Private computation on genomic data

by

Mohammad Zahidul Hasan

A thesis submitted to
The Faculty of Graduate Studies of
The University of Manitoba
in partial fulfillment of the requirements
of the degree of

Master of Science

Department of Computer Science
The University of Manitoba
Winnipeg, Manitoba, Canada
December 2017

Thesis advisor
Noman Mohammed

Author
Mohammad Zahidul Hasan

Private computation on genomic data

Abstract

Capturing the vast amount of information encoded in the human genome is a fascinating research problem. The outcomes of this research have significant influences on a number of health-related fields, such as personalized medicine, paternity testing, and disease susceptibility testing. To facilitate these types of large-scale biomedical research projects, it oftentimes requires sharing genomic and clinical data collected by disparate organizations among themselves. In that case, it is of utmost importance to ensure that sharing, managing, and analyzing the data does not reveal the identity of the individuals who contribute their genomic samples. The task of storage and computation on the shared data can be delegated to third-party cloud infrastructures, equipped with large storage and high-performance computation resources. Outsourcing these sensitive genomic data to the third party cloud storage is associated with the challenges of the potential loss, theft, or misuse of the data as the server administrator cannot be completely trusted as well as there is no guarantee that the security of the server will not be breached. In this thesis, I propose methods for secure sharing and computation of three different functions on genomic data.

Contents

Abstract . . . ii
Table of Contents . . . v
List of Figures . . . vi
List of Tables . . . viii
Acknowledgments . . . ix
Dedication . . . x

1 Introduction **1**
1.1 Motivation . . . 2
1.2 Contributions . . . 4
1.3 Thesis Organization . . . 6

2 Background **8**
2.1 Cryptographic Background . . . 9
2.1.1 Secure Computation . . . 9
Oblivious Transfer . . . 11
Secure Multiparty Computation . . . 11
2.1.2 Adversary Types . . . 13
Semi-honest adversary . . . 13
Malicious adversary . . . 14
2.1.3 Paillier Cryptosystem . . . 14
2.1.4 Advanced Encryption Standard (AES) . . . 15
2.1.5 Bloom Filter . . . 16
2.2 Biology Background . . . 18
2.2.1 Genomic Data . . . 18
2.2.2 Genomic Data Security . . . 19
2.3 Security Requirement . . . 20

3 Related Work **22**
3.1 Secure count query . . . 22
3.1.1 Using Homomorphic Encryption . . . 23

3.1.2 Using Cryptographic Hardware . . . 25
3.2 Statistical analysis . . . 26
3.3 Secure sequence similarity search . . . 28
3.4 Other relevant works . . . 31

4 Secure Count Query on Encrypted Genomic Data 32
4.1 Count Query . . . 33
4.2 Genomic Data Representation . . . 34
4.3 System Design Overview . . . 36
4.4 Threat Model . . . 38
4.5 Basic System Design . . . 39
4.5.1 Genomic Data Without Phenotype . . . 39
Generation of *Index Tree* . . . 39
Encrypting the *Index Tree* . . . 44
Encryption of Query . . . 45
Searching on Index Tree . . . 46
4.5.2 Genomic Data with Phenotype . . . 49
Insertion into the Bloom filter. . . . 50
Encryption of the Bloom Filter. . . . 51
Encryption of the Query. . . . 52
Search with Phenotypes in the *Index Tree*. . . . 52
4.6 Experimental Results . . . 53
4.7 Security Analysis . . . 58
4.8 Summary. . . . 59

5 Secure Sequence Similarity Search on Encrypted Genomic Data 61
5.1 Similar Patient Matching . . . 62
5.2 Hamming Distance . . . 62
5.3 Genomic Data Representation . . . 63
5.4 Query Types . . . 64
5.5 System Design Overview . . . 65
5.6 Threat Model . . . 65
5.7 Basic System Design . . . 66
5.7.1 Building Compressed *Prefix Tree* . . . 67
5.7.2 Encrypting the Compressed *Prefix Tree* . . . 71
5.7.3 Searching on Encrypted *Prefix Tree* . . . 71
5.8 Experimental Results . . . 74
5.9 Security Analysis . . . 78
5.10 Summary . . . 79

6 Identification of Similar Patients with Edit Distance Approximation 81
6.1 Edit Distance . . . 81
6.2 System Design Overview . . . 82
6.3 Threat Model . . . 83
6.4 Genomic Data Representation . . . 83
6.5 Basic System Design . . . 84
6.5.1 Edit Distance Approximation . . . 85
6.5.2 Bloom Filter Representation . . . 87
6.5.3 Construction of the BF-Tree . . . 89
6.5.4 Encryption of the BF-Tree . . . 91
6.5.5 Construction of the Query . . . 91
6.5.6 Search on Encrypted BF-Tree . . . 92
6.5.7 Runtime Complexity. . . . 97
6.6 Experimental Result . . . 97
6.7 Security Analysis . . . 99
6.8 Summary . . . 100

7 Conclusion 101
7.1 Summary . . . 101
7.2 Looking ahead . . . 102

Bibliography 113

List of Figures

2.1 Garbled circuit for an AND gate. 10
2.2 Example of a Bloom filter for two strings x and y with three hash functions 17

4.1 Architecture of our proposed solution. 36
4.2 Different states during the generation of *index tree*. Figure 4.2a, 4.2b, and 4.2c represents the tree after the insertion of the first, second, and third record respectively. 40
4.3 *Index tree* for Table 4.2. 40
4.4 Sequence diagram of our proposed model. 48
4.5 Information stored in a single node. 49
4.6 Tree building time and tree encryption time for count queries on datasets with different number of SNPs. 55

5.1 Different states during the generation of the *prefix tree*. Figure 5.1a, 5.1b, and 5.1c represents the tree after the insertion of the first, second, and third record respectively. 66
5.2 *Prefix tree* and compressed *prefix tree* generated from the data represented in Table 5.1. 69
5.3 Sequence diagram of our proposed model. 75
5.4 Data read and *prefix tree* building time. 76
5.5 Figure 5.5a shows the query execution time on different datasets with different number of records and a fixed hamming distance $k = 10$. Figure 5.5b shows the query execution time on a dataset of 10000 records and different hamming distances $k \in 1, 2, 3, 8, 10$. 77
5.6 Figure 5.6a shows the communication overhead on different datasets with different number of records and a fixed hamming distance $k = 10$. Figure 5.5b shows the communication overhead on a dataset of 10000 records and different hamming distances $k \in 1, 2, 3, 8, 10$. 78

6.1 Calculation of edit distance using reference genome 87

6.2 A sample BF-tree containing only 8 Bloom filters at child node. Each node except the root node has two children and is the union of its left and right child. Here, Bloom filter $\mathcal{B}_{12}$ is the union of Bloom filters $\mathcal{B}_1$ and $\mathcal{B}_2$. Similarly Bllom filter $\mathcal{B}_{1...4}$ denotes that it is the union of $\mathcal{B}_{12}$ and $\mathcal{B}_{34}$. 89
6.3 An example of traversing the tree. In each node, the Hamming distance between the query Bloom filter q and the Bloom filters at the child nodes are compared. The child node containing the more similar Bloom filter to q is traversed next. 93
6.4 The overall protocol of our proposed solution. Both the *CS* and *CI* have two common inputs, i) the public reference genome, **Ref** and ii) the hash function of Bloom filter, $\mathcal{H}$. The *CI* generates the tree T offline. The *CS* also generates the query Bloom filter, $\mathcal{B}_q$ offline. Then the server and the client engage in a secure computation protocol to find the appropriate leaf node containing the similar patient in the tree, T. 96
6.5 Average running time to execute a query. Note that, the reported time does not include the amount of time required to generate end encrypt the tree in the preprocessing stage. 98

List of Tables

2.1 Garbled truth table for an AND gate 10

4.1 Different properties of existing techniques for count query 33
4.2 Data representation in the *Certified Institution (CI)* 35
4.3 Configuration of the *CS*. 54
4.4 Query execution time. Times are measured in seconds. 56
4.5 Comparison of count query execution time on a dataset of 5000 records, where each record contains 300 SNPs, for different query sizes. 57
4.6 Communication overhead in MB. 58
4.7 Size of original database, unencrypted tree and encrypted tree in MB. 58

5.1 Sample Genomic data representation 63

6.1 Configuration of the server. 98
6.2 The cost of garbled circuit per query. 99

Acknowledgments

First of all, I would like to express my gratitude to the Almighty for granting me the opportunity to complete this thesis.

My sincerest gratitude to my supervisor, Dr. Noman Mohammed for giving me the opportunity to work under his supervision. I am grateful to him for his tremendous support, care and encouragement towards my thesis. I have learned a lot from him and without his guidance and suggestions, I would not have been able to finish this thesis. I also extend my thanks to Drs. Carson Kai-Sang Leung and Ken Ferens for being my thesis committee members. I appreciate the time they have spent for reading my thesis and their thoughtful suggestions.

I am indebted to my collaborators Md Safiur Rahman and Md Nazmus Sadat for their valuable insights and feedbacks. I also want to thank my colleagues at the Data Security and Privacy Lab for their support, in the form of friendship and encouragement: Md Momin Al Aziz, Kazi Wasif Ahmed, Md Waliullah, Reza Ghesemi, and Toufique Morshed. I also would like to thank the faculty and staff members of the Department of Computer Science who have helped me in various ways.

Finally, I would like to thank my wonderful family. I will forever be indebted to my sisters and especially to my parents for everything I have achieved so far in my life.

This thesis is dedicated to my parents and sisters for their unconditional love, endless supports and encouragements.

Chapter 1

Introduction

The rapid advancement of genome sequencing technologies produces newly sequenced genomes at a pace that the fully sequenced genomes have become widespread and affordable. It has also opened up the opportunity to develop new applications in the health-related fields. Paternity testing, personalized medicine, genetic compatibility testing, and disease susceptibility testing are some of the few applications. All of the applications involve the analysis of human genome which can reveal essential information about an individual. For example, disease susceptibility testing can determine an individual's predisposition to a specific disease such as breast cancer, diabetes, and Alzheimer's [1]. This kind of analysis is usually done by checking if the genome of an individual match with a list of already known variations and then calculating and predicting the disease susceptibility [1]. Most of these analyses rely on genome-wide association study (GWAS). GWAS helps to understand and identify the associations between the genetic variations and traits like major human diseases [2]. Also, in the personalized medicine, a physician can prescribe a safe and effective

medical treatment based on the patient's genetic profile to minimize the side effects. Needless to say, that to make all these applications effective, the analysis of the data has to be accurate.

To guarantee significant accuracy in this type of analysis, a large number of genomic sequences are required, the collection of which are sometimes beyond the capability of a sole organization [3]. Allowing the access of the genomic data surpassing the premise of the organization responsible for initial collection is a viable solution. But, delegating the access of the data, be it owned by a government organization or a private research institution, is not always very straightforward because of the nature of the genomic data.

1.1 Motivation

Genomic data cannot be treated as any other data; it has some distinctive features. Naveed *et al.* [4] identified six special features of genomic data. A person's DNA changes very little over time and it is unique – two individuals can easily be distinguished from their data. Furthermore, information about the genotype, phenotype, and blood relatives of an individual can also be inferred from his or her genomic profile. Due to this sensitive nature, disclosure of this data has significant privacy risks. For instance, a person carrying the mutation of a specific gene which increases the likelihood of developing a specific disease might be denied by an insurance company for his health coverage. Hence, while sharing genomic data among multiple institutions, safety measures should be taken to uphold the privacy of the individuals who contribute the data. For this purpose, different privacy policies have been

developed, thus facilitating the task of analysis to be done in a broader range.

The volume of the aggregate shared data is enormous and requires a vast amount of storage space. Due to the quality of services offered by the cloud infrastructures at a considerably lower rate, especially having the characteristics of high availability and scalability, cloud computing services can be adopted for this purpose. However, cloud services are vulnerable to the security threats and an adversary capable of breaching the security of the cloud server would be able to access the residing data. One published news clearly demonstrated that privacy should not be expected to be preserved from cloud service providers [5].

In this thesis, our aim is to design a secure framework for outsourcing genomic data and executing three specific operations on it. These operations are count query, Hamming distance and edit distance approximation. Count query determines the number of records in the database that match the query predicate and it is very useful for genetic association studies to compute several statistical algorithms (see Chapter 4). Hamming distance and edit distance are the metrics used to identify similar patients in a pool of patients which is known as similar patient matching and is one of the prerequisite steps in personalized medicine.

While designing our frameworks for solving these three different problems, we emphasized on providing the security of three parameters - i) the shared **data** on which the analysis will be done, ii) the **query** to be executed on the shared data, and iii) the **output** of the executed query. Providing the security of the data, query and output are important in any data sharing and computation framework. The detailed discussion on these three parameters are provided in Section 2.3.

Anonymization methods have been proved to be ineffective for protecting the genomic data [6; 7; 8] as these techniques incur high utility loss. Recent advancement of the cryptographic techniques makes it possible to compute a predefined function on encrypted dataset from multiple parties and return the function's result without revealing any information about the data from different parties [9]. For this reason, several privacy preserving techniques have been developed using cryptography to achieve the goal of sharing and computation on encrypted genomic data. In this thesis, to provide the security of our proposed models, we opt for some effective and computationally efficient state of the art cryptographic schemes.

1.2 Contributions

Along with providing the security of the shared data, we also emphasize improving the efficiency of our search algorithm. We have used a tree-based indexing to pre-filter the search result in all the three models we propose which improves the performance significantly. Our indexing techniques also provide an effective storage solution for large genomic datasets. In addition, modification of the trees is very easy. New records can easily be added, deleted or modified to or from the nodes of the tree.

Our proposed models also provide the security guarantee of all the three security requirements: *data privacy, query privacy, and output privacy* (the query result is only disclosed to the query initiator). We execute the queries in our models by traversing the nodes of the tree. We have used secure function evaluation (SFE) to take the decision which node to traverse. For SFE, we have used Yao's garbled circuits [10]. Our proposed methods do not require the active participation of a trusted entity (e.g.

a proxy server) for secure evaluation of the query or decryption of the result of the query.

Through experiments and evaluation, we demonstrate the effectiveness and superiority of our approaches in comparison with the previous approaches. The Three models we propose in this thesis can perform the following computations:

Secure Count Query. We present a secure method for executing count query operation on the encrypted data. Genomic data is outsourced after encryption to a third party cloud server. Execution of a query is done by traversing an encrypted tree, called *index tree* where the decision of traversing each node is made by checking whether a query predicate matches with a particular branch of the tree. Depending on the query, branches of the tree are traversed to calculate the result from the data stored in the nodes which match the query predicate. Our proposed method can handle datasets containing both genotype and phenotype. We have used a data structure called Bloom filter to process the phenotype attributes of the datasets in a privacy-preserving way.

Secure Hamming Distance. We present a secure method for determining the similar sequences in a database of genomic sequences. We have used a prefix tree based indexing algorithm to pre-filter the search result. We used Hamming distance as the metric of the similarity measure. During the execution of a query, each node is traversed by checking whether a query sequence (or a subset of a query sequence) matches with a particular branch of the tree within a certain threshold k.

Secure Edit Distance Approximation. We provide an alternative secure method to find similar patients using edit distance as the similarity measure. We

represent each sequence as a Bloom filter and generate a *BF-tree* using all the Bloom filters to pre-filter the search results. Each leaf node contains a Bloom filter generated from a genomic sequence and each Bloom filter in the parent node is the union of the Bloom filters of its left and right child. Traversal of this tree resembles a binary search. During the search on this tree, the Bloom filters of the left and right child of a particular node are compared with the query Bloom filter and the child node containing the more similar Bloom filter is traversed.

1.3 Thesis Organization

The rest of this thesis is organized as follows.

- **Chapter 2** provides the necessary cryptographic and biological background to understand the techniques we have adopted in our frameworks to provide the security of the genomic data.
- **Chapter 3** presents a brief discussion on the related literature. Part of this chapter appears in [11].
- **Chapter 4** addressed the secure count query operation on the encrypted genomic data. The results of this chapter appear in [12] and [13].
- **Chapter 5** presents a model to find similar patients in a database using Hamming distance. The results of this chapter appear in [14].
- **Chapter 6** describes an algorithm to find similar patients using a Bloom filter search tree and calculating edit distance. The result of this chapter has been

submitted in [15].

- **Chapter 7** concludes the thesis providing some future research directions.

Chapter 2

Background

Nowadays a large number of private information about individuals is being collected and stored by both government and corporate organizations. This information includes electronic medical records (EMR), location data, browsing histories, social networks and much more. The sensitivity of these data may vary. Sometimes seemingly an innocuous dataset can be combined with other publicly available datasets which may result in disclosure of private information. That is why securing the dataset containing the personal information of an individual is very important. In the case of health records such as diagnosed diseases or usage of medicines, this sensitivity is even more important.

Encryption of the data is a viable solution to this problem. It protects the data from the adversaries who intend to infer private information. But the main reason behind collecting this information is to use it in different applications and provide a more personalized solution. Searching is arguably the most important database operation. Encryption of the database makes it more complicated to search for a

particular record. The more strong the encryption scheme, the more difficult it is to search in the encrypted database. To bridge the contradictory interests of data security and efficient searching, a number of cryptographic techniques are developed. Here we discuss some cryptographic techniques and biology background relevant to this thesis.

