increase the maximum volume of noise that can be handled in the secure range query process.

Increasing the bit-length of the secret key p can increase the maximum volume of noise that can be handled in the secure range query process. The bit-length of the secret key p is usually λ^2, and the default range for generating the random integer r in the encryption process is $[0, 2^\lambda)$. When the secure range query process is running, more and more homomorphic operations are executed, and then more and more different values of r are accumulated to generate the noise. The association between the bit-length of the secret key p and the security parameter λ can be changed, instead of just increasing the security parameter. If the bit-length of the secret key p is changed to λ^3, then the value of the secret key is increased with the same value of the security parameter λ. Yet, the default range for generating the random integer r in the encryption process is still $[0, 2^\lambda)$. This illustrates that the gap between the secret key p and the value of r which can generate the noise is increased. At this time, if the value of the security parameter λ is increased, more noise can be handled without losing relatively too much efficiency. However, this way will lose some security. Not only the bit-length of the secret key p can be changed to λ^3, it can also be changed to other powers of λ. Details will be given in Section 5.2.

4.6.3 Bit-Length Hiding

Another security arrangement which is a very important parameter in the secure range query process is the number of the fake bits appended behind the encrypted binary forms of the integers. As mentioned before, there are some encrypted 0s that are appended behind the encrypted binary forms of the integers to hide the real bit-length of the integers in the database to avoid the detective attacks from the outsiders.

However, although these fake bits can improve the security of the secure range query process, they also increase the costs and the noise. Recalling the secure greater-than algorithm presented before, when the bit-length of two integers is increased, the volume of noise that is generated by the algorithm will also increase. For every single increase of one bit, the volume of noise will increase greatly. Assume that although the volume of the noise increases a lot, the noise still can be handled with some pretty good parameter settings. However, the increase of the volume of noise will lead much larger values for the homomorphic operations, and much larger final results of the secure range query. This will have a big loss for the computational efficiency and the communication efficiency of the secure range query process. Therefore, the number of the fake bits appended behind the encrypted binary forms of the integers is quite a difficult decision to make. In Section 5.2, the specific value settings for this parameter will be discussed.

4.6.4 Balance between Efficiency and Security

In this part, the balance between the efficiency and the security of the secure range query process will be discussed. This discussion is under an assumption that the noise that is generated in the secure range query process can be handled, since if the volume of noise is beyond the maximum volume that the secure range query process can handle, the final results will not be correct in the decryption process. The efficiency and the security of the secure range query process will lose their meaning.

The efficiency and the security of the secure range query process have an opposite relationship. Generally speaking, if the efficiency increases, the security will decreases and vice versa with some changings of the parameters. In the security parameter part, if the security parameter λ increases, the value of q will increase, and then the times from the secret key p to the public key N will also increase. At the end, the security will increase. However, if the value of the secret key p and the value of the public key N increase, the value of each encrypted bit will also increases and decreases the efficiency. Therefore, the practical situations need to be considered, like the values of the integers in the database, and the settings of other parameters, to decide the value of the security parameter λ.

Then, the default range for generating the secret key p is discussed. Although increasing the bit-length of the secret key p (not increase the value of λ directly) will increase the maximum volume of noise that can be handled in the secure range query process, the security will be lost, since the times from the secret key p to the public key N is decreased. This also is an important parameter setting that need to be balanced.

At the end, the effect that the number of fake bits which are appended behind the encrypted binary forms of the integers to the secure range query process is illustrated. This is a relatively simple problem, since the more fake bits being appended will give more security to the secure range query but lose efficiency. Therefore, a boundary of the number of fake bits need to be decided to ensure greatest security but not to lose unacceptable efficiency in the secure range query process. The parameters which will affect the secure range query process are discussed and how these parameters affect the efficiency and the security of the secure range query process. The specific parameter setting will be given in Section 5.2.

Chapter 5

Implementation and Results

5 Secure Range Query Process

In this chapter, firstly, the secure range query process is compared to the privacy-preserving range query process (PPRQ) from [35] to analyze their computational costs, data transmission costs, communication complexity, and summarize their advantages and disadvantages with each other. Finally, the contributions of the secure range query process can be shown. Then, the implementation of the secure range query is presented. The structure of the program for realizing the secure range query process is shown. Some experiments are designed to study the security and efficiency of the secure range query process with different parameter settings to find some relatively practical parameter settings.

5.1 Comparison with PPRQ

In this section, every sub process of the privacy-preserving range query process and the secure range query process to will be compared. The two processes are compared in three aspects. The first aspect is computational costs. The number of the basic operations on large integers in these two processes is count, which are basically addition, subtraction, multiplication, and division. The results are compared to find out which one has relatively less computational costs than the other. Thus, if the programming environment is the same, the one that has less computational costs will be the one that has a high computational efficiency between these two processes.

The second part in the comparison is the data transmission costs. The size of the data that is transmitted in each communication round among the three parties, which are the cloud service provider, the data owner, and the user, in the cloud environment are compared. If, with the same network data transmission rate, the one that has the less size of the total transmitted data in all the communication rounds will be the process that has a higher data transmission efficiency between the two processes.

The third part is the comparison of the communication complexity of the two processes. In this part, the number of communication rounds around the three parties in the cloud environment will be counted. If the communication activities are frequent and it is hard to determine if the data that is transmitted in each communication round will affect the security of the process, reducing the number of the communication rounds is a good way to prevent the sensitive information leakage. This part is mainly for the privacy-preserving range query process, since the communication complexity of the secure range query process is simplest.

The privacy-preserving range query process and the secure range query process are divided into three processes for comparison. The first process is the encryption process. In this process, the computational costs in the encryption process and the size of the data after the encryption process are compared. The second process is the query process. In this process, the computational costs in the process, the sizes of the final encrypted results, and the communication round complexity are compared. The third process is the decryption process. In this process, the computational costs in the decryption process are compared. At the end, the advantages and disadvantages of the secure range query process compared to the privacy-preserving range query process are finally summarized.

For the convenience of the calculations, let there be n integers in a column in the database, the average bit-length of these integers be b, so the average true value of an integer m is given by

$$\mathrm{m} = \frac{2^{\mathrm{b}-1}+2^{\mathrm{b}}}{2} = 3 \times 2^{\mathrm{b}-2}.$$

Let the number of the fake bits that are appended behind the encrypted binary forms of the integers in the secure range query process be h, and the security parameter be λ.

5.1.1 Encryption Process

The encryption process and data outsourcing process for the privacy-preserving range query process and the secure range query process will be compared in three parts, which are computational efficiency, data transmission efficiency, and communication round complexity.

The encryption scheme that is used in the secure range query process is the somewhat homomorphic encryption scheme, but the encryption scheme that is used in the privacy-preserving range query process is the Paillier cryptosystem, which is an encryption scheme that has some homomorphic properties. The Paillier [25] cryptosystem is as follows:

Key Generation:

$$N = pq,$$

where p and q are two large random prime integers;

$$\lambda' = \mathrm{lcm}(p-1, q-1) = \frac{(p-1)(q-1)}{\gcd(p-1, q-1)},$$

$$g \in Z_{N^2}^{*},$$

where $Z_{N^2}^{*}$ is the set of nonzero integers modulo N^2;

$$L(x) = \frac{x-1}{N},$$

$$\mu = (L(g^{\lambda'} \mathrm{mod} N^2))^{-1} \mathrm{mod} N;$$

Encryption:

$$c = g^m r^N \mathrm{mod} N^2,$$

where m is an integer, and $r \in Z_N^{*}$ that is the set of nonzero integers modulo N;

Decryption:

$$m = L(c^{\lambda'} \bmod N^2)\mu \bmod N.$$

For the security reason, the bit-length of the parameter N usually is 1024, which means

$$2^{1023} \leq N < 2^{1024},$$

so let

$$N = \frac{2^{1023}+2^{1024}}{2} = 3 \times 2^{1022}.$$

For the computational costs, since the average size of the parameter N in Paillier cryptosystem is very large, the basic operations in the encryption process are all on large integers that is the same with the encryption process of the somewhat homomorphic encryption scheme. The computational costs of encrypting a whole column will be calculated including the keys generation costs. There are n integers in the column.

From the above, the computational costs of the keys generation part approximately are

$$\lambda' + O_{gcd}(p-1, q-1),$$

where $O_{gcd}(p-1, q-1)$ is the number of the operations that are executed in $\gcd(p-1, q-1)$.