2.1 Cryptographic Background

We have utilized some state of the art cryptographic techniques to provide the security of our proposed models. So, we first provide some relevant background information necessary to understand those techniques.

2.1.1 Secure Computation

In the 1980s, Andrew Yao introduced the concept of secure computation. He proposed a cryptographic protocol to solve a problem where two parties Alice and Bob without disclosing their actual wealth wish to determine who is richer. This problem is known as the *Millionaires' Problem* and the protocol he proposed is known as *Yao's protocol* or *Yao's garbled circuit protocol* [10]. Yao's protocol can essentially compute almost any mathematical function.

Let, two parties Alice (A) and Bob (B) wish to compute a function, $f(x, y)$ where x and y denote their respective inputs. The protocol evaluates the function f through a Boolean circuit which is made of 2-input XOR and AND gates. The total number and kind of gates necessary to calculate $f(x, y)$ depends upon the function f. The amount of work done by each party grows proportionally to the number of gates in

A	B	Output	Encrypted output	Garbled value
A_0	B_0	K_0	$E_{A_0}(E_{B_0}(K_0))$	$E_{A_1}(E_{B_1}(K_1))$
A_0	B_1	K_0	$E_{A_0}(E_{B_1}(K_0))$	$E_{A_1}(E_{B_0}(K_0))$
A_1	B_0	K_0	$E_{A_1}(E_{B_0}(K_0))$	$E_{A_0}(E_{B_1}(K_0))$
A_1	B_1	K_1	$E_{A_1}(E_{B_1}(K_1))$	$E_{A_0}(E_{B_0}(K_0))$

Table 2.1: Garbled truth table for an AND gate

the circuit evaluating function f. After running the protocol, both A and B learns the output of $f(x, y)$ but neither of them learn about the input or any other information of the other party.

Construction and evaluation of Garbled Circuits involve two disparate entities known as garbler (A) and evaluator (B). In each wire of the Boolean circuit, the garbler generates six different keys (W_{A_0}, W_{A_1}, W_{B_0}, W_{B_1}, W_{K_0}, W_{K_1}) where W_{A_0}, W_{A_1} are the input bits for A; W_{B_0}, W_{B_1} are the input bits for B and W_{K_0}, W_{K_1} are the output bits. All input or output bits are associated with either wire 0 or wire 1. Then the garbler constructs a garbled version of the $f(x, y)$ by shuffling the rows of the computation truth table and sends the table to B (known as evaluator) along with the input of A. Suppose the input of A is $I(g)$. After receiving the circuit, the evaluator evaluates the circuit. He runs a 1-out-of-2 oblivious transfer protocol [16] to obliviously get the garbled-circuit input values to obtain its private input, $I(e)$. Therefore, from $I(g)$ and $I(e)$, the evaluator can calculate $f(x, y)$.

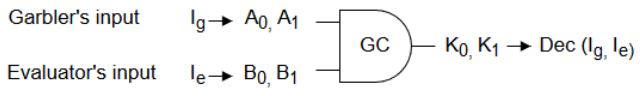


Figure 2.1: Garbled circuit for an AND gate.

For example, figure 2.1 represents a garbled circuit for an AND gate and Table 2.1 represents the garbled values for that AND gate. If A sends garbled value (column 4 of Table II), with his input A_1 and if B gets his input, B_0 from the 1-out-of-2 oblivious transfer protocol [16], then B can evaluate the output by decrypting only $E_{A_1}(E_{B_0}(K_0))$. Evaluator can also decrypt the other rows but for those rows he will only get the garbage values. The impressive property is that B receives the correct output but does not have any idea about the computation he carried out or what gate he has computed.

Oblivious Transfer

An oblivious transfer (OT) is a two-party (sender, receiver) cryptographic protocol which was introduced by Michael O. Rabin [16] in 1981 where the sender sends a message to the receiver but remains oblivious whether the receiver receives the message or not. Later, Even *et al.* [17] developed an improved and useful version of it. In their protocol, the sender holds the values (W_0, W_1) and the receiver holds the indexes $r \in \{0, 1\}$. The protocol is executed in a way that after the protocol execution, the receiver only learns about the W_r, but not the other values held by the sender. The sender also does not know anything about the indexes held by the receiver.

Secure Multiparty Computation

Yao's protocol [10] supports secure two-party computation. The subsequent research tries to support computation involving multiple parties.

In the distributed computing environment, a number of independent, yet con-

nected devices wants to jointly compute a function. In a server with database environment, this function might be a simple database operation like update or search. The aim of secure multiparty computation is to ensure that all the tasks in the distributed computing can be done in a secured manner. Distributed computing handles the issues like system failures, concurrency etc. during the computation, but it does not address the problem where the computation itself is under attack by one of the participating entities or an external entity. The adversary attacking the protocol may want to learn private information or manipulate the computation in a way that it yields an incorrect result.

So, the two most important requirements for a secure multiparty computation protocol are *correctness* and *privacy* [18]. The correctness requirement states that at the end of the computation, each party should receive the correct output. Producing correct output means that no internal or external adversary could manipulate the computation. So, the involved parties could successfully compute the function that they have set out. The privacy requirement states that from the computation, the engaged parties should only learn the output of the function. The input from each of the parties is not revealed to other parties.

We can formally define secure multiparty computation as follows. Let, the parties participated in the computation are $P_1, P_2, \ldots, P_n$. These n parties agreed to compute a function f which takes n inputs. To compute f, each party P_i gives his own input x_i. The computation is said to be secured if the computation of function $f = (x_1, x_2, \ldots, x_n)$ satisfies the *correctness* and *privacy* requirements.

As an example, we can consider the task of voting. Consider n parties participate

in the task of electronic voting with only yes/no decision. The input of party i is x_i which is 1 if he votes yes, otherwise $x_i = 0$. The function $f(x_1, x_2, \ldots, x_n) = \sum_{i=1}^{n} x_i$ reveals the number of votes. The correctness requirement ensures that the no party can manipulate the the result of the voting. The privacy requirement ensures that the individual votes of each of the parties are not revealed to other parties.

2.1.2 Adversary Types

Security in multiparty computation is discussed assuming that a subset of parties involved in the computation are controlled by an adversarial entity. The aim of this adversarial entity is to attack the protocol execution [18]. The subset of parties who follows the instructions and thus controlled by the adversarial entity are called *corrupted*. In the classic adversarial models there are the following two main types of adversaries:

Semi-honest adversary

The parties in the semi-honest adversarial model correctly follow the protocol specification and do not have the intention to behave maliciously to produce the incorrect result. However, they may attempt to learn information that should remain private during the protocol execution as they obtain the internal states of all other parties. Although this is a comparatively weak adversarial model, it is useful in some cases where only the leakage of the output to the parties is allowed. This type of adversary is also sometimes called “honest-but-curious” or “passive” adversary.

Malicious adversary

The corrupted parties in the malicious adversarial model can deviate from the specification of the protocol at will during the protocol execution to cheat. They can adopt different strategies to carry out their attack or learn the information they are not allowed to. Naturally, the protocols that guarantee security against malicious adversaries are more secure as it defends all other adversarial attacks. Although, this is the ideal security model, it generally makes the protocol less efficient.

2.1.3 Paillier Cryptosystem

Paillier cryptosystem [19] is a member of homomorphic cryptosystem family. Homomorphic encryption allows to perform computation on the encrypted data without decrypting it, and if we decrypt the result, it would be the same if we perform the computation on the plaintext. *Paillier cryptosystem* [19] supports addition and it is semantically secure: an adversary with the finite computational power and with the possession of the ciphertext would not be able to extract any information about the plaintext. To guarantee this security, this cryptosystem produces different ciphertexts when a same message is encrypted multiple times. This randomness implies that this cryptosystem is a probabilistic encryption scheme.

We use *Paillier Cryptosystem* [19] to encrypt the data and utilize its homomorphic properties to execute count query. In the *Paillier Cryptosystem* [19], a key generation algorithm produces a pair of keys: a secret key, sk and a public key, pk. The public key and the secret key are used for encryption and decryption purposes respectively. So after the encryption of a message m if we get two ciphertexts $c_1 = \xi_{pk}(m)$ and

$c_2 = \xi_{pk}(m)$, then $c_1 \neq c_2$ and $\xi_{sk}(c_1) = \xi_{sk}(c_2) = m$. Here, $\xi_{pk}(m)$ denotes encryption of message m using the public key pk and $\xi_{sk}(c_1)$ and $\xi_{sk}(c_2)$ denotes decryption of the ciphertexts c_1 and c_2 respectively using the secret key sk.

Homomorphic Properties: Assume that we encrypt two messages m_1 and m_2 using the same public key pk which produces ciphertexts c_1 and c_2 respectively, and k is a constant number. Then, *Paillier cryptosystem* [19] guarantees the following homomorphic properties which can be utilized to execute count query:

- If we multiply two ciphertexts, after decryption we will get the sum of their corresponding plaintexts.

$$\xi_{sk}(\xi_{pk}(m_1) \cdot \xi_{pk}(m_1) \ mod \ n^2) \ = \ m_1 + m_2 \ mod \ n$$

- If we raise a ciphertext to the power of a constant k, after decryption we will get the product of the corresponding plaintext and the constant.

$$\xi_{sk}(\xi_{pk}(m_1)^k \ mod \ n^2) \ = \ km_1 \ mod \ n$$

2.1.4 Advanced Encryption Standard (AES)

Advanced Encryption Standard (AES) is an encryption technique which uses a fixed length group of bits called block for encryption or decryption. We use a variation of AES known as *Counter Mode* (AES CTR) [20] to encrypt the data. Counter mode turns a block cipher into a stream cipher. It produces the next keystream block by encrypting consecutive values of a *counter*. The *counter* is a n bit string which is

non-repeating. The same combination of *initialization vector* (IV) and key must not be used more than once to ensure security. The encryption function is defined as:

$$y_i = \xi_k(IV \mid\mid CTR_i) \oplus x_i, i \geq 1$$

and the decryption function is defined as

$$x_i = \xi_k(IV \mid\mid CTR_i) \oplus y_i,\ i \geq 1$$

where x_i is the plaintext, y_i is the ciphertext, and IV is the *initialization vector*.

2.1.5 Bloom Filter

Bloom filter is a data structure designed to check whether an element is present in a set with a small probability of reporting false positive. It was first proposed by Burton H. Bloom [21]. It is implemented as a bit array of fixed length m. Let, the Bloom filter, $\mathcal{B}$ represents a set $X = \{x_1, x_2, ..., x_n\}$ of n elements. There are k independent hash functions $\mathcal{H} = \{h_1, h_2, ..., h_k\}$ which are associated with the indexes of the bloom filter. The range of each hash function is between 1 and m. Initially, all the indexes $\{0, ..., m\}$ of $\mathcal{B}$ are set to 0. To add an element $x \in X$ in $\mathcal{B}$, a hash result is computed as follows:

$$\mathcal{H}(x) = (h_1(x), h_2(x), ..., h_k(x))$$

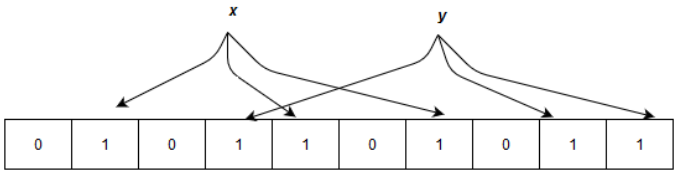


Figure 2.2: Example of a Bloom filter for two strings x and y with three hash functions

Then these hash values are used as an offset into the bit array, $\mathcal{B}$ and the corresponding bits are set to 1. To check if a query, q is in $\mathcal{B}$, similarly k hashes are computed over that Bloom Filter:

$$\mathcal{H}(q) = (h_1(q), h_2(q), ..., h_k(q))$$

Then, the bit positions from $\mathcal{H}(q)$ are checked in $\mathcal{B}$. If any corresponding bit is not 1, then that element is not present in $\mathcal{B}$. In this way, Bloom filter determines whether an element is definitely not in the set. Otherwise, we assume that the element is present in the Bloom filter. Figure 2.2 shows an example of Bloom filter.

The probability of false positive is:

$$p = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \tag{2.1}$$

$$\approx \left(1 - e^{-kn/m}\right)^k \tag{2.2}$$

Where $(1 - \frac{1}{m})^{kn}$ indicates the probability that a single bit is still 0 in the Bloom filter of length m after adding n elements using k hash functions. Equation 2.1 can be transposed to calculate the length of the Bloom filter, $\mathcal{B}$ as:

$$m = \frac{-1}{(1 - p^{1/k})^{1/kn} - 1} \tag{2.3}$$

Where $m, p,$ and k are respectively the length, the false positive probability, and the number of hash function of the Bloom filter.

The main advantage of Bloom filter is that it is very fast. A Bloom filter with length m and k hash functions, the time required for both insertion and membership testing is $\mathcal{O}(k)$. The more hash functions we have, it ensures better security but the array quickly fills up and the Bloom Filter becomes slower. Less hash functions make it faster but ultimately results in possibility of getting more false positive results.

2.2 Biology Background

This section provides some relevant biological background information and discusses why security is important while handling biological data.

2.2.1 Genomic Data

The genome contains the hereditary information of an organism. The human genome is encoded in *deoxyribonucleic acid* molecules which we commonly know as DNA. DNA molecules consist of two biopolymer chains each of which in turn consists of nucleotides. These nucleotides are represented as *A, C, G, T* which are the acronyms of *Adenine*, *Cytosine*, *Guanine* and *Thymine* respectively. In DNA, these nucleotides form base pairs by making bonds with each other: *A* bonds with *T* and *C* bonds with *G*. There are 3 billion base pairs in whole haploid human genome se-

quence, distributed across 23 chromosomes. The DNA of two different individuals are almost identical ($\sim$ 99%). In fact, Venter *et al.* [22] showed that DNA of two individuals differ no more than by 0.5%. This small amount of variations distinguish one individual from another. Several types of genetic variations occur in human population, such as single-nucleotide polymorphism (SNP), copy-number variations (CNVs), rearrangement etc. *Single Nucleotide Polymorphism* (SNP) is the most common form of DNA variation at a specific position in the genome, which represents a difference in a single nucleotide. Most of the SNPs do not have any effect on human health. But some SNPs are directly responsible for developing a particular disease in the human body.

2.2.2 Genomic Data Security

As genomic data is widely used in a number of applications, its security issues are well understood from their use cases. It is well established that change in the genomic sequence of an individual due to mutation can affect their health. Some mutations can affect immediately whereas some affect at some point in future. Also, some of these mutations are acute, whereas some are not. So, individuals might want to learn their genomic status for the sake of future planning or research. Sometimes these variations affect an individual's response to a specific treatment requiring the doctors to prescribe the different amount of drugs.

While existing known mutations are helpful in the aforementioned cases, new unknown mutations are being discovered regularly, thanks to the rapid advances in the sequencing technologies. The amount of genomic data being collected, stored

and analyzed today is unprecedented, rapidly accelerating the rate of new mutation discovery.

Nowadays a number of companies allow individuals to analyze their genomic data to determine disease susceptibility risks and perform genomic compatibility testing with their potential partners. DNA found at the crime scene is also analyzed to track the potential criminal. To do so it is legally allowed in some countries to collect and store the DNA of a suspect. The security issue is generated largely from these collection and storage of genomic data for these purposes.

As mentioned earlier, Genome-wide association study (GWAS) helps to identify the association between SNPs and human diseases. GWAS examines the SNPs from the DNA collected from thousands of individuals and tries to pinpoint SNPs that may be responsible for a particular disease [2]. Ensuring the security of the SNPs in the GWAS is very important which has been clearly demonstrated by the work of Lin *et al.* [23] who showed that only 75 SNPs are enough to uniquely identify an individual. Besides, sensitive personal information can also be inferred from the aggregate statistics in GWAS [24; 25; 26].

2.3 Security Requirement

Ensuring the security guarantee of the following three parameters are of paramount importance while designing a secure genomic data outsourcing and computation mechanism:

1. ***Data privacy.*** The data stored in the cloud server, as well as the computation, should be secured and should not leak any information about the data. Even

if the cloud server gets compromised, the confidentiality of the data should be ensured.

2. ***Query privacy.*** The institutions that contribute the data, the cloud service provider or an adversary in the possession of a compromised server should not learn anything about a query executed by a researcher or an institution.

3. ***Output privacy.*** The result of the query should not be disclosed to any party except the researcher who initiated the query. Here, by the term output privacy, we do not mean to prevent any inference attack that is possible using the results of the query (see Section 7.2 for further discussion).

In this thesis, we focus only on these three security requirements while designing our models. It is beyond the scope of this thesis to address other security requirements such as authentication, audit, and data integrity. But, we acknowledge that these issues are also essential to build a secure system and have been addressed extensively in the literature.

Chapter 3

Related Work

In this chapter, we provide an overview of the existing secure solutions for outsourcing genomic data to the cloud and executing the three secure operations we addressed in this thesis.

3.1 Secure count query

In recent years, several methods have been proposed addressing the problem of executing count query operation on the encrypted genomic data. These solutions differ in the way of sharing the data as well as the methodologies utilized for the computation. In this section, we present an overview of the two proposed methods for secure count query operation on outsourced genomic data. To the best of our knowledge, these are the only two research that has particularly addressed this problem. We also present some other relevant works that are closely related to this problem.

3.1.1 Using Homomorphic Encryption

In 2008, Kantarcioglu *et al.* [27] first addressed the problem of counting the number of records based on the genomic and clinical features on the encrypted genomic data. Their proposed framework incorporates four different types of participants:

i. **Data Holders.** The hospitals and research institutions who wants to share their data.

ii. **Data Users.** The individuals or organizations who might be biomedical researchers and interested in executing queries on the data.

iii. **Data Storage site (DS).** It is essentially a third party cloud server used as a repository of the shared data.

iv. **Key Holder Site (KHS).** It is a trusted third party responsible for the management of the keys and decryption of the result of the query.

So, their protocol involves two different third parties to provide the security of the framework.

Security assumptions: The authors assumed the participants to be non-colluding and semi-honest. The rationale behind using two third parties is to guarantee the confidentiality of the data as well as to ensure that there is no single point of failure. The distribution of the data and the key to only one party would enable an adversary to get the access of the unencrypted data in the event that the server is compromised.

Overview of the model: The model they adopted to ensure the secure sharing and computation of genomic data worked as follows: First, the KHS who possesses

both the public and private keys, gives the public key to the DS. The data holders, willing to participate in the sharing process get the public key from the DS. The data holders then use this public key to encrypt their data and send the encrypted data to the DS. DS works as the repository of the data who has sufficient storage and bandwidth capacity to manage large databases. The data users who are interested in analyzing the data send their query to the DS. The DS executes this query on the encrypted data and then sends the encrypted result to the KHS who uses the private key to decrypt the result. Finally, the KHS sends the decrypted result to the data users.

The authors evaluated their model using a database of SNP sequences. They represented each nucleotide as a pair of bits, and each sequence as a series of binary values. When the DS receives a query from a data user, it matches the query predicates with the encrypted database and produces an intermediate encrypted result. The DS then sends this encrypted intermediate result to the KHS to calculate the final result.

Limitations: Though this model provided the first solution for the addressed problem, it is not free from flaws. This approach has several drawbacks. *First*, colluding third parties might result in the exposure of sensitive information. *Second*, huge bandwidth is required for the communication between the DS and KHS. Also, the KHS needs to be online during the query execution for decrypting the final result. *Third,* the searching process during the query execution is linear to the number of records and the homomorphic encryption scheme used is very expensive. This resulted in a system that is not practical for a large database (see Section 4.6 for experimental

results). *Fourth*, all data owners use the same key for encrypting their data. In the event that the key is stolen, it may lead to the disclosure of the whole dataset. *Finally*, this method reveals the data access pattern to the cloud server.

3.1.2 Using Cryptographic Hardware

In the subsequent work, Canim *et al.* [28] opted to use a tamper-resistant cryptographic hardware to facilitate secure storage and processing of clinical data at a single third-party by abandoning the trusted entity, KHS incorporated in [27]. Thus, this model overcomes the limitations of [27]. In reality, the task of the trusted entity was realized by a trusted hardware.

Security assumptions: The authors assume that the tamper-resistant hardware is co-located with the DS. This inclusion of cryptographic hardware enables their model to withstand untrusted adversary. They used IBM 4764 PCI-X secure coprocessors (SCPs) as the cryptographic hardware and it offers some advantages in terms of security. It completely hides the computation from the server and as soon as it detects any tampering, it clears the internal memory. In addition, the secure coprocessor fetches only the required attributes from all the records in the database each time a query is executed. As all the records are accessed for executing each of the queries, this model ensures that it does not reveal the access pattern to the untrusted DS.

Overview of the model: The workflow of this model is as follows. Each data holder generates it's own symmetric encryption key using AES in counter (CTR) mode and uses it to encrypt their genomic and clinical records. The SCP provides a

public key to each data holder through a secure Ethernet channel. The data holder use this public key to encrypt their symmetric key and then transfer it to the SCP using the same Ethernet channel. They also send their encrypted records to the DS. SCP uses the sovereign join algorithm introduced in [29] to eliminate duplicate records and stores the encrypted records in the DS. The data users send their queries to the DS. The DS fetches the encrypted attributes required to execute the query based on the query predicates and forwards these attributes along with the data user's query to the SCP. SCP then decrypts these attributes and executes the data user's query. Finally, SCP sends the result to the data user using a secure socket layer (SSL) channel.

Limitations: The most notable limitation of this model is the cryptographic hardware itself. This model assumes the existence of a tamper-resistant hardware with the DS. This assumption may not be feasible as to guarantee the presence of cryptographic hardware by a cloud service provider might not be always possible. In addition, cryptographic hardware has very small memory capacities and computational power [4] which impedes the processing of larger queries.

3.2 Statistical analysis

To protect the privacy of the genome database, Lauter *et al.* [30] proposed a method where a contingency table is generated first from the genomic data and then this table is encrypted using a leveled homomorphic encryption to store on a single cloud server. Their method enables the cloud server to compute several statistical algorithms: *Pearson Goodness-of-Fit or Chi-Squared Test, Linkage Disequilibrium, Estimation Maximization (EM) and Cochran-Armitage Test for Trend (CATT)*. These

are commonly used algorithms in genetic association studies. However, their proposed method is not particularly designed to execute count query based on arbitrary predicates.

Kamm et al. [31] employed secret sharing technique to guarantee the security of shared data and used secure multi-party computation to compute the value of different tests like χ^2 *test*, *Cochran-Armitage Test for Trend* and *Transmission Disequilibrium Test*. However, secret sharing based techniques require at least three non-colluding parties and demand significant multi-way communication among these parties. Hence, these techniques may not be practical as the cloud-based applications are designed following the architecture of client-server model.

Feng et al. [32] recently proposed a distributed framework, PRINCESS, using the Intel Software Guard Extensions (SGX). They proposed a secure *Transmission Disequilibrium Test* algorithm for rare disease analysis (in particular Kawasaki Disease (KD) [33]). Their framework facilitates cross-institutional collaborations and enables multiple parties to compute functions over the distributed data securely (i.e., each institution holds data locally in clear text).

Huang *et al.* [34] proposed a method based on distribution-transforming encoder (DTE) scheme to protect genomic data from any brute-force attack. They only considered to solve two algorithms, *Pearson Goodness-of-Fit* and *Linkage Disequilibrium* using this model. Zhang *et al.* [35] used homomorphic encryption to compute *Chi-Squared Test* in untrusted public cloud.

Choi *et al.* [36] proposed a framework to execute queries on genomic data using a leveled homomorphic encryption technique. The proposed system is built on the i2b2

framework and is able to compute a number of functions such as reference/alternate allele frequencies, and frequency of genotypes.

Xie *et al.* [37] proposed a scheme for securely performing *meta-analysis* for genetic association study. Instead of storing data to multiple cloud storage (like Kamm *et al.* [31] and Zhang *et al.* [38]), they kept the data in the corresponding data owner's premises. Wang *et al.* [39] designed a *somewhat homomorphic encryption* based technique to compute *exact logistic regression* to discover rare disease variants to analyze disease susceptibility in an untrusted cloud environment.

3.3 Secure sequence similarity search

In sequence analysis, new features or structures of DNA, RNA or peptide sequences are discovered or understood by sequence alignment. With the rapid advancement of sequencing technologies, the number of sequenced genomes is also growing. So it can easily be anticipated that the task of finding similar patients from a large number of genomic sequence will increase day by day. Up to date, many researchers worked on sequence similarity search using different approaches. Dugan *et al.* [40] presented a survey of secure multiparty computation for privacy-preserving genetic tests. The survey paper presents that researchers mostly use Edit distance to measure the similarity between genomic sequences.

In 2015, Cheon *et al.* [41] proposed a technique to compute Edit distance on encrypted data. They implemented the Edit distance algorithm suggested by Wagner and Fischer [42] on two encrypted sequences. To ensure the security, they used somewhat homomorphic encryption scheme (SWHE). Security wise this scheme is

very solid, but runtime wise is not very efficient.

In 2013, Beck and Kerschbaum [43] proposed a secure protocol for approximate string matching. Their protocol does not involve any third party, non-interactive, and the computation and communication complexity is linear. It also only reveals whether there is a match between two strings or not. They used Bloom filter to represent the strings, and the distance between the strings was calculated by measuring the distance between the Bloom filters. As the length of the Bloom filter can be adjusted to an appropriate size, their protocol can hide the size of the actual strings.

First, the input strings from each of the parties are divided into q-grams. Then each party inserts all the q-grams generated from a string S into a Bloom filter. They used only a single hash function for both of the Bloom filters influenced by the work of Papapetrou *et al.* [44] who concluded the optimal number of hash functions to determine the cardinality of a Bloom filter is 1. Then they calculated the approximate edit distance between the two strings by calculating the Hamming distance between the bit vectors of two Bloom filters $\mathcal{B}_1$ and $\mathcal{B}_2$ as:

$$d = |\mathcal{B}_1 \cup \mathcal{B}_2| - |\mathcal{B}_1 \cap \mathcal{B}_2|$$

Thus $d = 0$ means both of the strings are identical. The similarity among the Bloom filters is calculated by checking the indices of both of the bit vectors from the two parties. The privacy is ensured by using an additively homomorphic encryption scheme. This protocol provides security under a semi-honest model.

Zhang et al. [38] used *secret sharing* and *secure multi-party computation* for computing Edit distance between two sequences. Wang et al. [45] also used the *secure*

multi-party computation scheme to compute *edit distance* to find similar patients based on the inputs from two different parties.

Perl *et al.* [46] proposed a method for searching on a biomedical database to identify similar sequence. They generated a binary tree of Bloom filters using all the data from a database. If A is a database, then each biomedical sequence from A is divided into Q-grams and those Q-grams are inserted into the Bloom filter. A similar Bloom filter is generated the same way for the searched sequence, s. The search operation on this tree is similar to the binary search algorithm. Their algorithm ensured the security of the results through *homomorphic encryption* and *Obfuscated Bloom Filter (OBF)*. They completely outsourced the task of searching in a third-party cloud server. The runtime and communication complexity of their scheme are $\mathcal{O}(\log |A| + |s| + |R|)$ and $\mathcal{O}(|s|)$, where A, R, and s are the database, results set, and search term respectively.

In 2008, Jha *et al.* [47] presented a secured method for calculating Edit distance and Smith-Waterman similarity score [48] (used for sequence alignment) between two sequences. They used Yao's garbled circuit [10] based protocols to ensure the security of the computation. They proposed three different protocols with a variation of circuit representation for two party computation on genomic data.

In 2010, Rheinländer *et al.* [49] presented prefix tree indexing for similarity search based on *edit distance* and *hamming distance*. The authors used various filterings such as length filtering, frequency distance filtering, and Q-gram filtering. Without any filtering, using an ESTs dataset of 10000 records, it takes approximately 11, 20, 100, 1200 milliseconds for the threshold value $k \in 1, 2, 3, 8$ respectively. As the authors

do not encrypt the query or the dataset, their method is unable to provide any of the aforementioned security requirements: data privacy, query privacy and output privacy. Wang *et al.* [50] applied a prefix tree based searching index for secure similarity search using edit distance as the similarity metric.

3.4 Other relevant works

There are several other solutions that have been proposed to protect the privacy of both the outsourced data and the analysis. Although these works do not address the problems of secure count query, statistical analysis or secure similarity search, they target closely related problems and use different cryptographic techniques to ensure the security of the genomic data. Here, we mention some of these works.

Ayday *et al.* [51] proposed a method for storing genomic data at a storage and processing unit and then processing it for medical tests and personalized medicine operations. The computation on this shared data are conducted using *homomorphic encryption* and *proxy re-encryption*. Yang *et al.* [52] proposed a hybrid method by combining the ideas of *privacy by statistics* and *privacy by cryptography* for secure clinical data sharing and computation in cloud environment. Their hybrid search operation is conducted across both plaintext and ciphertext.

Chapter 4

Secure Count Query on Encrypted Genomic Data

In the count query operation, the aim is to know how many records in the database match a given query predicate (i.e., a certain combination of genotype and phenotype values). In the genetic association studies, the researchers try to identify the genes that are responsible for developing a particular disease in the human body. This association is determined by computing several statistical tests, like *Pearson Goodness-of-Fit* or *Chi-Squared Test*, *Linkage Disequilibrium*, *Estimation Maximization (EM)* and *Cochran-Armitage Test for Trend (CATT)* [30]. One of the prerequisites to compute the value of these statistical tests is to know how many records in the database matches with the genotypes and phenotypes specified in the query predicates. It has been shown in the literature that simple count queries can be used to compute various statistical tests. For example, doctors are interested in mining the potential biomarkers for Kawasaki disease [32] and it is accomplished by a statistical test (transmission

Algorithms	**Method**	**Trusted Entity**	**Privacy**		
			Data	**Query**	**Output**
Kantarcioglu *et al.* [27]	Paillier	Online	✓		
Canim *et al.* [28]	Cryptographic Hardware, AES	N/A	✓		✓
Our method	Paillier, GC, AES	Offline	✓	✓	✓

Table 4.1: Different properties of existing techniques for count query

disequilibrium test). This statistical test is performed by computing a series of count queries.

In this chapter, we propose a secure framework that enables outsourcing genomic data in a cloud server and then execute count query on it. As we have mentioned earlier, to the best of our knowledge, [27] and [28] are the only two works that have addressed the problem of secure count query operation on encrypted genomic data. However, none of these techniques can overcome the three challenges mentioned in Section 2.3 simultaneously or scalable for real-life applications. Table 4.1 presents a brief comparison of our method with these two existing solutions. This table summarizes different aspects of the proposed solutions such as cryptographic methodology, involvement of trusted entity during query execution and different types of security requirements satisfied by those methods.

4.1 Count Query

We can formally define count query operation as follow:

Definition 4.1.1. Given a database D and a query q, count query can be defined as

finding the number of tuples in D which satisfies the predicate θ in q. If d_i denotes one database tuple, the total count can be represented as: $|\ \{\forall i,\ d_i \in D\ \mid d_i \text{ satisfies } \theta\}\ |$.

For example, let's consider the following query submitted by a *researcher*:

```
SELECT COUNT(*) FROM Sequences WHERE
SNP2 = CC AND SNP3 = TT AND SNP5 = CC AND
AND Diagnoses = High blood pressure
```

Query 4.1: A sample query executed by the researchers

If we execute the above query on the data represented in Table 4.2, the answer will be 2 because only Case # 6 and 8 match the query predicates. We call the total number of SNPs specified in the query predicates as the *query size*. In the above query, the query size is 3.

Count query is a simple and straightforward operation if the data is stored as plaintext. Traditional database management systems (DBMS) support a built in operation for executing count queries. However, these DBMSs are not designed to execute count query operation on the encrypted data.

4.2 Genomic Data Representation

In this section, we present the format of the data used in this research. The database contains both genomic and clinical information including the DNA sequences, patients' response to a specific treatment, results of genetic testing, diagnoses, and prescribed medications. In this thesis, we use two types of datasets - datasets with phenotype and genotype information and datasets with genotype but

	Sequence						
Case	**SNP$_1$**	**SNP$_2$**	**SNP$_3$**	**SNP$_4$**	**SNP$_5$**	...	**Diagnoses**
1	AG	CC	TT	AG	CT	...	Headache, High cholesterol
2	AA	CC	CT	AG	CT	...	Arthritis
3	AG	CT	CC	AA	TT	...	Hair Loss, Mumps
4	AG	CC	TT	AG	CT	...	Nausea, Asthma, Cold
5	GG	CT	TT	GG	CC	...	Acid reflux
6	AA	CC	TT	GG	CC	...	High blood pressure
7	AG	CT	CT	AG	CT	...	Migraine
8	AA	CC	TT	GG	CC	...	High blood pressure
9	GG	CT	CT	AG	CT	...	Obesity
10	AG	CT	CT	AG	CT	...	Hypertension

Table 4.2: Data representation in the *Certified Institution (CI)*

without phenotype information. In real-life applications, we see examples of both kinds of datasets. In the dataset which contains only genotypes, all the patients are already separated into a case-control group based on their medical condition (e.g., individuals with or without a particular disease). On the other hand, some clinical datasets contain both the patients' genomic sequences and the genomic conditions. The genomic conditions are the diagnoses associated with the SNPs.

We assume that a sequence S consists of multiple SNPs, and we represent such a sequence as $S = \{a_1, a_2, ..., a_n\}$ where a_i represents an SNP. Table 4.2 represents an example of the format of the data that *data owners* send to *Certified Institution* (see more in Section 4.3). Here, each row represents genomic sequences and diagnoses for one single patient. Each of the SNPs $a_1, a_2, ..., a_n$ are represented in a single column. A SNP, a_i can be represented as a pair of nucleotides and it is common in genomic data analysis [28; 53]. The last column in Table 4.2 represents the diagnoses as genomic conditions.