The computational costs of the encryption part approximately are

$$(3 \times 2^{b-2} + 3 \times 2^{1022})n.$$

Therefore, the computational costs of the Paillier cryptosystem in this process are

$$\left(3 \times 2^{b-2} + 3 \times 2^{1022}\right)n + \lambda' + O_{gcd}(p-1, q-1).$$

The computational costs of the somewhat homomorphic encryption scheme in this process approximately are

$$3(b + h).$$

From the above results, it is obvious to see that the computational costs of the Paillier cryptosystem is far more than the computational costs of the somewhat homomorphic encryption scheme in the encryption process.

For the data transmission costs, since $g \in Z_{N^2}^*$, and g is a nonzero integer,

$$1 \leq g \leq N^2 - 1.$$

Therefore, according to the average value of N, the average value of g is given as

$$g = \frac{N^2-1+1}{2} = \frac{3\times 2^{1023}}{2} = 3 \times 2^{1022}.$$

Since the average value of m is $3 \times 2^{b-2}$, the average value of g^m is

$$g^m = (3 \times 2^{1022})^{3\times 2^{b-2}}.$$

Since $r \in Z_N^*$, and r is a nonzero integer,

$$1 \leq \mathrm{r} \leq \mathrm{N} - 1.$$

Therefore, according to the average value of N, the average value of r is given as

$$\mathrm{r} = \frac{N-1+1}{2} = \frac{3\times2^{1022}}{2} = 3 \times 2^{1021}, \text{ and}$$

the average value of r^{N} is

$$r^N = (3 \times 2^{1021})^{3\times2^{1022}}.$$

Therefore, the size of one encrypted integer in the encryption process of the Paillier cryptosystem is calculated. The size of the encrypted integer is

$$(3 \times 2^{1022})^{3\times2^{\mathrm{b}-2}}(3 \times 2^{1021})^{3\times2^{1022}}\mathrm{mod}(3 \times 2^{1023}).$$

Since the somewhat homomorphic encryption scheme in the secure range query process is to encrypt the bits in the binary forms of the integers, for one encrypted integer, the sizes of all its encrypted bits should be the size of this integer after encryption. According to the default ranges for generating the secret key p, q, and r, the average value of N and 2r are

$$\mathrm{N} = \frac{2^{\lambda^{4}-1}+2^{\lambda^4}-1}{2} \times \frac{2^{\lambda^{5}-1}+2^{\lambda^5}-1}{2} \approx 2^{\lambda^5+\lambda^4-1} + 2^{\lambda^5+\lambda^4-4}, \text{ and}$$

$$2\mathrm{r} = 2\,\frac{2^{\lambda-1}+2^{\lambda}-1}{2} \approx 2^{\lambda} + 2^{\lambda-1}.$$

The formula $\mathrm{m} + 2\mathrm{r} + \mathrm{N}$ is to compute one encrypted bit, and there are b bits in an encrypted integer. Therefore, the size of one encrypted integer in the encryption process of the somewhat homomorphic encryption scheme is

$$\left(2^{\lambda^5+\lambda^4-1} + 2^{\lambda^5+\lambda^4-4} + 2^{\lambda} + 2^{\lambda-1}\right)\mathrm{b}.$$

where the secret key p is randomly generated from the default range $[2^{\lambda^4-1}, 2^{\lambda^4})$ because this secret key generation can be practical for some real data.

Due to the uncertainty of the security parameter and the secret key generation, a program is written to compare the sizes of the two single encrypted integers which are encrypted by these two encryption schemes. The results show that no matter what parameter settings are, the size of one encrypted integer from the somewhat homomorphic encryption scheme is always larger than the size of one encrypted integer from the Paillier cryptosystem. This illustrates that the data transmission cost of the secure range query process is more than the data transmission costs of the privacy-preserving range query process in the data outsourcing process.

Since there is only one communication activity that happens in the encryption process which is the data outsourcing for both range query processes, it is not necessary to compare the communication round complexity in this part.

5.1.2 Query Process

The first step to compare the two secure range query processes is to analyze the query generating process on the user’s side. Since the only communication activity in this process is that the user sends the query request message to the cloud service provider, the two range query processes are the same with respect to the communication complexity. Therefore, only the computational costs and the data transmission costs will be analyzed.

The computational costs in the query generating process are just the costs of encrypting two integers which are the lower and upper boundaries of the range in the query.

Since the computational costs for encrypting one integer of the encryption part in the privacy-preserving range query process approximately are

$$3 \times 2^{b-2} + 3 \times 2^{1022}.$$

Therefore, the computational costs of the query generating process for the privacy-preserving range query process approximately are

$$3 \times 2^{b-1} + 3 \times 2^{1023}.$$

The computational costs for the secure range query process in the query generating process are

$$6(b + h).$$

From the above results, the computational costs of the privacy-preserving range query process is far more than the computational costs of the secure range query process in the query generating process.

In the comparison of the encryption and data outsourcing process, for each encrypted integer, the secure range query process generates the encrypted data files with larger sizes than the privacy-preserving range query process. Therefore, in the query generating process, the secure range query process generates the query files with larger sizes than the privacy-preserving range query process, since the main parts of the query files are the two encrypted integer boundaries of the range in the query. The data transmission costs of the privacy-preserving range query process are less than the data transmission costs of the secure range query process in the query generating process.

Compared to the secure range query process, the privacy-preserving range query process is much more complicated. There are three sub processes in the query process of the privacy-preserving range query process. They are the secure bit decomposition process, secure comparison process, and the privacy-preserving range query process. The total costs for the privacy-preserving range query process consists of the sum of the computational costs and the data transmission costs of the three sub processes.

In the cloud environment of the privacy-preserving range query process, there are two cloud service providers. The first one is the primary cloud service provider, and the second one is the secondary cloud service provider. The primary cloud finishes the secure bit decomposition process by communicating with the secondary cloud. Since the parties that should be cared most is the user and the data owner, the data transmission costs of the communication rounds between the two clouds will not be counted.

In the secure bit decomposition process of the privacy-preserving range query process, the two clouds work together to convert the encrypted integers to the encrypted binary forms wit encrypted bits through communicating with each other securely. The communication details between the two clouds are quite complicated, and can be found in [35]. Based on the secure bit decomposition process, the computational costs of the primary cloud are

$$(2\lambda' + 3 \times 2^{1023} + 26)\mathrm{b}(\mathrm{n} + 2), \text{ and}$$

the computational costs of the secondary cloud are

$$\left(3 \times 2^{b-1} + 9 \times 2^{1023} + 3 \times 2^{1020} + 18 + \frac{3}{2}\right)\mathrm{b}(\mathrm{n} + 2).$$

The secure comparison process and the range query process are considered together. In this part, there is no need to count the computational costs, since just the other computational cost is already much larger than the computational costs of the query process of the secure range query process. Denote the computational costs of the primary cloud as A_1, and the computational costs of the secondary cloud as A_2 in this part.

Therefore, the total computational costs of the primary cloud in the query process are

$$(2\lambda' + 3 \times 2^{1023} + 26)\mathrm{b}(\mathrm{n} + 2) + \mathrm{A}_1, \text{ and}$$

the total computational costs of the secondary cloud in the query process are

$$\left(3 \times 2^{b-1} + 9 \times 2^{1023} + 3 \times 2^{1020} + 18 + \frac{3}{2}\right) b(n+2) + A_2.$$

The computational costs of the secure range query process in the query process are

$$36(b + bk)^2 n + 2(b + h)n.$$

Form the above results, it is obvious that no matter the computational costs of the primary cloud or the secondary cloud, they are all far more than the computational costs of the secure range query process in the query process.

In the comparison of the encryption and data outsourcing process, for each encrypted integer, the secure range query process generates the encrypted data files with larger sizes than the privacy-preserving range query process. Also, the encrypted query results in the secure range query process are equivalent to encrypt the integers more times, so the sizes of them will also increase more times, but in the privacy-preserving range query process, the encrypted query results are just equivalent to encrypt the integers one time.

Therefore, for the data transmission costs, it is obvious that the data transmission costs of the privacy-preserving range query process is less than the data transmission costs of the secure range query process in the query process.

Since there is no communication round in the query process of the secure range query process, and there are some communication rounds between the primary cloud and the secondary cloud in the secure comparison process of the privacy-preserving range query process, the secure range query process has less communication round complexity than the privacy-preserving range query process.

5.1.3 Decryption Process

In the decryption process, only the computational costs of the privacy preserving range query process and the secure range query process will be considered, since all the decrypting operations happens on the user's side, there is no data transmission and communication round in this process.

Assume that there are R encrypted values in the results of the range query. For the privacy-preserving range query process, the computational costs of the decryption process are

$$(\lambda' + 9)R.$$

For the secure range query process, the computational costs of the decryption process are

$$6(b + h)R.$$

Since the parameter $\lambda' = \mathrm{lcm}(p - 1, q - 1)$, and p and q are two large prime integers in the Paillier cryptosystem, the computational costs of the privacy-preserving range query process is far more than the computational costs of the secure range query process in the decryption process.

5.1.4 Comparison between PPRQ and SRQ

After all the comparisons in the previous subsections, the advantages and disadvantages of the secure range query process compared to the privacy-preserving range query process will be summarized and listed.

Advantages:

1. The computational efficiency of the secure range query process is higher than that of the privacy-preserving range query process in the encryption process;
2. There is only one cloud service provider that is involved in the cloud environment in the secure range query process. Therefore, there is no any communication round complexity in the secure range query process compared to the privacy-preserving range query process;
3. The computational efficiency of the secure range query process is higher than that of the privacy-preserving range query process in the query generating process;
4. Fake bits are not appended behind the encrypted integers to hide the real bit-length of the integers in the privacy-preserving range query process, since the Paillier cryptosystem encrypts the integers directly. Therefore, the secure range query process is more secure than the privacy-preserving range query process in this part;

5. The computational efficiency of the secure range query process is higher than that of the privacy-preserving range query process in the decryption process;
6. In the secure range query process, no middle results will be known by the cloud service provider in the query process. The primary cloud in the privacy-preserving range query process knows all the middle results of the query process. Therefore, the secure range query process is securer than the privacy-preserving range query process in this part;
7. The computational efficiency of the secure range query process is higher than that of the privacy-preserving range query process in the query process.

Disadvantages:

1. The original encrypted data in the secure range query process is larger than the original encrypted data in the privacy-preserving range query process;
2. The query request file in the secure range query process is larger than the query request file in the privacy-preserving range query process;
3. For the purpose of controlling the noise, the bit-length of the secret key p is increased, not just increase the value of the security parameter λ, but change the mathematical relation between the parameter η and λ in the somewhat homomorphic encryption scheme. This will decrease the gap between the secret key p and the public key N, which leads the loss in the security of the process. There is no such problem in the privacy-preserving range query process, since it

uses a very large parameter N in the encryption process of the Paillier cryptosystem;

4. Since the cloud service provider does not know the middle results in the query process, the final query results can be quite large. This loses some data transmission efficiency compared to the privacy-preserving range query process.

In conclusion, the secure range query process is faster, but generates larger files. However, the privacy-preserving range query process is slower, but generates smaller files.

5.2 Experiments & Results

In this section, some experiments are designed to test the practicability of the secure range query process, and the results of the experiments are illustrated. First, some tables are presented to show the security parameter λ that can enable the secure range query process for different values of the integers in the database with the different secret key generations. Then, some relatively practical parameter settings are chosen to run the secure range query process with some datasets whose values are practical. Finally, the practicability of the secure range query process is discussed.

In [35], if the maximum value of the integers that the secure data range query process can handle is 2^{20}, then this secure range query can be used in most practical applications. For covering more practical applications, the largest integer in the tables is set to be 2^{64}. Although some practical applications may involve the integers whose values are 2^{128}, but this value is too large to handle by the secure range query process, since it will lose efficiency. The ability to handle integers whose values are 2^{64} is enough for the secure range query process to have a considerable degree of practicability. Tables 2 to 4 show the settings of the security parameters for three different secret key generations with $p \in [2^{\lambda^4-1}, 2^{\lambda^4})$, $p \in \left[2^{\lambda^5-1}, 2^{\lambda^5}\right)$, and $p \in [2^{\lambda^6-1}, 2^{\lambda^6})$. Since it is hard to calculate all the security parameters for the integers from 2 to 2^{64} with different secret key generations, the following tables are generated by a computer program.

Table 2: Security Parameters for First Secret Key Generation

bit-length	λ	bit-length	λ	bit-length	λ	bit-length	λ	bit-length	λ	bit-length	λ
1	3	12	8	23	11	34	14	45	17	56	19
2	3	13	8	24	11	35	15	46	17	57	20
3	4	14	8	25	12	36	15	47	17	58	20
4	4	15	9	26	12	37	15	48	18	59	20
5	5	16	9	27	12	38	15	49	18	60	20
6	5	17	9	28	13	39	16	50	18	61	21
7	6	18	10	29	13	40	16	51	18	62	21
8	6	19	10	30	13	41	16	52	19	63	21
9	6	20	10	31	13	42	16	53	19	64	21
10	7	21	11	32	14	43	16	54	19		
11	7	22	11	33	14	44	17	55	19		

Table 2 shows the security parameters λ for the integers from 2 to 2^{64} for the secret key generation $p \in [2^{\lambda^4-1}, 2^{\lambda^4})$.

Table 3: Security Parameters for Second Secret Key Generation

bit-length	λ	bit-length	λ	bit-length	λ	bit-length	λ	bit-length	λ	bit-length	λ
1	2	12	5	23	7	34	8	45	9	56	10
2	3	13	5	24	7	35	8	46	9	57	10
3	3	14	5	25	7	36	8	47	9	58	10
4	3	15	5	26	7	37	8	48	9	59	10
5	3	16	6	27	7	38	8	49	9	60	10
6	4	17	6	28	7	39	8	50	9	61	10
7	4	18	6	29	7	40	8	51	9	62	10
8	4	19	6	30	7	41	9	52	9	63	10
9	4	20	6	31	8	42	9	53	10	64	10
10	5	21	6	32	8	43	9	54	10		
11	5	22	6	33	8	44	9	55	10		

Table 3 shows the security parameters λ for the integers from 2 to 2^{64} for the secret key generation of $p \in [2^{\lambda^5-1}, 2^{\lambda^5})$.

Table 4: Security Parameters for Third Secret Key Generation

bit-length	λ	bit-length	λ	bit-length	λ	bit-length	λ	bit-length	λ	bit-length	λ
1	2	12	4	23	5	34	6	45	6	56	7
2	2	13	4	24	5	35	6	46	6	57	7
3	3	14	4	25	5	36	6	47	6	58	7
4	3	15	4	26	5	37	6	48	6	59	7
5	3	16	4	27	5	38	6	49	6	60	7
6	3	17	4	28	5	39	6	50	6	61	7
7	3	18	4	29	5	40	6	51	6	62	7
8	3	19	5	30	5	41	6	52	6	63	7
9	4	20	5	31	5	42	6	53	6	64	7
10	4	21	5	32	5	43	6	54	6		
11	4	22	5	33	5	44	6	55	7		

Table 4 shows the security parameters λ for the integers from 2 to 2^{64} for the secret key generation of $p \in [2^{\lambda^6-1}, 2^{\lambda^6})$.

In Table 2, when $p \in [2^{\lambda^4 - 1}, 2^{\lambda^4})$, if the maximum value of the integers in the database is 2^{64} and the secure range query process for this size integers need to be enabled, the value of the security parameter λ should be 21. That will generate really large encrypted data and middle results in the encryption process and the query process. Therefore, the secret key generation $p \in [2^{\lambda^4 - 1}, 2^{\lambda^4})$ is not practical for the integers that large like 2^{64} because it will not be efficient. However, for the integers whose values are 2^{20}, the value of the security parameter λ is just 10, which will be enough to handle the noise problem in the secure range query process, and will also be pretty efficient.

When $p \in [2^{\lambda^5 - 1}, 2^{\lambda^5})$, the most exciting results are obtained. In Table 3, if the maximum value of the integers in the database is 2^{64}, and the secure range query process with integers that are large like this need to be enabled, the value of the security parameter λ should be 10. This is quite less than the value in Table 2, since each addition of the security parameter will increase a lot of costs. This parameter setting has already been pretty practical. For the integers whose values are 2^{20}, the value of the security parameter λ should only be 6 that will be enough to handle the noise problem in the secure range query process and will also be pretty efficient.