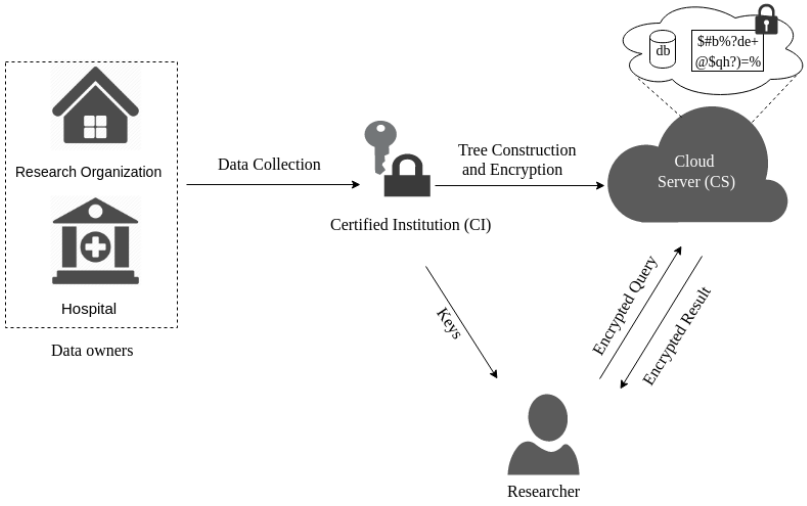


Figure 4.1: Architecture of our proposed solution.

4.3 System Design Overview

Figure 4.1 presents a general architecture of our proposed framework. As depicted in the figure, it incorporates four main participants: *Data Owners*, *Certified Institution (CI)*, *Cloud Server (CS)* and *Researchers*. Each entity is responsible for performing different specific tasks to make the overall system secure and functional. The roles performed by each of the entities are discussed below –

1. ***Data owners.*** *Data owners* consists of the institutions who agreed to share the genomic data they possess. These institutions might be any academic institutions, non-academic research organizations, government research agencies or health departments such as the main contributors of data samples to dbGap [54]. They send the genomic data to the *CI* in plaintext. Prior to sending the data to the *CI*, *data owners* process their data in a formerly agreed format.

2. ***Certified Institution (CI).*** The data shared by different *data owners* reside

in a database owned by a trusted entity which we call the *CI*. Any government institution such as National Institute of Health (NIH) in United States can play this role. The main responsibilities performed by *CI* are two:

(a) ***Generation of index tree.*** Upon receiving the data from the contributing *data owners*, *CI* builds an encrypted searchable version of the aggregate shared data and sends it to the *CS*. The search operation is basically performed on an encrypted *index tree*. In our proposed system, the *CI* builds only a single *index tree* that contains all the records from aggregate shared data and sends the encrypted version of the tree to the *CS*. For any addition and deletion of records, *CI* can update the tree accordingly.

(b) ***Management of the keys.*** Another responsibility of *CI* is to manage the keys used for encryption and decryption. It provides the key that the *researchers* use to decrypt the result of their query returned by the *CS*. The sensitive data stored at each node of the *index tree* are encrypted. *CI* supplies the necessary keys to the *CS* so that it can execute queries on encrypted data.

3. ***Cloud Server (CS).*** *CS* gets the encrypted version of the *index tree* and all the queries are executed on this tree. *CS* is responsible for handling all the communications with the *researchers*. The *researchers* send their encrypted query to *CS*, *CS* then executes this query and sends back the encrypted result to the *researchers*.

4. ***Researchers.*** *Researchers* might be any individual or organization who is

interested in executing query on the aggregate shared data residing in the *CS*. To execute query on the outsourced data, *researchers* need to obtain keys (both public and secret) from the *CI*. *Researchers* use the public key to encrypt their query and then send it to the *CS*. *CS* evaluates this query on the encrypted tree and sends back the encrypted result to the *researchers*. After decrypting this result using the secret key, the *researchers* get the final output.

4.4 Threat Model

Our goal is that the *CS* does not learn anything about the shared genomic data and both the *CI* and *CS* learn nothing about the query performed by the *researchers*. Furthermore, we want to ensure that the *researchers* do not infer any information from the data. We assume the *CI* to be trusted entity as it is responsible for the generation and encryption of the *index tree*. The *CI* can verify the identity of the individuals or organizations who apply for the access of the data before handovering them the keys. This role of verification performed by *CI* can be considered as the same as the *Data Access Committee (DAC)* of NIH [54].

In our proposed system architecture, we assume that the *CS* to be *semi-honest*, also known as *honest but curious* adversary [18]. This adversary correctly follows the protocol and does not have the intention to behave maliciously to produce the incorrect result. However, they may try to gather more information than necessary during or after the protocol execution. Therefore, we require the view of each party during protocol execution not to disclose any information. Thus, we assume that none of the parties *data owner*, *CI*, or the *CS* has any intention to behave maliciously

in the wish of generating incorrect output.

Our method is also designed based on the the following assumptions:

- We assume that the *CI* does not collude with the *CS* and *CS* also does not collude with the *researchers*. This is an essential requirement for guaranteeing data and query privacy.
- We assume that the keys received by the *researchers* from the *CI* are correct.

4.5 Basic System Design

In this section, we present our proposed model. At first, the *CI* creates an *index tree* from the datasets it receives from the *data owners*, encrypts it and then sends it to the *CS*. The *CS* uses this encrypted *index tree* to execute queries on behalf of a *researcher*.

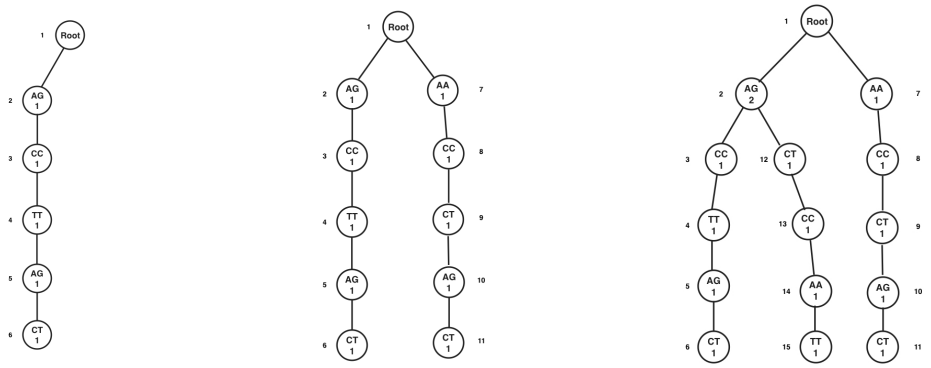
We first present our model for data without phenotype and then we discuss how to integrate the phenotype information into our index tree.

4.5.1 Genomic Data Without Phenotype

We will first discuss how to build the index tree without the phenotype and then the search procedure on this tree.

Generation of *Index Tree*

When the *CI* receives the data from the *data owners*, it first creates a search tree, T which we call the *index tree*, using the SNPs from the database D. There is only



(a) (b) (c)

Figure 4.2: Different states during the generation of *index tree*. Figure 4.2a, 4.2b, and 4.2c represents the tree after the insertion of the first, second, and third record respectively.

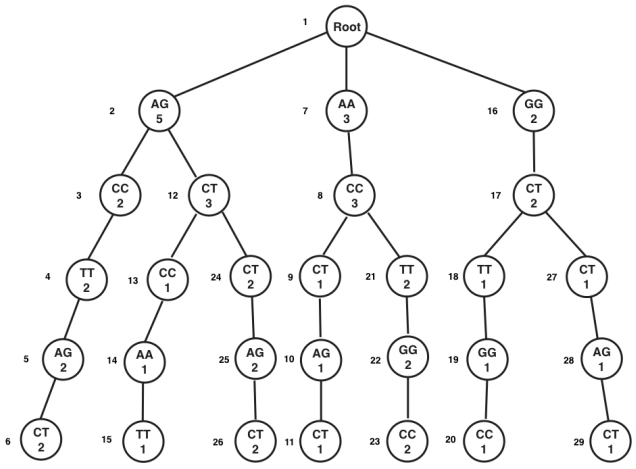


Figure 4.3: *Index tree* for Table 4.2.

one such tree in our system. After the creation of this tree, for each of the records from additional *data owners*, the *CI* only needs to create or update the nodes in T. For each record d_i in database D, the *CI* encodes each SNP as:

$$d_i^j = k : 1 \leq i \leq |D|;\ 1 \leq j \leq |d_i|;\ 1 \leq k \leq 16$$

Here, |D| = number of records in the database and $|d_i|$ = number of columns in each record. Then, *CI* checks if a node containing that SNP and SNP identifier is already in the tree or not. If not, then the *CI* creates a new node for that SNP. Otherwise *CI* just updates the corresponding existing node. Each node of T contains:

a) *sid*: the unique identifier for a SNP, which occurs at a particular position in the genome.

b) *val*: the actual SNP, which is encoded as $\{1, 2, ..., 16\}$ for each of the 16 possible sequences. In Figure 4.2 and 4.3, we have shown the actual SNP only for understanding purpose.

c) *count*: the number of times a SNP occur in that position.

d) *list*: the list of children (not shown in Figure 4.2 and 4.3).

We denote a node as σ and represent as, σ(*sid*, *val*, *count*, *list*). The tree T is generated in the following way:

At first there is only one node in the tree which is the root node. Beginning from this root node, for each of the records in the database we start creating new nodes in T. We denote a record as $d_i^j \in D$ where i indicates the record number and j indicates the column number. For each d_i^j, the child of the root node is the corresponding first column d_i^1, the child of the node containing d_i^1 is the second column d_i^2 and we continue to create the tree in this way. So, in the *index tree* the data from the first column is always on level 1, data from the second column is on level 2 and so on.

Example 1: *The generated tree, T after the insertion of first record d_1 from Table 4.2 is shown in figure 4.2a. Here, the first column, $d_1^1 = AG$ is inserted as the child of the root node. The second column, $d_1^2 = CC$ is inserted as the child of the node containing d_1^1 and so on. Each SNP occur only once in the first record. So, each node contains the count value* 1. *We can represent node # 2 as $\sigma_2(SNP_1, AG, 1, \langle CC \rangle)$.*

Now while inserting the second record, d_2 for each of the columns we first check whether the current column has already been inserted into a node in the corresponding level of T. If it has been inserted, we just increment the count value. Otherwise, we create a new node in that level to store d_2^j.

Example 2: *Continuing from Example 1. The tree, T after the insertion of second record d_2 is shown in Figure 4.2b. The first column for the second node d_2^1 is AA. We check if any existing node in T already contains this SNP at level 1. Here, root node has only one child AG. So, we create a new node and insert AA as the child of the root node at level 1 and the following columns are added in the above mentioned way. For the third record, the first column $d_3^1 = AG$ has already been inserted at level 1. So, we increment the value of count at node 2. Now for second column, d_3^2 there is no child node of node # 2 which contains CT. So, we create a new child node of node # 2 at level 2 and then add the remaining columns similarly.*

Figure 4.3 represents the *index tree* containing all the records from Table 4.2. All the nodes belonging to the same level represent a SNP all of which occur at a particular position of a genome which are actually represented as columns in Table 4.2. Each node in the tree T except the root node contains a value from a column. If there are θ_{n} number of columns in the database D, then the height of the index tree

T will be θ_{n}.

The building cost of the tree is $\mathcal{O}(mn)$ where, m = number of records in the database and n = number of different SNPs in the sequence. The features of this *index tree* can be listed as:

- If we traverse the tree starting from the root node to the leaf nodes, we get different SNP sequences belonging to the same record in the database. For example, if we consider first record of Table 4.2, the SNPs of this record are represented in the nodes 1, 2, 3, 4, 5 and 6. At each level, along with the SNP sequence, we also store the number of times that SNP sequence appears in that particular position of a genome. For example, in Figure 4.3, for SNP_1, AG occurs 5 times, AA occurs 3 times, and GG occurs 2 times. Considering AG as parent node for level 2, CC occurs 2 times — in this way all the nodes are created in T with each SNP and the number of their occurrence.

- We can reconstruct the original database record by traversing the corresponding nodes of T.

- For the addition or removal of records, we do not need to regenerate the tree, we can simply update or delete the data stored at each node.

- Unique SNP values at a particular level create new nodes and the following SNPs are added as the children of that node.

- One noticeable characteristic of this tree is that if multiple predicates are involved, i.e. more than one SNP sequences are present in the query, then the resulting count value is equal to the value of the count stored at the node which

matches the predicate located at the deepest level of the tree. So, if the *researcher* is interested in SNP positions x, y and z, and the position of x, y and z are such that $x < y < z$, then the count value is the value stored at node that represents the SNP sequence at position z. For example, consider the following query:

```
SELECT COUNT(*) FROM Sequences WHERE
SNP1 = GG AND SNP3 = TT AND SNP5 = CC AND
AND Diagnoses = Acid reflux
```

Query 4.2: A sample query executed by the researchers

Here, the value of count is 1 and it is the value that is stored at node that represents SNP_5 as this node is actually located at the deepest level of the tree among the nodes that matches the query.

Encrypting the *Index Tree*

After building the *index tree* T from the database, *CI* encrypts the *index tree* and then sends encrypted version of T to the *CS*. The detailed process can be elaborated as:

- *Key Generation:* The *CI* generates a key pair (pk, sk) for a semantically secure additively homomorphic encryption scheme (*Paillier Cryptosystem* [19]) which consists of the following algorithms:
  - *KeyGen*: a key generation algorithm, which generates a key pair (pk, sk) where pk is the public key and sk is the secret key.

  - *Enc:* an encryption algorithm, which takes as input a message m and encrypts it using the public key pk. This is denoted as $\xi_{\text{pk}}(m)$.
  - *Dec:* a decryption algorithm, which takes as input a ciphertext, c and decrypts it using the secret key, sk. This is denoted as $\xi_{\text{sk}}(c)$. Note that these encrypted records are not used in the search operation.

- *Encrypting the Index Tree: CI* uses the public key, pk to encrypt all the nodes in T. To make the overall search process fast enough while maintaining the security of the system, it only encrypts the sensitive attributes in each node. For each node σ in T, it does $\xi_{\text{pk}}(\sigma)$. After the encryption, each node is like $\sigma(sid, \xi_{\text{pk}}(val), \xi_{\text{pk}}(count), list)$. We represent the encrypted tree as $\widetilde{T}$.

- *Key Distribution:* Finally, *CI* sends $(pk, \widetilde{T})$ to the *CS*. *CI* also shares the key pair (pk, sk) with the *researchers*.

Encryption of Query

The *researchers* know about the format of the query they are allowed to perform. Once *CI* sends $(pk, \widetilde{T})$ to the *CS*, the *researchers* can execute their query on $\widetilde{T}$ stored in *CS*. A researcher encrypts her query q as $\xi_{pk}(q)$. Here, for the computation purpose, only *val* is encrypted and *sid* is kept in plaintext. So, we can represent the encrypted query as $\phi(sid, \xi(val))$. For example, after encryption, Query 4.1 will actually look like:

```
SELECT COUNT(*) FROM Sequences WHERE
SNP2 = +a=#?h AND SNP3 = z@0x* AND SNP5 = !?[h} AND
AND Diagnoses = #ir*q!
```

Query 4.3: A sample encrypted query executed by the researchers

Searching on Index Tree

Our system supports the count operation. The search process starts with the *researcher* sending the encrypted query ϕ to the *CS*. The *CS* needs to execute ϕ on $\widetilde{T}$ and find the number of records, which matches the SNPs in the query predicate. For this, it requires to perform search operation on $\widetilde{T}$ and find the intended nodes which contain the *count* values for corresponding *sid*s.

The main idea is to match the value of *val* stored in the indented nodes (the *sid* of these nodes matches with the *sid* of ϕ) which we denote as val_n with the corresponding value of *val* in the *researcher's* query which we denote as val_q. If they match, *CS* traverses the children of that node. This process continues until *CS* finds all the nodes for the corresponding query or *CS* finished searching all the nodes of $\widetilde{T}$. As both the val_q and val_n are encrypted and the encryption scheme we use is probabilistic, *CS* cannot determine whether those values match or not. The *CS* can send the encrypted value of val_n to the *researcher* and as they have the secret key, they can decrypt val_n and check the equality. But the problem of this approach is that the *researchers* would be able to determine the structure of the tree using multiple query operations.

To enable search in this scenario while ensuring less information leakage to the *researcher* and the *CS*, we execute an interactive protocol between them to check this equality. This equality checking is basically done using garbled circuit. The *CS* and

researchers compute this circuit via secure computation for each of the node which matches the value of *sid* in the ϕ. Here the *researcher* is the garbler and *CS* is the evaluator. Only the evaluator will know the output of the computation. As the val_n is encrypted, this value can be decrypted into the circuit before checking the equality, but this process is computationally expensive [55].