When $p \in [2^{\lambda^6 - 1}, 2^{\lambda^6})$, much security will be lost, since the value of p is more than the value of q. This will be not secure for the outside attacks. In Table 4, if the maximum value of the integers in the database is 2^{64}, and the secure range query process need to be enabled with the integers that are large like this, the value of the security parameter λ should only be 7. This is a lot less than the value in Tables 2 and 3. This parameter setting is very practical, if the security loss is not considered. For the integers whose values are 2^{20}, the value of the security parameter λ should only be 5 that will be enough to handle the noise problem in the secure range query process, and be very efficient. For the problem that the value of p is more than the value of q, it is not useful to increase the value of q to solve it because this will decrease the efficiency again. Therefore, this parameter setting is not practical in security.

After the comparison among the three tables, if the integers whose values are 2^{20} need to be handled, the secret key generation should in Table 2 be used. If the integers whose values are 2^{64} need to be handled, the secret key generation in Table 3 should be used, since the value of the parameter can balance the efficiency and security well in the secure range query process. This setting will also lose some security, but it will gain more benefits in efficiency to make the secure range query to be practical.

Now, some experiments will be presented. Two experiments are executed with two different parameter settings. The two values of the integers in the database that is most practical to handle are 2^{20} and 2^{64}, since 2^{20} can fit most applications in the practical world, and 2^{64} can handle almost all the practical applications, but will sacrifice some security. Therefore, these two most practical parameter settings will be chosen that can separately handle the 2^{20} integers and 2^{64} integers to do the experiments. In the practical applications, the data owner can choose these two parameter settings based on the features of their databases and their needs.

The experimental programs are written in Java programming language. Since in the secure range query process, there are a lot of operations that are with big integers, a Java class which is called as BigInteger is used from the Java library to handle large integers involved in the process. The structure of the program is given in Figure 8

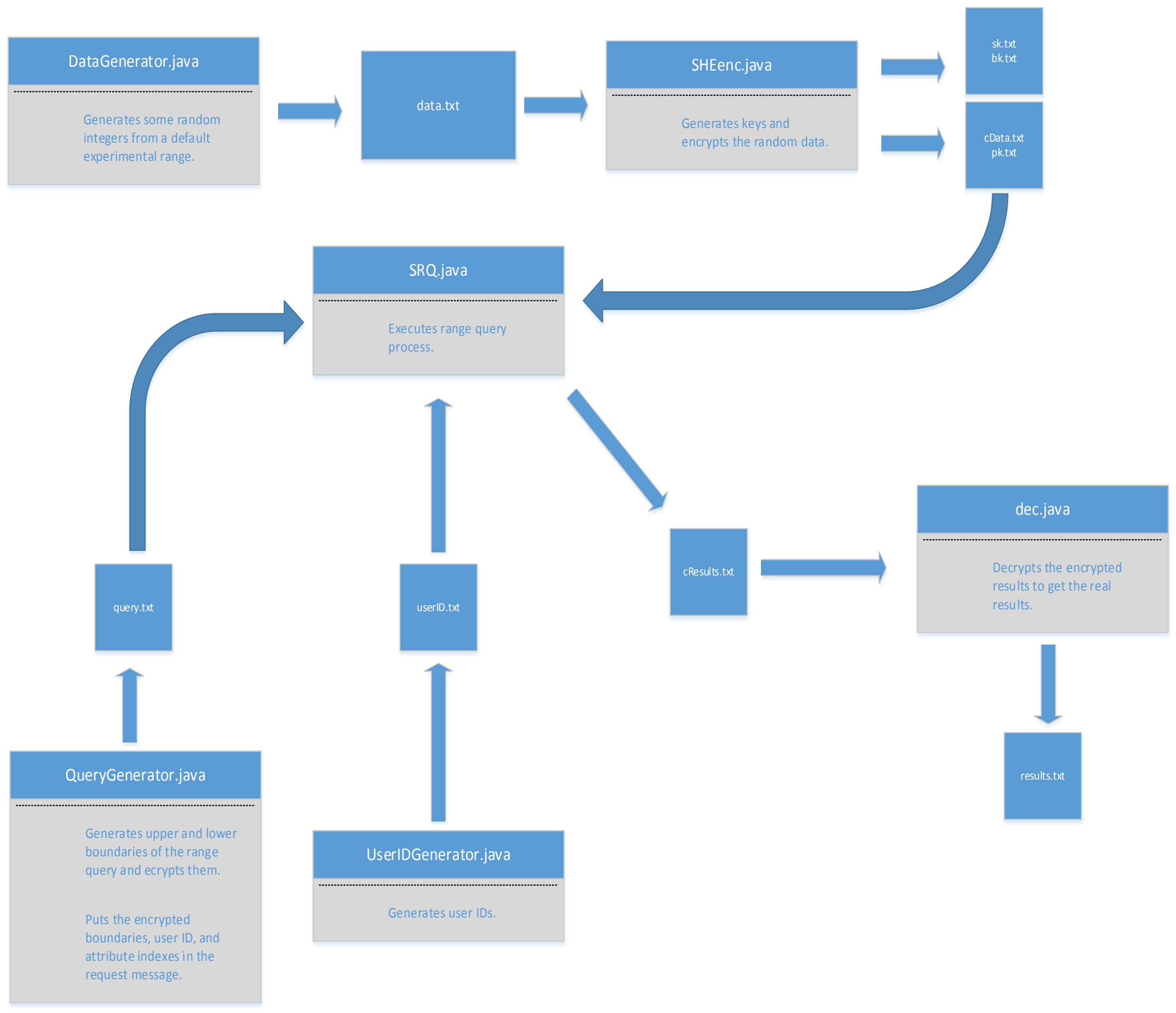
DataGenerator.java
Generates some random integers from a default experimental range.
data.txt
SHEenc.java
Generates keys and encrypts the random data.
sk.txt
bk.txt
cData.txt
pk.txt
SRQ.java
Executes range query process.
query.txt
userID.txt
cResults.txt
dec.java
Decrypts the encrypted results to get the real results.
results.txt
QueryGenerator.java
Generates upper and lower boundaries of the range query and ecrypts them.
Puts the encrypted boundaries, user ID, and attribute indexes in the request message.
UserIDGenerator.java
Generates user IDs.

Figure 8: Experimental Program Structure

In the experiments, a cloud environment is simulated. Since most operations that need to be done on the side of the cloud service provide, a cloud service provider is chosen who can supply more powerful computational ability than the local computer as the cloud in the experiments. The cloud service provider that is chosen is SHARCNET.

SHARCNET [39] is a consortium of Canadian academic institutions who share a network of high performance computers to support worldwide academic researches with its high computational ability. This is a perfect place for the experiments. goblin.sharcnet.ca is used which is a contributed gigabit Ethernet cluster from the SHARCNET. A system with an Intel Xeon CPU 2.53 GHz 16 cores processor and 12GB memory is used in the experiments. PuTTY, which is a free terminal emulator, serial console and network file transfer application is used to control the operations on cloud’s side from the local side, and WinSCP which is a free SFTP, SCP and FTP client for Microsoft Windows is used to organize the files on the cloud’s side from the local side. The command line that is used to run the program on the cloud’s side is “sqsub –r 15h –o results.txt java –Xmx512m NAME”. The systems in SHARCNET totally support the Java programs [40].

For the sides of the data owner and the user, a laptop iss used with an Intel(R) Core(TM) i7-3610QM CPU @ 2.30GHz and 4GB RAM in Microsoft Windows 8.1 operating system. This laptop is used as both the data owner and the user, since the data transmission costs between these two parties can be ignored.

Two parameter settings for the experiments are used. For the first experiment, the secret key generation is set to be $p \in [2^{\lambda^4-1}, 2^{\lambda^4})$, since this generation can handle the 2^{20} integers without too large security parameter λ. Since this setting is for the first experiment that is with the 2^{20} integers, the integers that are generated randomly are all from $[2^{19}, 2^{20}]$. The security parameter λ is set as 10 since this is the least value to handle the 2^{20} integers with $p \in [2^{\lambda^4-1}, 2^{\lambda^4})$. For the second experiment, the secret key generation was set to be $p \in [2^{\lambda^5-1}, 2^{\lambda^5})$, since this generation is to handle the relatively largest data set which is the 2^{64} integers also without too large security parameter λ. Since this setting is for the second experiment with 2^{64} integers, the integers that are generated randomly are all from $[2^{63}, 2^{64}]$. The security parameter λ was set as 10 since this is the least value to handle 2^{64} integers with $p \in [2^{\lambda^5-1}, 2^{\lambda^5})$. There are also some other settings for the experiments. There are 1000 integers in the experimental database, which contains ten attributes, and each attribute having 100 integers. In the experimental range query process, 5 attributes are randomly selected to query, so that 500 integers could be queried.

In the experiments, the results of the computational time and the size of the files that are transmitted in each communication round are separately recorded. To be specific, for the computational time, the encryption time, the query generating time, the secure range query process time, and the decryption time are recorded and analyzed separately. For the size of files that are transmitted in each communication round, the encrypted database for data outsourcing, the request message that is sent to the cloud service provider to request a range query, the encrypted results that are sent back to the user are recorded and analyzed separately.

Table 5: Experimental Results of Running Time for First Secret Key Generation

Encryption Time	Query Generating Time	SRQ Time	Decryption Time
5107.77s	5.55ms	186.56m	221.58s
5216.69s	5.64ms	181.64m	213.46s
5172.5s	5.65ms	193.54m	218.46s
5106.03s	5.51ms	191.42m	224.23s
5120.09s	5.38ms	185.41m	214.26s
4946.51s	5.59ms	193.84m	225.34s
5205.27s	5.39ms	185.81m	227.13s
5118.45s	5.54ms	189.12m	226.11s
5217.14s	5.52ms	187.84m	224.71s
5191.65s	5.62ms	192.88m	224.89s

4956.65s	5.61ms	183.21m	221.58s
5013.49s	5.65ms	184.74m	226.05s
5108.51s	5.64ms	184.23m	224.46s
5040.51s	5.74ms	191.95m	217.48s
5113.57s	5.47ms	184.74m	228.05s
5194.49s	5.58ms	190.24m	223.8s
5305.49s	5.6ms	191.48m	215.05s
4997.79s	5.55ms	192.85m	224.27s
4950.84s	5.74ms	189.28m	222.62s
5005.25s	5.69ms	191.17m	229.55s

Table 6: Experimental Results of Transmitted File Sizes for First Secret Key Generation

Encrypted Data	Request Message	Query Results
645.18mb	1320kb	1.74gb
667.94mb	1318kb	1.7gb
664.04mb	1355kb	1.73gb
662.38mb	1328kb	1.8gb
644.52mb	1267kb	1.74gb
639.21mb	1294kb	1.68gb
653.51mb	1341kb	1.77gb
663.08mb	1357kb	1.7gb
634.44mb	1358kb	1.73gb
654.78mb	1312kb	1.76gb
625.58mb	1328kb	1.71gb
656.14mb	1364kb	1.79gb
633.71mb	1337kb	1.73gb

621.91mb	1339kb	1.72gb
664.52mb	1288kb	1.75gb
654.53mb	1360kb	1.74gb
650.44mb	1304kb	1.8gb
633.37mb	1341kb	1.68gb
630.98mb	1366kb	1.76gb
642.37mb	1314kb	1.76gb

Table 7: Experimental Results of Running Time for Second Secret Key Generation

Encryption Time	Query Generating Time	SRQ Time	Decryption Time
6823.14s	177.04ms	633.98m	649.35s
6760.82s	172.97ms	651.29m	656.57s
6698.66s	181.85ms	623.21m	652.93s
6914.91s	182.55ms	608.86m	649.86s
6849.26s	170.41ms	626.33m	632.02s

6718.06s	180.58ms	623.68m	629.25s
6796.38s	178.78ms	643.41m	664.61s
7013.53s	177.16ms	650.74m	664.31s
6892.76s	175.69ms	627.98m	638.27s
6720.48s	180.97ms	636.71m	673.06s
7077.81s	180.92ms	625.75m	625.36s
6599.13s	177.31ms	611.54m	659.27s
6952.6s	172.11ms	609.92m	659.25s
6555.34s	179.07ms	619.36m	672.1s
6764.65s	178.46ms	630.74m	648.01s
6900.83s	178.69ms	618.93m	626.62s
6630.51s	180.19ms	652.5m	627.72s
6822.08s	181.95ms	644.96m	657.89s
6695.62s	173.82ms	657.63m	667.57s
6783.45s	174.73ms	650.45m	665.7s

Table 8: Experimental Results of Transmitted File Sizes for Second Secret Key Generation

Encrypted Data	Request Message	Query Results
3375.05mb	6912kb	10.13gb
3246.57mb	6989kb	10.33gb
3273.86mb	6963kb	9.92gb
3449.43mb	6749kb	9.85gb
3311.59mb	7025kb	9.76gb
3247.79mb	6673kb	9.8gb
3480.62mb	7126kb	10.14gb
3457.75mb	6866kb	9.82gb
3504.05mb	6664kb	10.41gb
3290.44mb	6925kb	10.45gb
3465.48mb	6720kb	9.73gb
3462.19mb	6950kb	10.38gb

3438.18mb	6747kb	9.94gb
3345.21mb	6867kb	10.34gb
3380.74mb	6725kb	9.94gb
3470.17mb	6815kb	9.8gb
3354.28mb	6654kb	10.52gb
3256.56mb	6715kb	9.92gb
3403.15mb	6822kb	10.18gb
3318.17mb	6712kb	9.83gb

For the computational time analysis, from Table 5, the encryption time is around 5100s. The keys generation process takes the most of this time, especially the public key generation process. Once the keys generation process is finished, the encryption process does not take much time. This is also the reason why the query generating time is so little that is just around 5.5ms because the query generating process is the process to encrypt two boundaries of the range query. The time of the secure range query process is relatively long that is around 185 minutes, but it happens on the cloud's side. Therefore, it is still practical for using, since the decryption time is just around 220s. Thus, it is still efficient for the user, since the part that should be cared about most is the data owner and the user. From Table 7, the results for $p \in [2^{\lambda^5-1}, 2^{\lambda^5})$ are very similar. The encryption time is around 6800s. Since this is a one-time process, it is acceptable, and this is for the 2^{64} integers. Still, the keys generation process takes the most of this time, especially the public key generation process. The query generating time is just around 170ms. The time of the secure range query process is relatively long that is around 630m, and the decryption time is around 650s. In general, for the integer data that is as large as 2^{64}, the efficiency of the secure range query process is still acceptable.

For the data transmission costs analysis, from Table 6, the size of the whole encrypted data is around 650 MB. The size of the request message is very small which is around 1300 KB. The size of the encrypted query results is pretty large which is around 1.7 GB. Although this size is large, with a good network that has a high data transmission rate, this size still can work. From Table 8, the results for $p \in [2^{\lambda^5-1}, 2^{\lambda^5})$ are larger. The size of the whole encrypted data is around 3350 MB. The size of the request message is larger which is around 6900 KB, but it is still not very large, so it does not affect the process. The size of the encrypted query results is pretty large which is around 10 GB. This size is very large. However, the values of the integer data in the experiment are around 2^{64}, so this is necessary to sacrifice some efficiency and security for handling larger integers in the secure range query process.

For setting the number of fake bits that are appended to the encrypted binary forms of the integers, the value of one-half of the average bit-length of the integers in the dataset is a very secure way. However, this will lead much more costs. This number is set with one third, one fourth, or even some smaller proportion of the average bit-length of the integers in the dataset. This needs to be decided by the data owner and according to his practical security requirements.

In conclusion, when the secure range query process is with 2^{20} integers, it is already practical for most applications. Although, with the 2^{64} integers, the secure range query process seems to be slow and generate large result files, it is still successful to handle the larger numbers. No matter the $\mathrm{p} \in [2^{\lambda^4-1}, 2^{\lambda^4})$ or $\mathrm{p} \in [2^{\lambda^5-1}, 2^{\lambda^5})$, they all sacrifice some security to trade for some efficiency when the loss of the security is acceptable and worth for. Actually, when the data owner companies or enterprises have more computational power and higher data transmission rate, they can try to use even smaller secret key generations with some larger security parameter that can enable the secure range query process, and ensure the efficiency and security.