We choose to use random mask to avoid this decryption inside the garbled circuit. The idea is to use the additive mask to obscure the input of *CS* as the homomorphic property allows addition over encrypted data. We refer the additive mask we use as *noise* and denote it as μ. After the addition of the *noise*, the encrypted masked value of val_n is:

$$\widetilde{\delta} \;=\; \xi_{pk}(val_n) \;+\; \xi_{\mathrm{pk}}(\mu) \tag{4.1}$$

Here, $\mu \in \mathcal{M}$ where $\mathcal{M}$ is the message space and μ is random. *CS* then sends the resulting obscure value $\widetilde{\delta}$ to the *researcher*. *Researcher* get the masked value after the decryption as:

$$\delta \;=\; \xi_{sk}(\widetilde{\delta}) \;=\; val_n + \;\mu \tag{4.2}$$

Researchers then subtract the corresponding value of val_q from δ and get the noise as:

$$\mu' \;=\; \delta \;-\; val_q \tag{4.3}$$

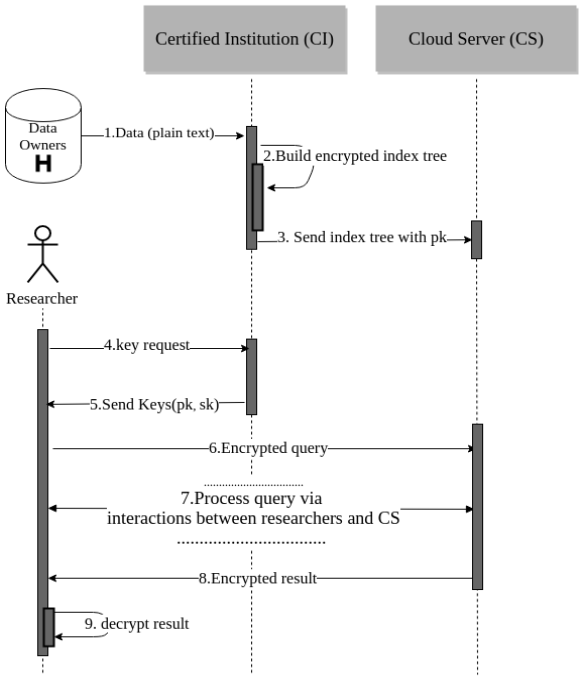


Figure 4.4: Sequence diagram of our proposed model.

As μ is random, the *researchers* will get random values for δ after decryption from Equation 4.2. As a result, though μ' is revealed to the *researchers* from Equation 4.3, they will not be able to infer useful information from it.

The *researcher* is the garbler of the circuit through which we check the equality. The input of the *researcher* is μ'. The input from *CS* (evaluator) to this circuit is the actual noise it added, μ. If the output of the circuit is true, then $\mu' == \mu$, which actually implies $val_n == val_q$. That means the SNP sequence in the *researcher's* query matches with the SNP sequence in the database. Only *CS* knows this output and *CS* then continues traversing the children of that node. This process continues until *CS* finds all the matched nodes for the corresponding query or *CS* searched all the nodes of $\widetilde{T}$.

Let q be the query consisting of the SNPs the *researchers* are interested in and r

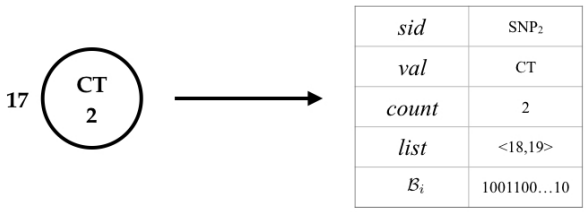


Figure 4.5: Information stored in a single node.

be the root node of T. Let *sid* be the SNP identifiers in q. Our search algorithm takes r and *sid* as input and returns the number of SNP sequences (*count*) that match the records in the database. Figure 4.4 summarizes each of the steps of our proposed method as sequence diagram.

4.5.2 Genomic Data with Phenotype

The tree described above is only capable of counting the number of genotypes at a particular position of the genome. To determine the effect of genotypes on the phenotypes, we need to incorporate the phenotype information at each node of the tree.

As we have mentioned earlier, Bloom filter can be used to check the membership of an element in a set. This property can be utilized to facilitate search operation over the *index tree*. For this, we have added a Bloom filter as the fifth component of each of the node in the tree to incorporate the phenotype information. The basic idea is to use a similar data structure like Bloom filter search tree [56].

Insertion into the Bloom filter.

The *CI* sets the domain of the hash functions $\mathcal{H}$ and common alphabet $\sum$ used in the Bloom filters. The domain of the common alphabet is the set of all possible phenotypes. For each of the SNPs in the genomic sequence, there will be one insertion of the corresponding phenotypes in the Bloom filter. Each phenotype from $\sum$ is mapped to a unique number and that number will be inserted into the Bloom filter.

Each node i in tree T except the root node and the leaf nodes contains a Bloom filter $\mathcal{B}_i$. While generating the *index tree*, all the phenotypes of the corresponding patient are inserted into the bloom filter at each of the nodes which represents the patients genomic sequences. So, $\mathcal{B}_i$ contains all the phenotypes associated with that node or its descendants. Figure 4.5 represents all the information stored in a single node.

Example 3. Figure 4.2a represents the tree after the insertion of the first record from Table 4.2. All the nodes except the root node will contain a Bloom filter where the mapped phenotype values for record #1 (Headache, High cholesterol) will be inserted. So, all the Bloom filters at node 2 to 6 will contain the same phenotype information.

Now, while inserting record # 3 (see Figure 4.2c), node 2 will contain the all the phenotypes of record # 1 and 3. But, the Bloom filter between node 3 and 6 will contain only the phenotype information of record # 1 (Headache, High cholesterol). Similarly, node 12 to 15 will contain only the phenotype information of record # 3 (Hair loss, Mumps).

The overall procedure to generate the *index tree* including the genotype and phe-

Algorithm 1 Algorithm for building *index tree*

Input: Root node, r and a database, D

Output: This algorithm will return an *index tree*, T

```
1: function buildTree(r, D_i^j)
2:     for each D_i do
3:         parent ← r
4:         a ← π(D_i^j)                          ▷ create new node for each D_{i,j}
5:         for each n ∈ T do
6:             if a.sid = n.sid and a.val = n.val then
7:                 n.count++
8:                 parent ← n
9:             else
10:                a.count ← 1
11:                parent.addChildren(a)
12:                parent ← a
13:            end if
14:            parent.bloomFilter ← ρ[j]
15:        end for
16:    end for
17:    return T
18: end function
```

notype is described in Algorithm 1.

Encryption of the Bloom Filter.

The length of each of the Bloom Filter at each node of the tree are the same. To encrypt the Bloom filter, we have used AES in CTR mode with a key size of 128 bits. This key, s is also generated by the *CI*. The *CI* encrypts each Bloom filter $\mathcal{B}_i$ as $\widetilde{\mathcal{B}}_i = \mathcal{B}_i \oplus s$. This encryption is done at the same time when other data at each node are encrypted as discussed in Section 4.5.1. After the encryption of the *index*

tree, the tree is sent to the *CS*.

Encryption of the Query.

Besides providing the *Researchers* the secret key s_k, the *CI* also provides the keystream s, and the hash functions $\mathcal{H}$ for Bloom filter. The phenotypes from the *researcher's* query are also mapped to the same unique numbers and then inserted into the Bloom filter. The generated Bloom filter has the same length as each of the Bloom filters at each node of the tree. We denote this Bloom filter generated at the *researcher's* end as $\mathcal{B}_q$.

Search with Phenotypes in the *Index Tree*.

To match the phenotypes at node i during the search we need to match the Bloom filter $\mathcal{B}_i$ at that node with the Bloom filter $\mathcal{B}_q$ which is generated from the *researcher's* query. For this purpose, we need to match the positions of $\mathcal{B}_q$ those hash to 1 with $\mathcal{B}_i$. If all the same positions of $\mathcal{B}_i$ are also hashed to 1, it means the phenotypes from the query match with the phenotypes stored in the tree node. Let, for the query Bloom filter $\mathcal{B}_q$, Z denotes all the positions to be checked. The *researcher* generates this Z and gives it as input to the circuit.

As the Bloom filter represented at each node of the tree are encrypted, to check the positions of any of these Bloom Filter, the *CS* first needs to decrypt it. The *CI* provides the decryption key s to *CS*. The *CS* decrypts the encrypted Bloom filter $\widetilde{\mathcal{B}}_i$ as $\mathcal{B}_i = \widetilde{\mathcal{B}}_i \oplus s$. Then each of the positions from Z will be checked inside the circuit.

The procedure for searching the *index tree* is described in Algorithm 2.

Algorithm 2 Algorithm for searching in the *index tree*

Input: Root node r of encrypted *index tree*, list of SNP identifiers in query ρ, indices of Bloom filter

Output: Resulting count value of the query

```
1: function SearchTree(node, index)
2:     count ← 0
3:     s ← ρ[index]
4:     if node.sid = s then
5:         c ← Rand()
6:         d ← ξ_pk(c)
7:         if ξ_sk(d) - i = c then          ▷ The equality checking is done using garbled circuit
8:             count ← b.getCount()
9:         end if
10:    end if
11:    children ← node.getChildren()
12:    for each child in children do
13:        count = SearchTree(child, index)
14:    end for
15:    return count
16: end function
```

4.6 Experimental Results

We have built a prototype of our privacy preserving system to evaluate its practicality and tested its performance on real datasets. The *CS* and the *CI* run on two different machines. The configuration of the *CS* is described in Table 4.3. The source code is written in JAVA programming language. For the simulation purpose, we considered the users separately.

We estimate the efficiency of our proposed method using the following parameters:

1. *Tree building time:* Time required to read the genomic data from mySQL

	Cloud Server
Operating System	Ubuntu 16.04
Processor	Intel Core i5-4590
Memory	8 GB
Database	MySQL

Table 4.3: Configuration of the *CS*.

database and build the *index tree*.

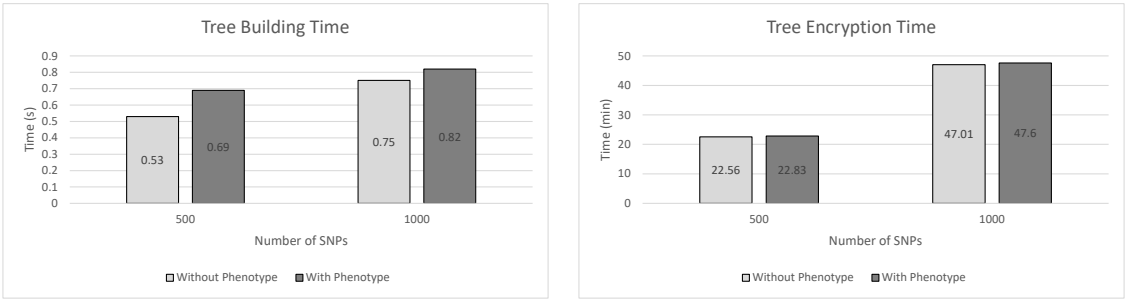
2. *Tree encryption time:* Time needed to encrypt the tree.

3. *Query execution time:* Time needed to execute a query submitted by a *researcher*.

4. *Communication overhead:* Bandwidth requirement between the evaluator (*CS*) and the garbler (*researcher*) in order to execute a count query.

We have implemented the cryptography building blocks that were described in Section 4. We investigated different garbled circuit libraries such as *FastGC* [57], *ObliVM-GC* [58], *JustGarble* [59] and used the *FlexSC* [60] library to implement the garbled circuits. We also used the *Paillier Cryptosystem* [19] to implement the *homomorphic encryption*.

We evaluated our system on real life dataset with 500 and 1000 SNPs. We experimented with three different query sizes that involved 10, 50 and 100 randomly selected SNP sequences. We also experimented with different number of records (10K to 50K) and found that the tree building time increases linearly with the increase of the number of records, while increasing the number of records does not directly influence the tree encryption time, query execution time and communication overhead.



(a) Tree Building Time.

(b) Tree Encryption Time

Figure 4.6: Tree building time and tree encryption time for count queries on datasets with different number of SNPs.

The is because the structure of the index tree remains same for a fixed number of SNPs. Due to the space limitation, we do not report the result of those experiments in this paper. For each experiment, we executed 10 runs and averaged the result over the runs.

A. Tree Building Time. We analyzed the time required to build the *index tree* for different datasets containing different number of SNPs. In this experiment, the number of records was the same and the number of SNPs we used in our experiment was 500 and 1000. Figure 4.6a plots the time required for building the tree with and without the phenotypes. As expected, the time increases with the number of SNPs. Also, we can notice from the graph that the additional time required to insert phenotype data into the Bloom filter is not very significant (< 0.2 seconds in both case).

B. Tree Encryption Time. Figure 4.6b plots the time required to encrypt the tree for datasets with 500 and 1000 SNPs. This majority of the time spent in this step is due to the encryption of the *count* values at each node. It is also evident

from the figure that encrypting the Bloom filter using the AES CTR scheme does not take very long. Our experiments show that the increase in number of records do not significantly impact the encryption time. This is due to the fact that the encryption time depends on the depth of *index tree* and depth of the tree in turn depends on the total number of SNP sequences in the dataset.

C. Query Execution Time. To calculate the query execution time, we executed a number of query of different sizes on the encrypted tree. The queries we used were determined by randomly selecting 10, 50 and 100 SNP sequences. The execution times of these queries on the tree are listed in Table 4.4. If we increase the query size on the tree without the phenotype, the execution time decreases. This is because the increase in query size also increases the probability of finding a matched node adjacent to the root. In the case of the tree with phenotype, we need to traverse all the nodes to get the exact count value. The query execution time is almost similar in this tree as only the leaf nodes contain the exact information.

# of SNPs	**500**			**1000**		
Query size	10	50	100	10	50	100
Without Phenotype	9.8	5.1	4.2	9.3	6.9	5.9
With Phenotype	77.8	69.7	67.1	130.8	128.7	130.1

Table 4.4: Query execution time. Times are measured in seconds.

Table 4.5 presents the comparison of the query execution time of our method with [27] and [28]. We report the times of [27] and [28] directly from the original articles. However, we have estimated the running time of [27] using the same platform as our

model. The query execution time for [27] is approximately 2.56 minutes for query size 10 (instead of 25 mins as reported in the original article). In this estimation, we have considered only the cost of cryptographic operations since they are much more expensive relative to other operations. The solution of [28] uses a specific hardware (IBM 4764 PCI-X SCPs) and most of the operations take place inside this trusted hardware. Hence, we expect the execution time to remain the same. However, the query execution time of our method is linear to the number of SNPs whereas it is linear to the number of records for both [27] and [28].

Query size	**10**	**20**	**30**	**40**
Kantarcioglu *et al.* [27]	25 min	27 min	28 min	30 min
Canim *et al.* [28]	20 sec	40 sec	60 sec	80 sec
Our method	2.4 sec	2.7 sec	1.5 sec	1.4 sec

Table 4.5: Comparison of count query execution time on a dataset of 5000 records, where each record contains 300 SNPs, for different query sizes.

D. Communication Overhead. The amount of data transferred between the *CS* and the *researcher* to execute the query with different query size is listed in Table 4.6. This overhead generally increases if the query execution time increases and decreases if the query execution time decreases. Both the query execution time and communication overhead generally depends on the number of nodes need to be accessed to execute a query. In the case of the tree with the phenotypes, we always need to find the corresponding leaf node to get the count value. So, the communication overhead is almost the same for different query sizes.

# of SNPs	**500**			**1000**		
Query size	10	50	100	10	50	100
Without Phenotype	0.09	0.05	0.04	0.08	0.05	0.05
With Phenotype	5.6	6.3	5.2	11.6	12.0	10.5

Table 4.6: Communication overhead in MB.

5. Storage Analysis. Table 4.7 lists the amount of spaces required to represent the original data, unencrypted tree and the encrypted tree. We stored the original data in the MySQL. The expansion in the encrypted tree size is due to the encrypting the data using the *Paillier cryptosystem* [19].

# of SNPs	**500**			**1000**		
Database	Original	Unencrypted Tree	Encrypted Tree	Original	Unencrypted Tree	Encrypted Tree
Without Phenotype	0.45	13.98	64.7	0.89	25.9	129.8
With Phenotype	0.61	16.13	95.94	1.1	31.11	188.2

Table 4.7: Size of original database, unencrypted tree and encrypted tree in MB.

4.7 Security Analysis

We assume that the security of our proposed system is compromised if the SNP sequences are revealed to any of the participants except the *CI* as it is the trusted entity. We also consider the participants' ability to infer information in different stages of the system. The leakage profiles of different participants in our proposed model are given below –

Leakage during the tree building and tree encryption phase: *CI* is only responsible for the generation and encryption of the *index tree* and is considered as a trusted entity. So the leakage to the *CI* is none in this phase. The *CS* cannot infer

any information during this phase as it only gets the encrypted *index tree*, $\widetilde{T}$.

Leakage to *CI*. The *CI* is not involved at all during the query execution, it's only responsibility is to provide the key pair (pk, sk) to the *researchers*. So, there is no information leakage to *CI* during the query execution.

Leakage to *researchers*. The leakage to the *researchers* is the final output which is the result of the query. *Researchers* also know the noise value, μ' from Equation 4.3 but μ' is a random number and uniformly distributed. Hence, the *researchers* cannot infer anything from the value of μ'. Note that we do not consider here any privacy leakage through the output. Such inference attack can be avoided using *differential privacy* and has been studied extensively in the literature [61].

Leakage to *CS*. The *CS* can know all the nodes in $\widetilde{T}$ which are accessed during the query execution, that means the tree traversal path is revealed to the *CS*. The traversal pattern depends on the query and it includes the path that either reaches the leaf or stops at an internal node. *CS* can learn about the SNP identifiers from a query but not the SNP sequences, because the SNP sequences are encrypted but the SNP identifiers are not. As the output of the circuit computation is only known to the *CS*, it can know which node actually contains which SNP identifier. But as the SNP sequences and all other information stored in that node are encrypted, *CS* cannot learn about any other values from that node.

4.8 Summary.

Our experimental results on different datasets by varying the number of records, SNPs and query sizes can be summarized as:

- Our method can effectively preserve both data privacy and data utility supporting large datasets by building an *index tree*. We observe that the time required to read the data from the database and build *index tree* using this data is linear. Also the tree encryption time does not have direct impact on the number of records.

- We have used a data structure similar to Bloom filter search tree to incorporate phenotype information and count the number of records.

- Our proposed model is also scalable for large datasets.

These characteristics make our proposed method a promising one for executing count query securely on encrypted genomic data.

Chapter 5

Secure Sequence Similarity Search on Encrypted Genomic Data

In personalized medicine, a medical unit (company or hospital) needs to check the medicine compatibility, that is to determine how suitable a particular medication would be for a patient. For this, it requires the user to share their genome with the medical unit. Also, the medical units (i.e. pharmaceutical companies) require checking the compatibility of their newly developed drugs by conducting medical tests on the user's genomic data. Due to the privacy concerns, the users do not want to share their genomic data with the medical units. The medical units also are reluctant to reveal the properties of their drugs under development.

To bridge the gap between the contradictory goals of the users and the pharmaceutical companies, a secured personalized medical solution is required. One of the steps to provide the personalized medical solution is to identify similar patients from a database of genome [62]. This is called similar patient matching. In this chapter,

we provide a secured method for the similar patient matching operation.

5.1 Similar Patient Matching

Similar Patient Matching (SPM) is the identification of similar patients from a large number of Electronic Medical Records (EMRs). The major focus of SPM is to implement a distance metric which can be used to measure the numerical similarity of attributes of patients from their medical and personal records. Many countries have recently emphasized the research on identifying similar patients in a number of healthcare projects, some countries has even already implemented it [63]. But, as these records contain highly sensitive personal and medical information of the patients, storage and computation on this data have significant privacy concerns. That is why, in SPM it is required to execute a query on the encrypted data which preserves the privacy of the patients

5.2 Hamming Distance

Definition 5.2.1. Let $u = \{u_1, u_2, \ldots, u_n\}$ and $v = \{v_1, v_2, \ldots, v_n\}$ be two strings over an alphabet Σ where $|u| = |v|$. The Hamming distance $d(u, v)$ between them can be defined as the number of positions where u and v have mismatched characters.

Mathematically it can be expressed as:

$$d(u, v) = \sum_{i=1}^{i=n} u_i \neq v_i$$

Case	Genomic Sequence	Diagnosis
1	AGCCTTA...	Negative
2	CACCCTA...	Negative
3	AGCTCCA...	Negative
4	AGCCTTA...	Negative
5	GGCTTTG...	Positive
6	AACCTTG...	Positive
7	AGCTCTG...	Positive
8	AACCTTG...	Positive
9	GGCTCTA...	Negative
10	AGCTCTG...	Positive

Table 5.1: Sample Genomic data representation

The Hamming distance between two sequences u and v satisfies the following conditions:

i. $d(u, v) \geq 0$ and $d(u, v) = 0$ if and only if $u = v$

ii. $d(u, v) = d(v, u)$

iii. $d(u, w) \leq d(u, v) + d(v, w)$ for any $u, v, w \in \Sigma$

5.3 Genomic Data Representation

Table 5.1 represents an example of the format of the data. Here, each row represents genomic sequences for one single patient. The last column indicates whether a genomic sequence is associated with cancer (positive) or not (negative). The dataset might contain other information about the phenotypes, but for keeping the structure of the data simple, we do not show those in Table 5.1.

5.4 Query Types

Our objective is to securely execute similar sequence search query operation in a database containing a large number of sequences. The *researchers* provide a reference sequence which is used to retrieve other sequences whose similarity against the reference sequence is less than or equal to a predetermined threshold k. Note that *researchers* can search for exact match by setting $k = 0$. We can formally define similar sequence query operation as follow:

Definition 5.4.1. Given a query of string s and a threshold k, a database representing collection of strings S, sequence similarity search returns all $s' \in S$ for which the difference, $d(s, s') \leq k$.

For example, let's consider the following query submitted by a *researcher*:

```
SELECT (*) FROM Sequences
WHERE s = AGCCTGT AND k = 2
```

Query 5.1: A sample query executed by the researchers

If we execute the above query on Table 5.1, *researchers* will receive $AGCCTTA$ and $AGCCTTA$ as the answer of the query because only Case # 1 and 4 have hamming distance ≤ 2 with the query sequence $AGCCTGT$ ($d = 2$ in both case). Receiving these similar sequences based on a threshold value helps the *researchers* to determine the similar patient and predict phenotype.

5.5 System Design Overview

The architecture of our proposed system is the same as presented in Figure 4.1 in Chapter 4. So, again there are four entities: *Data Owners*, *Certified Institution (CI)*, *Cloud Server (CS)* and *Researchers*.

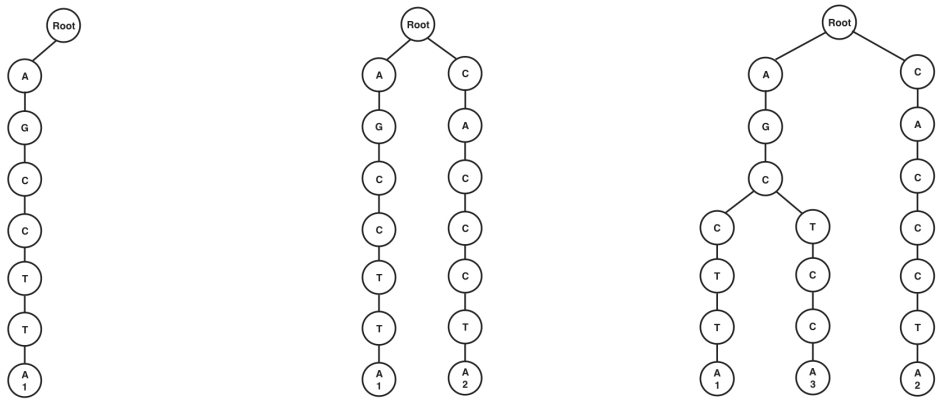
The *Data Owners* shares the genomic data with the *CI*. The *CI* builds an encrypted searchable version of the aggregated shared data which we call *prefix tree* and sends it to the *CS*. The search operation is basically performed on this encrypted *prefix tree*. *CI* also manages the encryption keys. It shares keys with the *CS* and *researchers* to facilitate secure search on the *prefix tree*. All the queries submitted by the *researchers* are evaluated against the nodes of this encrypted tree at *CS*. *CS* sends the encrypted result of the query to the *researchers* who can decrypt the result using the private keys from the *CI*.

5.6 Threat Model

The threat mode for this problem is also the same as the threat mode described in Section 4.4. Our objective is to provide the privacy of the data, query and output. We assume that the *CI* to be a trusted entity as it is responsible for the generation and encryption of prefix tree. The *CS* and *researchers* in our system are *semi-honest*.

Our security of our proposed system also require that the view of each entity during the protocol execution not to disclose any information or collaborate with one another. So, our method is designed based on the the following assumptions:

- We assume that the *CI* do not collude with the *CS* and *CS* also do not collude



(a) (b) (c)

Figure 5.1: Different states during the generation of the *prefix tree*. Figure 5.1a, 5.1b, and 5.1c represents the tree after the insertion of the first, second, and third record respectively.

with the *researchers*. This is an essential requirement for guaranteeing query privacy.

- We assume that the keys received by the *researchers* from the *CI* are correct.

5.7 Basic System Design

In this section, we present our proposed model. At first, *CI* creates a compressed *prefix tree* from the aggregated datasets of *data owners*, then encrypts it, and finally sends it to *CS*. The *CS* uses this encrypted compressed *prefix tree* to execute queries.

5.7.1 Building Compressed *Prefix Tree*

When *CI* receives the data from the *data owners*, it first creates a search tree, T which we call *prefix tree*, using the SNP sequences from the database D. There is only one such tree in our system. After the creation of the tree, for the records from each additional *data owner*, *CI* only needs to create or update the nodes in the T.

As discussed in Section 2.2.1, there are four nucleotides $\{A,\ C,\ G,\ T\}$. We can represent each nucleotide as a pair of bits. For example, we map $\{A,\ C,\ G,\ T\}$ to $\{00,\ 01,\ 10,\ 11\}$. Then each genomic sequences will be represented as a series of binary values. *CI* checks if a node containing a nucleotide is already in the tree or not. If not, then the *CI* creates a new node for that nucleotide. Otherwise *CI* just updates the corresponding existing node. Each node of T contains:

a) *val*: the actual nucleotide, which is encoded as $\{00, 01, 10, 11\}$ for each of the 4 possible sequences. In Figure 5.1 and 5.2, we have shown the actual nucleotide only for understanding purpose.

b) *list*: the list of children (not shown in Figure 5.1 and 5.2).

Each record $d_i \in D$ maps to a leaf node such that if we concatenate all nodes along a path from root to leaf, we will get d_i. Each leaf node of T contains:

a) *val*: the actual nucleotide, which is encoded as $\{00, 01, 10, 11\}$ for each of the 4 possible sequences.

b) *id*: we keep a unique identifier to each record inserted in tree T so that we can reference the original record using this identifier later. If several records in the

database D contain the same sequence, the corresponding leaf node will contain multiple *id*s.

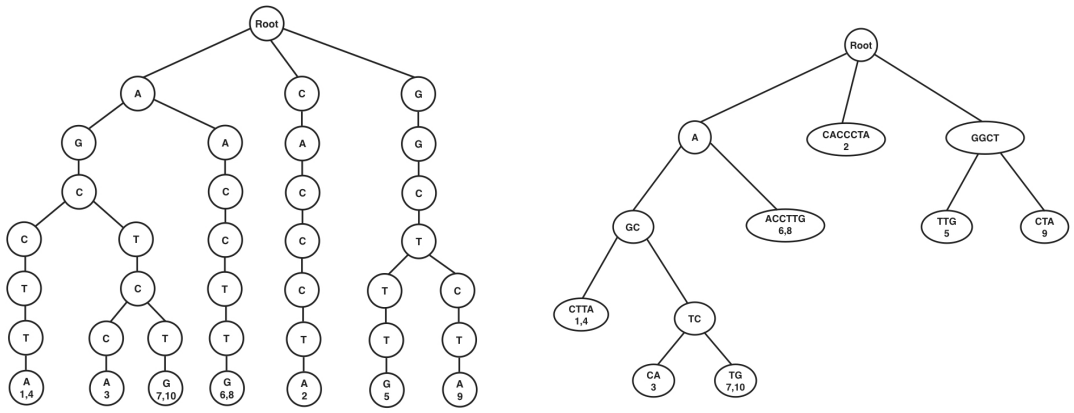
We denote a node as σ and represent all the nodes as $\sigma(val, list)$ except the leaf node. The leaf node can be denoted as $\sigma(val, list)$. The tree T is generated in the following way:

At first, there is only one node in the tree which is the root node. Beginning from this root node, for each of the records in the database we start creating new nodes in T. We denote a record as $d_i^j \in D$ where i indicates the record number and j indicates the nucleotide position. For each d_i^j, the child of the root node is the corresponding first nucleotide d_i^1, the child of the node containing d_i^1 is the second nucleotide d_i^2 and we continue to create the tree in this way.

Example 1: *The generated tree, T after the insertion of the first record d_1 from Table 5.1 is shown in figure 5.1a. Here, the first nucleotide, $d_1^1 = A$ is inserted as the child of the root node. The second nucleotide, $d_1^2 = G$ is inserted as the child of the node containing d_1^1 and so on. We can represent this node as $\sigma_2(A, \langle G \rangle)$.*

Now while inserting the second record, d_2 for each of the nucleotides we first check whether the current nucleotide has already been inserted into a node in the corresponding level of T. If it has been inserted, we just traverse its children. Otherwise, we create a new node in that level to store d_2^j.

Example 2: *Continuing from Example 1. The tree, T after the insertion of second record d_2 is shown in Figure 5.1b. The first nucleotide for the second node d_2^1 is C. We check if any existing node in T already contains this nucleotide at level 1. Here, root node has only one child A. So, we create a new node and insert C as*



(a) *Prefix tree.* (b) Compressed *Prefix tree.*

Figure 5.2: *Prefix tree* and compressed *prefix tree* generated from the data represented in Table 5.1.

the child of the root node at level 1 and the following nucleotides are added in the above mentioned way. For the third record, the first nucleotide $d_3^1 = A$ *has already been inserted at level 1. So, we traverse the next node* (d_3^2) *and find the same node value* G*. Similarly, the next node* (d_3^3) *also matches the value* C*. Now for the fourth nucleotide* (d_3^4)*,* C *does not have any child which contains* T*. So, we create a new child node of* C *at level 4 with the value* T *and then add the remaining nucleotides similarly. Figure 5.2a represents the prefix tree containing all the records from Table 5.1.*

Algorithm 3 provides pseudocode for building the *prefix tree*. The building cost of the tree is $\mathcal{O}(mn)$ where, m = number of records in the database and n = length of each sequence. The features of this *prefix tree* can be listed as:

- If we traverse one node at each level starting from the root node to a leaf node, we get different SNP sequences belonging to the same record in the database.

Algorithm 3 Algorithm for building *prefix tree*

Input: Root node and the database (r, D_i^j)

Output: This algorithm will return an index tree, T

for each D_i **do**
  $parent \leftarrow r$
  $a \leftarrow \pi(D_i^j)$ ▷ create new node for each $D_{i,j}$
  for each $n \in T$ **do**
    if $a.val \neq n.val$ **then**
      $parent.addChildren(a)$
      $parent \leftarrow a$
    end if
  end for
end for
return T ▷ The generated tree

- We can again reconstruct the original database record by traversing the corresponding nodes of T.
- For the addition or removal of records, we do not need to regenerate the tree, we can simply update or delete the data stored at each node.
- Unique SNP values at a particular level create new nodes and the following SNPs are added as the children of that node.

After building the *prefix tree*, we compress the suffixes and infixes of the *prefix tree*. Figure 5.2b represents the *compressed prefix tree*. For example, $ACCTTG$ is a suffix compression and $GGCT$ is an infix compression in figure 5.2b.

5.7.2 Encrypting the Compressed *Prefix Tree*

After building the *prefix tree* T from the database, *CI* encrypts the *prefix tree* and then sends encrypted version of T to the *CS*. The detailed process can be elaborated as:

- *Key generation:* The *CI* generates a key, an IV and a counter CTR value for each node. The encryption key ξ_k remain same for all the nodes of the compressed prefix tree.

- *Encrypt the compressed prefix Tree:* *CI* uses the key, IV and CTR value to build a *keystream* by executing $\xi_k(IV||\ CTR_i)$. *CI* build a unique *keystream* to encrypt all the nodes in T. For each node σ in T, *CI* encrypts the node by $keystream \oplus (\sigma)$. We represent the encrypted tree as $\tilde{T}$.

- *Share:* Finally, *CI* sends $\tilde{T}$ to the *CS*. *CI* also shares the secret key with the *researchers*.

5.7.3 Searching on Encrypted *Prefix Tree*

Once *CI* sends $\tilde{T}$ to the *CS*, the *researchers* can execute his query on $\tilde{T}$ stored in *CS*. The researcher encodes his query q by mapping $\{A,\ C,\ G,\ T\}$ to $\{00,\ 01,\ 10,\ 11\}$. The query also includes the threshold value k. The search process starts with *researchers* sending the encoded query ϕ to the *CS*. The *CS* needs to execute ϕ on $\tilde{T}$ and find the similar sequences within the threshold k.

The main idea is to measure the length of the value stored in the indented nodes (denoted as len_n) and use the same length to prepare corresponding *researcher's*

query (denoted as len_q). We denote the indented node's value as val_n and query value (up to len_q) as val_q. If the hamming distance between val_n and val_q is less than or equal to the given threshold k, *CS* traverses the children of that node. This process continues until *CS* finds all the nodes for the corresponding query that satisfies the hamming distance k.

Example 3: *CS and researchers execute a secure interactive protocol via garbled circuit. Query processing starts with the researcher sending keystream (derived from secret key), and threshold value k to the garbled circuit. CS performs a pre-order traversal of the encrypted tree and sends encrypted node value δ to garbled circuit. Inside the garbled circuit, at first we achieve the original node value val_n by computing $\delta \oplus$ keystream. Suppose from Figure 5.2b, we achieve* 00 *after the decryption. Now, if the query is AGCCTGT ($\phi =$ 00100101111011), only the corresponding length of the decrypted value (length of* $00 = 2$*) will be considered as the query length. The hamming distance between the updated query (*00*) and decrypted value (*00*) will be calculated inside the garbled circuit. Then the calculated hamming distance ($dist = 0$) is compared inside the garbled circuit whether $dist \leq k$ (suppose $k = 2$). As $dist < k$, we traverse the children of the current node. Similarly, we traverse the children of the next node with the updated query q (*100101111011*) and updated threshold k ($2-0 = 2$) if $D_{hd}(q, val_n) \leq k$. This process will continue until the leaf node. If $dist > k$ at any iteration, we stop traversing children of the current node and start the similar process traversing the next child of the root.*

To calculate Hamming distance while ensuring less information leakage to the *researcher* and the *CS*, we execute a secure interactive protocol between the parties.