Chapter 6

Conclusions and Future Work

6 Conclusions

In this thesis, a secure range query process was proposed which is based on a somewhat homomorphic encryption scheme for a cloud environment. This encryption scheme can supply the homomorphic properties on some operations with binary encrypted integers. If there is an effective way to control the noise that is generated in the process, this encryption scheme can handle all the homomorphic operations, and it will turn to a fully homomorphic encryption scheme. All the integers are encrypted in their binary forms bit by bit. Then, a greater-than algorithm is used to compare two encrypted integers through all their bits. The results of the greater-than algorithm are not revealed to the cloud service provider. Different parameter settings are discussed and some settings that can make the secure range query process to be used in practical applications are found. The secure range query process is compared to the privacy-preserving range query process which is proposed in [35], and the advantages and disadvantages of the secure range query process compared to the privacy-preserving range query process are enumerated. In the experiments, two relatively practical parameter settings are proposed for 2^{20} and 2^{64}

size integers. From the results of the experiments, if sacrifice some efficiency or security, or both, the secure range query process can be more functional.

6.1 Future Work

One of the future works of this thesis is to find a better secure comparison method as the results of the query are still very large. Another improvable part is that the user access control in this thesis cannot prevent the collusion between two users with different data access authorization. Some methods that are used in the secure range query process are also probably useful for some other kinds of query processes. This will contribute towards realizing the fully database as a service (DAAS) in a cloud environment. However, further researches about the fully homomorphic encryption scheme still will continue, since the fully homomorphic encryption scheme indeed is a powerful way to realize fully database as a service in a cloud environment.

References

[1] Armbrust M, Fox A, Griffith R, et al. A view of cloud computing[J]. Communications of the ACM, 2010, 53(4): 50-58.

[2] Buyya R, Yeo C S, Venugopal S, et al. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility[J]. Future Generation computer systems, 2009, 25(6): 599-616.

[3] Marston S, Li Z, Bandyopadhyay S, et al. Cloud computing—The business perspective[J]. Decision Support Systems, 2011, 51(1): 176-189.

[4] Hofmann P, Woods D. Cloud computing: the limits of public clouds for business applications[J]. Internet Computing, IEEE, 2010, 14(6): 90-93.

[5] Weinhardt C, Anandasivam D I W A, Blau B, et al. Cloud computing–a classification, business models, and research directions[J]. Business & Information Systems Engineering, 2009, 1(5): 391-399.

[6] Keahey K. Cloud Computing for Science[C]. SSDBM. 2009: 478.

[7] Hacigumus H, Iyer B, Mehrotra S. Providing database as a service[C]. Data Engineering, 2002. Proceedings. 18th International Conference on. IEEE, 2002: 29-38.

[8] Curino C, Jones E P C, Popa R A, et al. Relational cloud: A database-as-a-service for the cloud[J]. 2011.

[9] Mykletun E, Tsudik G. Aggregation queries in the database-as-a-service model[M]. Data and Applications Security XX. Springer Berlin Heidelberg, 2006: 89-103.

[10] Junzhou Luo, Jiahui Jin, Aibo Song, Cloud Computing: Architecture and Key Technology[J]. Journal on Communications, 2011, 32(7): 3-21.

[11] Tsai W T, Sun X, Balasooriya J. Service-oriented cloud computing architecture[C]. Information Technology: New Generations (ITNG), 2010 Seventh International Conference on. IEEE, 2010: 684-689.

[12] Ostermann S, Iosup A, Yigitbasi N, et al. A performance analysis of EC2 cloud computing services for scientific computing[M]. Cloud computing. Springer Berlin Heidelberg, 2010: 115-131.

[13] Kandukuri B R, Paturi V R, Rakshit A. Cloud security issues[C]. Services Computing, 2009. SCC'09. IEEE International Conference on. IEEE, 2009: 517-520.

[14] Ramgovind S, Eloff M M, Smith E. The management of security in cloud computing[C]. Information Security for South Africa (ISSA), 2010. IEEE, 2010: 1-7.

[15] Bisong A, Rahman M. An overview of the security concerns in enterprise cloud computing[J]. arXiv preprint arXiv:1101.5613, 2011.

[16] Chen Y, Paxson V, Katz R H. What’s new about cloud computing security[J]. University of California, Berkeley Report No. UCB/EECS-2010-5 January, 2010, 20(2010): 2010-5.

[17] Keefe T F, Thuraisingham M B, Tsai W T. Secure query-processing strategies[J]. Computer, 1989, 22(3): 63-70.

[18] Wang S, Agrawal D, El Abbadi A. A comprehensive framework for secure query processing on relational data in the cloud[M]. Secure Data Management. Springer Berlin Heidelberg, 2011: 52-69.

[19] Takabi H, Joshi J B D, Ahn G J. Security and privacy challenges in cloud computing environments[J]. IEEE Security and Privacy, 2010, 8(6): 24-31.

[20] Shi E, Bethencourt J, Chan T H H, et al. Multi-dimensional range query over encrypted data[C]. Security and Privacy, 2007. SP'07. IEEE Symposium on. IEEE, 2007: 350-364.

[21] Gentry C. A fully homomorphic encryption scheme[D]. Stanford University, 2009.

[22] Van Dijk M, Gentry C, Halevi S, et al. Fully homomorphic encryption over the integers[M]. Advances in cryptology–EUROCRYPT 2010. Springer Berlin Heidelberg, 2010: 24-43.

[23] Dianhua Tang, Shixiong Zhu, Yunfei Cao. A Fast Fully Homomorphic Encryption Scheme over Integers[J]. Computer Engineering and Application, 2012, 48(28): 117-122.

[24] Rivest R L, Shamir A, Adleman L. A method for obtaining digital signatures and public-key cryptosystems[J]. Communications of the ACM, 1978, 21(2): 120-126.

[25] Paillier P. Public-key cryptosystems based on composite degree residuosity classes[C]. Advances in cryptology—EUROCRYPT’99. Springer Berlin Heidelberg, 1999: 223-238.

[26] Okamoto T, Uchiyama S. A new public-key cryptosystem as secure as factoring[M].

Advances in Cryptology—EUROCRYPT'98. Springer Berlin Heidelberg, 1998: 308-318.

[27] Naehrig M, Lauter K, Vaikuntanathan V. Can homomorphic encryption be practical?[C]. Proceedings of the 3rd ACM workshop on Cloud computing security workshop. ACM, 2011: 113-124.

[28] Yacine Ichibane, Youssef Gahi, Zouhair Guennoun, Mouhcine Guennoun. Private Video Streaming Service Using Leveled Somewhat Homomorphic Encryption. School of Electrical Engineering and Computer Science University of Ottawa, 2014.

[29] Hirt M, Sako K. Efficient receipt-free voting based on homomorphic encryption[C]. Advances in Cryptology—EUROCRYPT 2000. Springer Berlin Heidelberg, 2000: 539-556.

[30] Aggarwal G, Bawa M, Ganesan P, et al. Two can keep a secret: A distributed architecture for secure database services[J]. CIDR 2005, 2005.

[31] Hacigümüş H, Iyer B, Li C, et al. Executing SQL over encrypted data in the database-service-provider model[C]. Proceedings of the 2002 ACM SIGMOD international conference on Management of data. ACM, 2002: 216-227.

[32] Yao A C. Protocols for secure computations[C]. 2013 IEEE 54th Annual Symposium on Foundations of Computer Science. IEEE, 1982: 160-164.

[33] Samanthula B K, Elmehdwi Y, Howser G, et al. A secure data sharing and query processing framework via federation of cloud computing[J]. Information Systems,

2015, 48: 196-212.

[34] Blake I F, Kolesnikov V. One-round secure comparison of integers[J]. Journal of Mathematical Cryptology, 2009, 3(1): 37-68.

[35] Samanthula B K, Jiang W. Efficient privacy-preserving range queries over encrypted data in cloud computing[C]. Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on. IEEE, 2013: 51-58.

[36] Mani M, Shah K, Gunda M. Enabling secure database as a service using fully homomorphic encryption: Challenges and opportunities[J]. arXiv preprint arXiv:1302.2654, 2013.

[37] http://docs.oracle.com/javase/7/docs/api/java/math/BigInteger.html

[38] Howgrave-Graham N. Approximate integer common divisors[M]. Cryptography and Lattices. Springer Berlin Heidelberg, 2001: 51-66.