The Hamming distance calculation is done using the garbled circuit. Here the *researcher* is the garbler and *CS* is the evaluator. Only the evaluator will know the output of the computation. As the val_n is encrypted, it will be decrypted inside the garbled circuit before calculating the hamming distance.

To decrypt inside the garbled circuit using *CTR* method, we choose to use *XOR* operation between the val_n and *keystream*. The detailed process can be elaborated as –

- Hamming distance is calculated between the decrypted node value (val_n) and the query with equal length. This hamming distance calculation is done using the garbled circuit. We denote this distance as $dist$.
- To ensure *CS* does not know any knowledge about the calculated Hamming distance, we add a random mask μ with the calculated $dist$. We denote this as $k' = dist + \mu$.
- If $k' \leq k + \mu$, then only the children of that node will be traversed. We need to save the old distance by $old\_dist = k' - \mu$. All the operations such as addition, subtraction are done by the garbled circuit.
- Again by traversing the child node, new distance is calculated with the child node and the remaining query and it needs to satisfy $dist + old\_distance \leq k$. This process continues until *CS* finds all the matched nodes for the corresponding query and k or *CS* searched all the nodes of $\tilde{T}$.

Algorithm 4 provides the pseudocode for the similar sequence search operation on T. Let q be the query, r be the root node of T, k be the threshold value, and s be

Algorithm 4 Searching similar sequence in the tree

Input: Root of encrypted compressed *prefix tree*, query, and threshold value (r, q, k)

Output: Resulting similar sequences (S)

```
1: a ← r.getChildren()
2: while a ≠ {φ} do
3:     b ← a.pop()
4:     if b.getChildren() ≠ φ then
5:         buildTree(node.l, B, id)
6:         Update query (q') up to length of b.val
7:         if D_hd(q', b.val) <= k then
8:             Update k' = k − D_hd(q', b.val)
9:             Algorithm 2 (r, q', k')
10:        end if
11:    else if b.getChildren() = φ and D_hd(q, b.val) ≤ k then
12:        S ← S + b.val
13:    end if
14: end while
15: return S
```

the database sequences. Our search algorithm takes r, q, k as input and returns the similar sequences (S) that satisfy $D_{hd}(q, s) \leq k$. Figure 5.3 summarises each of the steps of our proposed method as sequence diagram.

5.8 Experimental Results

We have implemented our proposed method for secure similar sequence search problem and evaluated its performance on both real and synthetic datasets. The *CS* and the *CI* run on two different machines. Both of them were Intel Core i5 3.3 GHz processors with 8 GB RAM, running Ubuntu Linux 16.04. The source code is written in JAVA programming language. For the simulation purpose, we considered the users

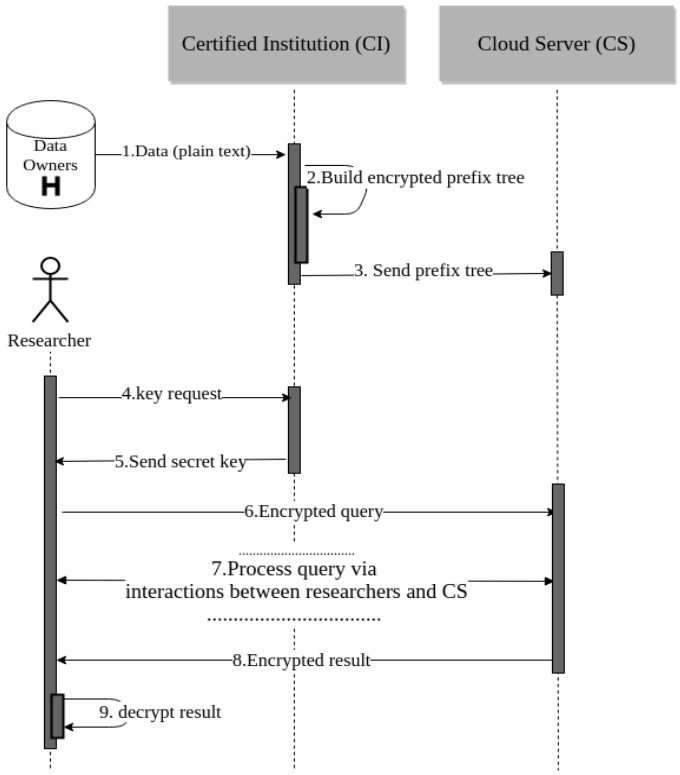


Figure 5.3: Sequence diagram of our proposed model.

separately.

We estimate the efficiency of our proposed method using the following parameters:

1. *Data read and tree building time:* Time required to read the genomic data from mySQL database and build *prefix tree*.

2. *Tree encryption time:* Time needed to encrypt the *prefix tree*.

3. *Query execution time:* Time needed to execute a query submitted by th *researchers*.

4. *Communication overhead:* Bandwidth requirement between the evaluator (*CS*) and garbler (*researchers*) in order to process a query.

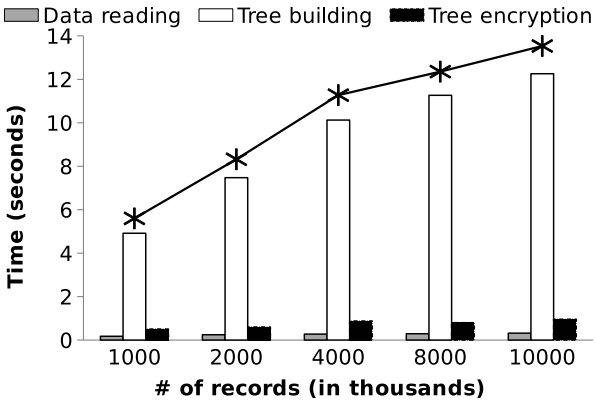
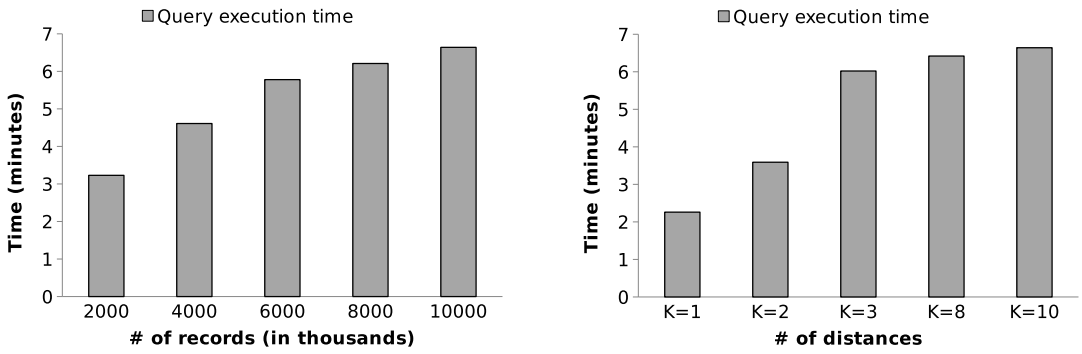


Figure 5.4: Data read and *prefix tree* building time.

We have implemented the cryptography building blocks of garbled circuit and AES in CTR mode described in Chapter 2. We investigated different garbled circuit libraries and used the *FlexSC* [60] library to implement the garbled circuits.

To evaluate the performance of our method on real life dataset, we used the dataset available from the iDash competition 2015 [64] where there are 400 different participants divided into case and control groups. As the real-life dataset was not large enough to evaluate the scalability of our proposed model, we generated different synthetic datasets varying the number of records (between 2K to 10K) by randomly adding records to the iDash competition 2015 [64] dataset. For each experiment, we executed 10 runs and averaged the result over the runs.

Data read, tree building and encryption time. To determine the scalability of our proposed system, we analyzed the time required for different datasets containing a different number of records. Figure 5.4 plots the time required for reading the data from the database, building *prefix tree* using this data, and encrypting the prefix tree. Here we vary the number of records from 2K to 10K, where each record contains



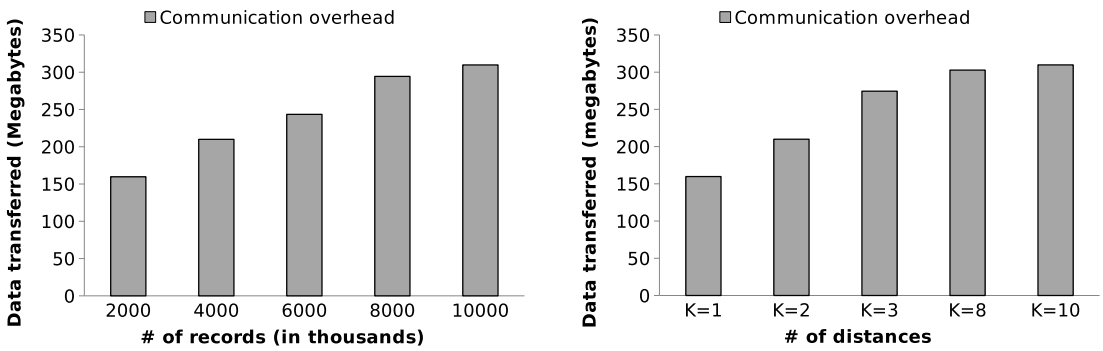
(a) (b)

Figure 5.5: Figure 5.5a shows the query execution time on different datasets with different number of records and a fixed hamming distance $k = 10$. Figure 5.5b shows the query execution time on a dataset of 10000 records and different hamming distances $k \in 1, 2, 3, 8, 10$.

500 nucleotides. As expected, the time increases linearly with the increase of number of records. Thus, when we increase our number of records to 10000, the data read, compressed *prefix tree* building time, and tree encryption time increases to approximately 0.5 seconds, 12 seconds, and 1 second respectively. Note that, compressed *prefix tree* building takes significant time than the data read and tree encryption time.

Query Execution Time. Figure 5.5 shows the query execution time based on two different parameters. These parameters are: *a*) number of records in the dataset and *b*) number of hamming distances (k). The size of the datasets and the value of k have an effect on the query execution time. As expected, the time increases linearly with the increase of number of records and the value of the hamming distance (k).

Communication overhead. Figure 5.6 shows the amount of data transferred between the *researchers* and the *CS* during the evaluation of the garbled circuits.



(a) (b)

Figure 5.6: Figure 5.6a shows the communication overhead on different datasets with different number of records and a fixed hamming distance $k = 10$. Figure 5.5b shows the communication overhead on a dataset of 10000 records and different hamming distances $k \in 1, 2, 3, 8, 10$.

In these experiments, we similarly considered two different parameters as mentioned earlier. As expected, the overhead increases linearly with the increase of number of records and increasing number of hamming distance (k).

5.9 Security Analysis

The security of the system is compromised if any sequence is revealed to the *CS*. On the other hand, researchers are only allowed to know the final result of their query. Following we discuss various leakages of our proposed model.

Leakage during the tree building and tree encryption phase: *CI* is only responsible for the generation and encryption of the *BF-tree tree* and is considered as a trusted entity. So the leakage to the *CI* is none. The *CS* and *researchers* cannot

infer any information during this phase.

Leakage to *CI*. The leakage to the *CI* is none as it is not involved during the query execution. The only responsibility of *CI* is to provide the secret key to the *researchers*.

Leakage to *researchers*. The leakage to *researchers* is the final output which is the result of the query. Note that we do not consider here any privacy leakage through the output. Such inference attack can be avoided using *differential privacy* and has been studied extensively in the literature [61].

Leakage to *CS*. The *CS* can know all the nodes in $\widetilde{T}$ which are accessed during the query execution, that means the tree traversal path is revealed to the *CS*. Depending on the result of the query, the tree traversal pattern includes either the paths reaching the leaves or the paths stopping at some internal nodes. *CS* can learn about the *researchers'* interested hamming distance k from a query. As the output of the circuit computation is only known to the *CS*, it can know which node is actually accessed during the query execution. But as the sequences and all other information stored in that node are encrypted, *CS* cannot learn about any other values from that node.

5.10 Summary

In this chapter, we have presented a secure and efficient method for similar sequence search on encrypted data. The proposed method constructs a compressed *prefix tree* from the aggregated genomic data and then outsources it to the third party cloud server. By employing a secure interactive protocol, the cloud server can

traverse the nodes of the tree and execute Hamming distance. We have demonstrated that our model does not reveal any sensitive genomic data during the data processing as well as query execution phase.

Chapter 6

Identification of Similar Patients with Edit Distance Approximation

In this section, we propose a model for secure outsourcing of genomic data and then execute query to find similar sequence on this outsourced data. We pre-filter the search result using a Bloom filter tree. The similarity between two sequences is measured by calculating the edit distance.

6.1 Edit Distance

In this current era, genomic sequence analysis can reveal a lot of private information of an individual. Sequence analysis is used to understand and discover the relationships between sequences by aligning them appropriately. Edit distance (also known as Levenshtein distance) algorithm is one of the most frequently used methods in genomic sequence analysis and to preserve the privacy of the individuals, this

operation is performed on the encrypted data.

Definition 6.1.1. Let S_1 and S_2 be two strings over an alphabet Σ. Let the lengths of S_1 and S_2 be n and m, respectively. The edit distance between S_1 and S_2 (denoted by $d(S_1, S_2)$) is the minimum number of edit operations (insertions, deletions and substitutions) required to convert S_1 into S_2.

Typically, the cost of insertion and deletion operations is 1 and substitution is 2. Calculate the edit distance between the genomic sequences is one of the major steps in the SPM.

6.2 System Design Overview

The architecture of our proposed system is the same as presented in Figure 4.1 in Chapter 4. So, again there are four entities: *Data Owners*, *Certified Institution (CI)*, *Cloud Server (CS)* and *Researchers*.

The *CI* is responsible for the collection of the data from the *data owners*. Using all the collected data it builds a Bloom filter tree (see details in Section 6.5). The encrypted version of this tree is made available to the *researchers* by hosting it in a third party cloud server. All the query will be executed on this encrypted tree. This tree is very easy to update or delete new records. Each query submitted by the *researchers* are executed using secured function evaluation facilitated by garbled circuit.

6.3 Threat Model

In this model our objective is to protect the privacy of the data, query and the output as described in Chapter 1. In our model we assume that the both the *CS* and *researchers* are semi-honest. The *CI* is the trusted entity responsible for data collection, tree generation and the management of the encryption keys.

So, our method is designed based on the the following assumptions:

- We assume that the *CI* do not collude with the *CS* and *CS* also do not collude with the *researchers*. This is an essential requirement for guaranteeing both data and query privacy.
- We assume that the keys received by the *researchers* from the *CI* are correct.

6.4 Genomic Data Representation

The heredity information of an organism is encoded in a genome which contains both the gene and the non-coding sequences of DNA or RNA. Each DNA molecule in human genome contains 2 biopolymer chains called a *double helix* which is formed by the spiral arrangement of nucleotides. These nucleotides are represented as *A, C, G, T* which are the acronyms of *Adenine*, *Cytosine*, *Guanine*, and *Thymine* respectively. Each DNA molecule in the human genome contains 3 billion nucleotides and most of them ($\sim 99\%$) are the same in all people.

Storing such a large amount of data requires a large amount of storage. As most of the ($\sim 99\%$) DNA sequences between two individuals are identical, storing all of the genetic data is redundant. That is why a specific format named Variant Call Format

(VCF) is used for storing the gene sequence variations. The variations are determined by aligning a human genome with a reference genome. Information stored in the VCF file includes the type of variations (insertions, deletions or substitutions), the change in nucleotide(s) and the genetic position where variation occurs. The whole sequence can be reproduced again by aligning the short sequences from the VCF file with the reference genome.

In our model, all the genomic sequences are aligned using a reference genome and then stored in VCF files.

6.5 Basic System Design

A simple solution to find similar patients from a large data set is to compare the genomic sequence with each genomic sequence of the large data set. Dynamic programming algorithm can compute the edit distance between two sequences S_1 and S_2 in time $\mathcal{O}(mn)$ where m and n are the lengths of the strings. Though this approach guarantees to find the exact edit distancee, this process is inefficient and time-consuming for a number of reasons. *First*, The length of the genomic sequences to be compared are enormous. Also, to find a similar sequence on a database containing n number of records, it requires n number of comparisons. *Second*, to ensure the security of the genomic data, this computation has to be performed using cryptographic techniques which tend to be expensive and requires more time than the same computation on the plaintext.

We pre-filter the search result by using an indexing data structure called Bloom filter tree (BF-tree). The bloom filters represent each of the genomic sequences of the

patients. This approach enables us to identify similar patients from large datasets efficiently, without significant loss of accuracy.