[39] https://www.sharcnet.ca/help/index.php/Getting_Started_with_SHARCNET

[40] https://www.sharcnet.ca/help/index.php/OPENJDK

Appendix A

Somewhat Homomorphic Encryption Scheme

```
import java.math.BigInteger;
import java.util.Random;
import java.util.Scanner;
import java.io.File;
import java.io.PrintWriter;
import java.io.IOException;

public class SHEenc
{
    public static int sp = 7;
     public static int skr = sp * sp * sp * sp * sp;
     public static int sp5 = sp * sp * sp * sp * sp;
     public static int bk = 10;
     public static int n = 100;
     public static int an = 10;
     public static Random rnd = new Random();
     public static BigInteger two = BigInteger.valueOf(2);

     public static BigInteger skGen()
     {
         BigInteger p = BigInteger.ZERO;
         for (int i = 0; i < skr; i++)
         {
             if (i == (skr - 1))
             {
                 BigInteger head = BigInteger.ONE;
                 for (int j = 0; j < i; j++)
                 {
                     head = head.multiply(BigInteger.valueOf(2));
                 }
                 p = p.add(head);
             }
             else
             {
                 BigInteger body = new BigInteger(1, rnd);
                 for (int j = 0; j < i; j++)
```

```
                    {
                        body = body.multiply(BigInteger.valueOf(2));
                    }
                    p = p.add(body);
                }
            }
            return p;
        }

        public static BigInteger pkGen(BigInteger p)
        {
            BigInteger q = BigInteger.ZERO;
            for (int i = 0; i < sp5; i++)
            {
                if (i == (sp5 - 1))
                {
                    BigInteger head = BigInteger.ONE;
                    for (int j = 0; j < i; j++)
                    {
                        head = head.multiply(BigInteger.valueOf(2));
                    }
                    q = q.add(head);
                }
                else
                {
                    BigInteger body = new BigInteger(1, rnd);
                    for (int j = 0; j < i; j++)
                    {
                        body = body.multiply(BigInteger.valueOf(2));
                    }
                    q = q.add(body);
                }
            }
            return p.multiply(q);
        }

        public static BigInteger enc(BigInteger N, String m)
        {
            BigInteger M = new BigInteger(m);
            BigInteger r = new BigInteger(sp, rnd);
            BigInteger c = ((two.multiply(r)).add(M)).add(N);
            return c;
        }

        public static String dec(BigInteger p, BigInteger c)
        {
            BigInteger M = (c.mod(p)).mod(two);
            String m = M.toString();
            return m;
        }

        public static void main(String[] args) throws IOException
        {
            String tData[][] = new String[an][n];
            long data[][] = new long[an][n];
            String bData[][] = new String[an][n];
            for (int i = 0; i < an; i++)
            {
                Scanner in = new Scanner(new File("data/" + i + ".txt"));
                int index = 0;
                while (in.hasNextLine())
                {
                    tData[i][index] = in.nextLine();
                    data[i][index] = Long.parseLong(tData[i][index]);
                    bData[i][index] = Long.toBinaryString(data[i][index]);
                      index++;
                }
                in.close();
            }
            File file = new File("cData");
            file.mkdirs();
            PrintWriter out1 = new PrintWriter("sk.txt");
            PrintWriter out2 = new PrintWriter("pk.txt");
            PrintWriter out3 = new PrintWriter("cData/pk.txt");
            PrintWriter out4 = new PrintWriter("bk.txt");
            BigInteger p = skGen();
            out1.println(p);
            BigInteger N = pkGen(p);
            out2.println(N);
            out3.println(N);
            out4.println(bk);
            out1.close();
```

```
            out2.close();
            out3.close();
            out4.close();
            for (int k = 0; k < an; k++)
            {
                  File file1 = new File("cData/" + k);
                  file1.mkdirs();
                  BigInteger cData[][] = new BigInteger[n][];
                  for (int i = 0; i < n; i++)
                  {
                        File file2 = new File("cData/" + k + "/" + i);
                        file2.mkdirs();
                        cData[i] = new BigInteger[bData[k][i].length() + bk];
                        for (int j = 0; j < cData[i].length; j++)
                        {
                              PrintWriter out = new PrintWriter("cData/" + k + "/" + i + "/" + j +
".txt");
                              if (j < bData[k][i].length())
                              {
                                    cData[i][j] = enc(N, bData[k][i].substring(j, j+1));
                              }
                              else
                              {
                                    cData[i][j] = enc(N, "0");
                              }
                              out.println(cData[i][j]);
                              out.close();
                        }
                  }
            }
      }
}
```

Appendix B

Query Generator

```
import java.math.BigInteger;
import java.util.Random;
import java.util.Scanner;
import java.io.File;
import java.io.PrintWriter;
import java.io.FileNotFoundException;

public class QueryGenerator
{
       public static int sp = 12;
       public static long lower = (long)((Math.pow(2, 62) - 1) / 3);
       public static long upper = lower * 2;
        public static int userID = 7;
        public static int an = 10;
        public static Random rnd = new Random();
        public static BigInteger two = BigInteger.valueOf(2);

        public static BigInteger enc(BigInteger N, String m)
        {
              BigInteger M = new BigInteger(m);
              BigInteger r = new BigInteger(sp, rnd);
              BigInteger c = ((two.multiply(r)).add(M)).add(N);
              return c;
        }

       public static void main(String[] args) throws FileNotFoundException
        {
              Scanner in1 = new Scanner(new File("pk.txt"));
              Scanner in2 = new Scanner(new File("bk.txt"));
              Scanner in3 = new Scanner(new File("userID/" + userID + ".txt"));
              PrintWriter out = new PrintWriter("querySenderA.txt");
              String pq = "";
              while (in1.hasNextLine())
              {
                     pq = pq + in1.nextLine();
              }
```

```
String sbk = in2.nextLine();
int bk = Integer.parseInt(sbk);
BigInteger N = new BigInteger(pq);
String sQU = Long.toBinaryString(upper);
String sQL = Long.toBinaryString(lower);
BigInteger cQU[] = new BigInteger[sQU.length() + bk];
BigInteger cQL[] = new BigInteger[sQL.length() + bk];
File file = new File("query");
file.mkdirs();
File file1 = new File("query/upper");
file1.mkdirs();
for (int i = 0; i < cQU.length; i++)
{
    PrintWriter out1 = new PrintWriter("query/upper/" + i + ".txt");
    if (i < sQU.length())
    {
        cQU[i] = enc(N, sQU.substring(i, i + 1));
        out1.println(cQU[i]);
        out1.close();
    }
    else
    {
        cQU[i] = enc(N, "0");
        out1.println(cQU[i]);
        out1.close();
    }
}
File file2 = new File("query/lower");
file2.mkdirs();
for (int i = 0; i < cQL.length; i++)
{
    PrintWriter out2 = new PrintWriter("query/lower/" + i + ".txt");
    if (i < sQL.length())
    {
        cQL[i] = enc(N, sQL.substring(i, i + 1));
        out2.println(cQL[i]);
        out2.close();
    }
    else
    {
        cQL[i] = enc(N, "0");
```

```
                    out2.println(cQL[i]);
                    out2.close();
                }
            }
            String queryA[] = new String[an];
            int index = 0;
            while (in3.hasNextLine())
            {
                queryA[index] = in3.nextLine();
                index++;
            }
            out.println(userID);
            out.println("=");
            for (int i = 0; i < index; i++)
            {
                out.println(queryA[i]);
            }
            in1.close();
            in2.close();
            in3.close();
            out.close();
        }
}
```

Appendix C

Secure Range Query Process

```
import java.math.BigInteger;
import java.util.Random;
import java.util.Scanner;
import java.io.File;
import java.io.PrintWriter;
import java.io.IOException;

public class SRQ
{
	public static int sp = 12;
	public static int n = 100;
	public static int an = 10;
	public static Random rnd = new Random();
	public static BigInteger two = BigInteger.valueOf(2);

	public static BigInteger xor(BigInteger c1, BigInteger c2)
	{
		BigInteger c = c1.add(c2);
		return c;
	}

	public static BigInteger and(BigInteger c1, BigInteger c2)
	{
		BigInteger c = c1.multiply(c2);
		return c;
	}

	public static BigInteger not(BigInteger c0, BigInteger N)
	{
		BigInteger encOne = enc(N, "1");
		BigInteger c = c0.add(encOne);
		return c;
	}

	public static BigInteger or(BigInteger c1, BigInteger c2, BigInteger N)
	{
        BigInteger c = xor(and(c1, c2), xor(and(not(c1, N), c2), and(c1, not(c2, N))));
        return c;
    }

    public static BigInteger gt(BigInteger cNum1[], BigInteger cNum2[], BigInteger N)
    {
        BigInteger result = enc(N, "0");
        BigInteger done = enc(N, "0");
        if(cNum1.length > cNum2.length)
        {
            return result = enc(N, "1");
        }
        else if(cNum1.length < cNum2.length)
        {
            return result;
        }
        else
        {
            for(int i = 0; i < cNum1.length; i++)
            {
                BigInteger t1 = and(cNum1[i], not(cNum2[i], N));
                BigInteger t2 = and(not(cNum1[i], N), cNum2[i]);
                result = xor(and(done, result), and(not(done, N), t1));
                done = xor(done, and(not(done, N), or(t1, t2, N)));
            }
            return result;
        }
    }

    public static BigInteger enc(BigInteger N, String m)
     {
        BigInteger M = new BigInteger(m);
        BigInteger r = new BigInteger(sp, rnd);
        BigInteger c = ((two.multiply(r)).add(M)).add(N);
        return c;
     }

    public static void main(String[] args) throws IOException
    {
        Scanner in1 = new Scanner(new File("pk.txt"));
        String pq = in1.nextLine();
```

```
 BigInteger N = new BigInteger(pq);
 in1.close();
 String sUL[][] = new String[2][];
 File file1 = new File("query/upper");
File list1[] = file1.listFiles();
sUL[0] = new String[list1.length];
for (int i = 0; i < list1.length; i++)
{
      Scanner in2 = new Scanner(new File("query/upper/" + i + ".txt"));
      sUL[0][i] = in2.nextLine();
      in2.close();
}
File file2 = new File("query/lower");
File list2[] = file2.listFiles();
sUL[1] = new String[list2.length];
for (int i = 0; i < list2.length; i++)
{
      Scanner in3 = new Scanner(new File("query/lower/" + i + ".txt"));
      sUL[1][i] = in3.nextLine();
      in3.close();
}
  BigInteger cU[] = new BigInteger[list1.length];
  BigInteger cL[] = new BigInteger[list2.length];
  for (int i = 0; i < cU.length; i++)
  {
      cU[i] = new BigInteger(sUL[0][i]);
  }
  for (int i = 0; i < cL.length; i++)
  {
      cL[i] = new BigInteger(sUL[1][i]);
  }
 Scanner in4 = new Scanner(new File("querySenderA.txt"));
 String userIDs = in4.nextLine();
 int userID = Integer.parseInt(userIDs);
 in4.nextLine();
 String queryAs[] = new String[an];
 int QIndex = 0;
 while (in4.hasNextLine())
 {
      queryAs[QIndex] = in4.nextLine();
      QIndex++;
 }
 int queryA[] = new int[QIndex];
 for (int i = 0; i < queryA.length; i++)
 {
      queryA[i] = Integer.parseInt(queryAs[i]);
 }
 in4.close();
 Scanner in5 = new Scanner(new File("userID/" + userID + ".txt"));
 String aqueryAs[] = new String[an];
 int aQIndex = 0;
 while (in5.hasNextLine())
 {
      aqueryAs[aQIndex] = in5.nextLine();
      aQIndex++;
 }
 int aqueryA[] = new int[aQIndex];
 for (int i = 0; i< aqueryA.length; i++)
 {
      aqueryA[i] = Integer.parseInt(aqueryAs[i]);
 }
 in5.close();
 int fqueryAt[] = new int[an];
 int fQIndex = 0;
 for (int i = 0; i < QIndex; i++)
 {
      for (int j = 0; j < aQIndex; j++)
      {
           if (queryA[i] == aqueryA[j])
           {
                fqueryAt[fQIndex] = queryA[i];
                fQIndex++;
           }
      }
 }
 int fqueryA[] = new int[fQIndex];
 for (int i = 0; i < fqueryA.length; i++)
 {
      fqueryA[i] = fqueryAt[i];
 }
File file4 = new File("cResults");
file4.mkdirs();
```

```
        File subList3[][] = new File[fqueryA.length][];
        for (int i = 0; i < fqueryA.length; i++)
        {
            File subFile4 = new File("cResults/" + fqueryA[i]);
            subFile4.mkdirs();
            File subFile3 = new File("cData/" + fqueryA[i]);
            subList3[i] = subFile3.listFiles();
            File subsubList3[][] = new File[subList3[i].length][];
            String scData[][] = new String[subList3[i].length][];
            BigInteger cData[][] = new BigInteger[subList3[i].length][];
            BigInteger crData[][] = new BigInteger[subList3[i].length][];
            for (int j = 0; j < subList3[i].length; j++)
            {
                File subsubFile4 = new File("cResults/" + fqueryA[i] + "/" + j);
                subsubFile4.mkdirs();
                File subsubFile3 = new File("cData/" + fqueryA[i] + "/" + j);
                subsubList3[j] = subsubFile3.listFiles();
                scData[j] = new String[subsubList3[j].length];
                cData[j] = new BigInteger[subsubList3[j].length];
                crData[j] = new BigInteger[subsubList3[j].length];
                for (int k = 0; k < subsubList3[j].length; k++)
                {
                    Scanner in = new Scanner(new File("cData/" + fqueryA[i] + "/" + j
+ "/" + k + ".txt"));
                    scData[j][k] = in.nextLine();
                    cData[j][k] = new BigInteger(scData[j][k]);
                    in.close();
                }
                for (int k = 0; k < subsubList3[j].length; k++)
                {
                    crData[j][k] = and(cData[j][k], xor(gt(cData[j], cL, N), gt(cData[j],
cU, N)));
                    PrintWriter out = new PrintWriter("cResults/" + fqueryA[i] + "/" + j
+ "/" + k + ".txt");
                    out.println(crData[j][k]);
                    out.close();
                }
            }
        }
    }
}
```

Appendix D

Security Parameter Calculator for Different Bit-Lengths

```
import java.math.BigInteger;
import java.io.PrintWriter;
import java.io.FileNotFoundException;

public class blANDspRelations
{
	public static BigInteger two = BigInteger.valueOf(2);

	public static void main(String[] args) throws FileNotFoundException
	 {
		PrintWriter out = new PrintWriter("blANDspRelations.txt");
		for (int i = 1; i < 193; i++)
		{
			int sp = 2;
			 BigInteger result;
			BigInteger done;
			int temp;
			do
			{
				BigInteger r2 = two.pow(sp + 1);
				BigInteger t1 = two.pow(2 * sp + 3);
				 BigInteger t1ort2 = (two.pow(4 * sp + 7).add(two.pow(4 * sp +
6))).add(two.pow(3 * sp + 5));
				 BigInteger p = two.pow(sp * sp * sp * sp * sp - 1);
				result = two.pow(sp + 1);
				done = two.pow(sp + 1);
				for (int j = 1; j <= i; j++)
				{
					result = (done.multiply(result)).add((done.add(r2)).multiply(t1));
					done = done.add((done.add(r2)).multiply(t1ort2));
				}
				result = (result.add(result)).multiply(r2);
				temp = result.compareTo(p);
				sp++;
```

```
                } while (temp == 1);
                System.out.println(sp - 1);
                out.println(sp - 1);
            }
            out.close();
        }
}
```

Appendix E

Comparator of Sizes of Encrypted Query Results

```
import java.math.BigInteger;
import java.io.PrintWriter;
import java.io.FileNotFoundException;

public class cDataComparator
{
	public static int sp = 10;
	public static int n = 10;
	public static BigInteger two = BigInteger.valueOf(2);
	public static BigInteger three = BigInteger.valueOf(3);

	public static void main(String[] args) throws FileNotFoundException
	{
		PrintWriter out = new PrintWriter("comparisonResults.txt");
		BigInteger temp1 = BigInteger.valueOf((int)(3 * Math.pow(2, n - 2)));
		BigInteger temp2 = three.multiply(two.pow(1022));
		BigInteger PPRQ = temp2.modPow(temp1.add(temp2), 
three.multiply(two.pow(1023)));
		int temp3 = (int)(Math.pow(sp, 5) + Math.pow(sp, 4));
		BigInteger temp4 = BigInteger.valueOf(n);
		BigInteger SRQ = ((two.pow(temp3)).add(two.pow(sp + 1))).multiply(temp4);
		out.println(PPRQ.subtract(SRQ));//the deviation is about 2^temp1
		out.close();
	}
}
```