6.5.1 Edit Distance Approximation

Computing edit distance between two individuals over the whole genomes which consist of 3 billion DNA bases are very expensive. As mentioned earlier, the dynamic programming algorithm computes the distance between two sequences in time $\mathcal{O}(mn)$. Focusing only on the variations between two genomes, we can approximate the edit distance between two sequences. Again, the reason being most of the genomic sequences ($\sim$ 99%) are similar and most of the edits ($>$ 95%) recorded in the VCF file with respect to the reference genome occur at the non-adjacent regions. This information can be utilized to design an efficient approximation of edit distance scheme between two human genome. Wang *et al.* [45] adopted the same approach to determine the edit distance.

The first step of our algorithm is to generate the edit sequences which are essentially the variations between genomes. Both parties (*CI* and *researcher*) use the same reference genome to generate the edit sequences. In practice, all the edit sequences derived from an individual's genome are stored in a single VCF file. The input to the Bloom filter is all the single-character edits recorded in the VCF file. If an edit sequence contains multiple characters, we convert it to multiple single-character edits the same way it was done by Wang *et al.* [45]. In brief, conversion of each multi-character edit to multiple to single character edits for the insertions and deletions are done in the following way:

Insertion of a string, $S = \{s_1, s_2, \ldots, s_n\}$ of length n at position, pos is converted as $(pos, ins, 1, s_1)$, $(pos, ins, 2, s_2), \ldots, (pos, ins, n, s_n)$. Also, deletion of a string S at position, pos is converted as (pos, del, s_1), $(pos + 1, del, s_2), \ldots, (pos + n - 1, del, s_n)$. Here, ins and del are the short forms of the operations insertion and deletion respectively. We do not need any conversion for substitutions as they are already defined with respect to a single character.

Let, A = ACCAGT and B = ACCAAT be two genomic sequences and Ref = ACTAAGT be a reference genome. The minimum edits that require to convert Ref to A are $A' = \{(3,\ sub,\ C),\ (4,\ del,\ A)\}$ and Ref to B are $B' = \{(3,\ sub,\ C),\ (6,\ del,\ G)\}$. We use the formula to calculate the edit distance is $d(A', B') = |A' \cup B'| - |A' \cap B'|$. For this example, $A' \cup B' = \{(3,\ sub,\ C),\ (4,\ del,\ A),\ (6,\ del,\ G)\}$ and $A' \cap B' = \{(3,\ sub,\ C)\}$. Therefore, $d(A', B') = 3 - 1 = 2$, which coincides with the Levenshtein distance between A and B. Figure 6.1 depicts the calculation of edit distance for this example. The procedure to insert the single-character edits to a Bloom filter is discussed in Section 6.5.2.

But, this approach does not work for some sequences. For example, if A = GTATCAGT, B = GCATCAGT and Ref = GCAACTGT. Then the minimum edits to convert Ref to A are $A' = \{(2,\ sub,\ C),\ (4,\ sub,\ T)\}$ and Ref to B are $B' = \{(4,\ sub,\ T),\ (6,\ sub,\ A)\}$. Therefore, $d(A', B') = 4$ in our method whereas theoretically it is actually 1. This error is due to converting A to B through the reference sequence as both of them are derived from the reference sequence in different ways.

Wang *et al.* [45] specifies that scenarios like this occur very rarely due to the

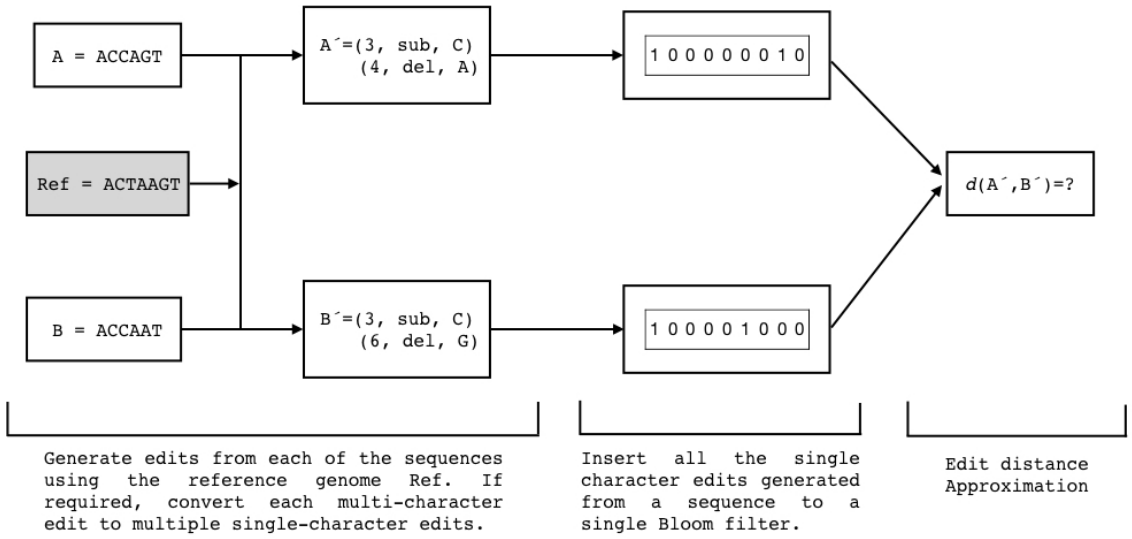


Figure 6.1: Calculation of edit distance using reference genome

fact that almost 90% of the edits are short edits. The problematic edits due to overlapping insertion and deletions almost never happen in practice. Also, most of the edits ($80 \sim 99\%$) occurred are substitutions.

6.5.2 Bloom Filter Representation

We insert each of the single character edit operations generated in the previous step in a Bloom filter. To find the similar patients, we compare the similarity of the Bloom filters. The similarity is measured by computing the Hamming distance between two Bloom filters. The less the distance is, the more similar the Bloom filters are. As the Bloom filter is a probabilistic data structure, the calculated distance measure will give an approximate value. We calculate the distance between the query Bloom filter and the most similar Bloom filter among a set of patients using the following formula to get an approximate value of edit distance:

$$d = |\mathcal{B}_1 \cup \mathcal{B}_2| - |\mathcal{B}_1 \cap \mathcal{B}_2| \tag{6.1}$$

Where $\mathcal{B}_1$ and $\mathcal{B}_2$ are two Bloom filters. Here, our approximate distance measure is the difference of the cardinality of the union Bloom filter and the cardinality of the intersected Bloom filter. We have used only one hash function for the cardinality estimation as Papapetrou *et. al* [44] determined that one is the optimal number of the hash function while estimating the cardinality using Bloom filter. Beck and Kerschbaum [43] also used only one hash function to calculate the cardinality estimation of Bloom filters.

Each edit operation from a VCF file is inserted into a Bloom filter. A single Bloom filter is used to insert all the single character edits of a single patient. Like the number of hash function $\mathcal{H}$, the length of the Bloom filter l is also fixed and we calculated this length the same way it was calculated by Beck and Kerschbaum [43], which is:

$$m = \frac{-1}{(1-p)^{1/n} - 1} \tag{6.2}$$

This equation is the simplified form of Equation 2.3 where the number of hash functions, $k = 1$. If $\mathcal{H}$, m, and k are identical in Bloom filters $\mathcal{B}_1$ and $\mathcal{B}_2$, we can perform set union and intersection operations between them in Equation 6.1 to calculate edit distance. The less the distance is, the more similar the Bloom filters are. $d = 0$ indicates that the Bloom filters $\mathcal{B}_1$ and $\mathcal{B}_2$ are identical.

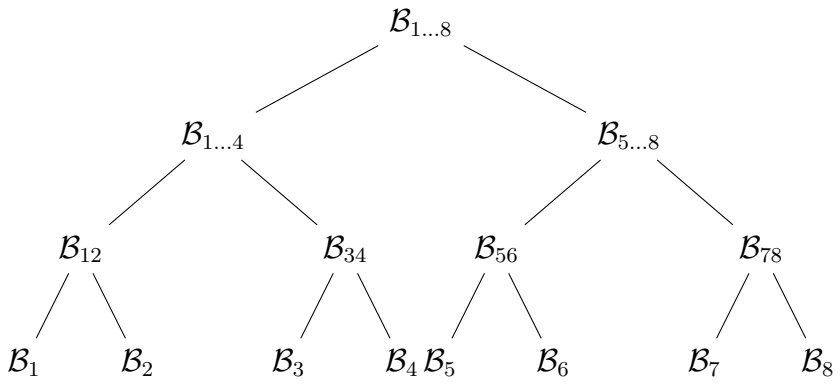


Figure 6.2: A sample BF-tree containing only 8 Bloom filters at child node. Each node except the root node has two children and is the union of its left and right child. Here, Bloom filter $\mathcal{B}_{12}$ is the union of Bloom filters $\mathcal{B}_1$ and $\mathcal{B}_2$. Similarly Bllom filter $\mathcal{B}_{1...4}$ denotes that it is the union of $\mathcal{B}_{12}$ and $\mathcal{B}_{34}$.

6.5.3 Construction of the BF-Tree

We use a BF-tree to identify similar patients from large datasets. A BF-tree resembles a binary tree that can be generated by repeated insertion of bloom filters as nodes containing genomic data. Every node of this tree contains a tuple $(\mathcal{B}, d, l, r)$, where $\mathcal{B}$ is the Bloom filter associated with that node, l and r are respectively the references of the left and right child of that node and if the associated node is the leaf node then d holds the reference to the corresponding database entry. So, the original record can be referenced from the leaf node.

Given a BF-tree T and a new sequence s, then s can be inserted into T by first computing a Bloom filter $\mathcal{B}(s)$ and then finding the appropriate node, n in the tree by traversing it starting from the root node. If n has no children, then a new node

Algorithm 5 Algorithm for constructing *BF-tree*

Input: Root node (*node*), Bloom filter $\mathcal{B}(s)$, and id

Output: BF-Tree, T

```
1: function buildTree(node, B(s), id)
2:     if node.l = null then
3:         newNode.bf ← B
4:         newNode.d ← id
5:         node.l = newNode
6:     else
7:         if d(node.l.B, B(s)) ≤ d(node.r.B, B(s)) then
8:             buildTree(node.l, B(s), id)
9:         else
10:            buildTree(node.r, B(s), id)
11:        end if
12:    end if
13:    return T
14: end function
```

containing $\mathcal{B}(s)$ is created as the left child of n. If n has only one child, then the new node representing $\mathcal{B}(s)$ is inserted as the right child of n. But, if n contains two children, then the Bloom filter to be inserted is compared against the Bloom filters of both left and right child of node n which are $n(l)$ and $n(r)$. The node with the more similar Bloom filter becomes the current node and this step is repeated. The similarity between two Bloom filters is measured by calculating the Hamming distance between them. The leaf node contains the Bloom filter and the reference to the corresponding database entry. The l and r references of the leaf node are set to null.

When a new node is inserted as the child node of a node n, then the Bloom filter of node n (parent node) is updated as the union of the Bloom filters of its children. So,

if a new node is added as the left child of n, then the Bloom filter of node n is updated as $l(\mathcal{B}) \cup r(\mathcal{B})$. Figure 6.2 shows an example of sample BF-tree. The procedure for building a BF-Tree is presented in Algorithm 5. The step of constructing the tree is independent of any queries and therefore can be constructed beforehand and used to execute multiple queries.

6.5.4 Encryption of the BF-Tree

After constructing this BF-tree T, the Bloom filters at each of the nodes of T are encrypted and then kept in a cloud server (CS) and available to the *researchers* to execute queries.

To encrypt the tree T, the CI chooses a random key k for a pseudorandom function (PRF) F. The Bloom filter $\mathcal{B}_i$ at node i is encrypted as:

$$\widetilde{\mathcal{B}}_i = \mathcal{B}_i \oplus F_k \tag{6.3}$$

This encrypted Bloom filter can be decrypted inside secure function evaluation (SFE) by garbled circuit (GC) during the search on the tree. After the encryption of all the nodes of T, CI sends the encrypted BF-Tree, $\widetilde{T}$ to CS.

6.5.5 Construction of the Query

Here we show how a query is constructed that is to be executed on the encrypted BF-tree. *Researcher* U has a genomic sequence q and he wants to find the patient with the most similar genomic sequence using the tree $\widetilde{T}$ resided in the CS. But, U

does not want to reveal the query or the result to *CS*. First, U will use the public reference genome, **Ref** to generate the edit sequences and then convert each multi-character edit into multiple single character edits the same way described in Section 6.5.1.

U and *CS* agree on the domain of each hash function (Σ) used in the Bloom filters. U inserts each of the single character edits into the query Bloom filter, $\mathcal{B}_q$. This Bloom filter will then be evaluated against the encrypted Bloom filters stored in the nodes of Tree $\widetilde{T}$ using SFE.

6.5.6 Search on Encrypted BF-Tree

Given a query Bloom filter $\mathcal{B}_q$ generated from the query sequence q and an encrypted BF-tree $\widetilde{T}$. The search process starts with the researchers transforming $\mathcal{B}_q$ into corresponding Boolean circuit, Q. Then the server and the client engage in a secure computation protocol to evaluate Q through which starting from the root node of $\widetilde{T}$ to find a similar node containing the Bloom filter which matches $\mathcal{B}_q$.

Let, l and r be the children of a node n of tree $\widetilde{T}$. The next node to be traversed is decided by calculating the Hamming distance between $\mathcal{B}_q$ and Bloom filters at node l and r which are $\widetilde{\mathcal{B}_l}$ and $\widetilde{\mathcal{B}_r}$ respectively. The child with more similar Bloom filter becomes the current node and its children are again compared with the query Bloom filter and this process continues until we reach the leaf node. Figure 6.3 shows how the tree is traversed during the query execution. Here q is the query Bloom filter. This Bloom filter will be compared with the left and right child at each node and the child with a Bloom filter with less distance is traversed next.

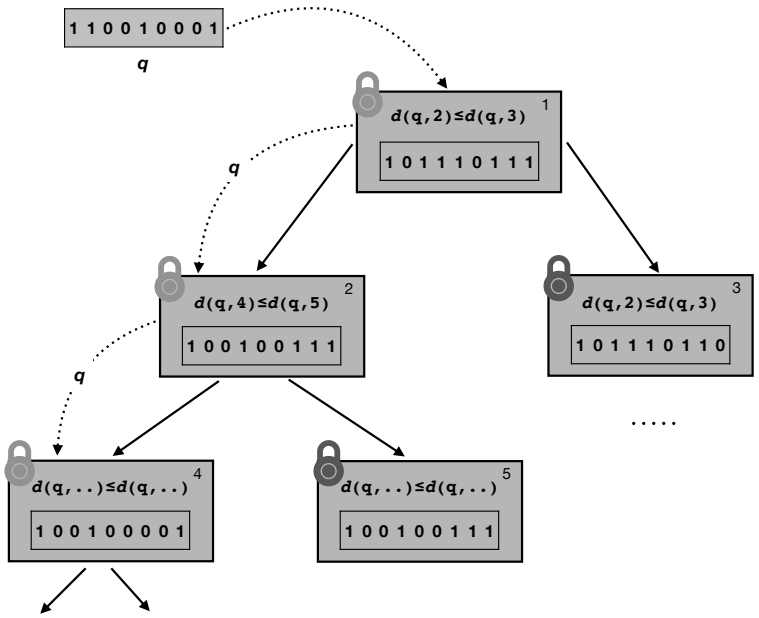


Figure 6.3: An example of traversing the tree. In each node, the Hamming distance between the query Bloom filter q and the Bloom filters at the child nodes are compared. The child node containing the more similar Bloom filter to q is traversed next.

The computation of Hamming distance and comparison of the values are done via very efficient secure function evaluation by a garbled circuit. Recall that, the Bloom filters at node i is encrypted using a PRF as specified in Equation 6.3. Before comparing and calculating the Hamming distance we have to decrypt the Bloom filter at node i. The client provides the key k for PRF to decrypt the encrypted Bloom filter inside the circuit. The client gets this key k from *CI*. The required circuit to decrypt the Bloom filter is :

$$\begin{aligned}&\mathbf{DEC}((b_1, b_2, \ldots, b_n), (r_1, r_2, \ldots, r_n))\\ &= b_i \oplus r_i \text{ for all } i \in n\end{aligned} \tag{6.4}$$

Here, $(b_1, b_2, \ldots, b_n)$ is from encrypted Bloom filter and $(r_1, r_2, \ldots, r_n)$ is from the PRF. In the circuit, the client will provide the PRF and the server will provide the encrypted Bloom filter. Both of the Bloom filters from the left child and right child will be decrypted in the same way. Let, $\widetilde{b}_n^1$ and $\widetilde{b}_n^2$ be the decrypted Bloom filters from the left and right child respectively. After decrypting the Bloom filter, the Hamming distance between the query Bloom filter and the decrypted Bloom filter is calculated, which is equivalent to xoring the decrypted Bloom filters from the left and right child and then calculating the number of ones from the xored value inside the circuit. The circuit to calculate the Hamming distance can be represented as:

$$\begin{aligned}&\mathbf{HD^j}((\widetilde{b}_1^j, \widetilde{b}_2^j, \ldots, \widetilde{b}_n^j),\\ &(q_1, q_2, \ldots, q_n)) = \sum_{i=1,j=1}^{i=n,j=2} (q_i \oplus \widetilde{b}_i^j)\end{aligned} \tag{6.5}$$

Here, $(\widetilde{b}_1^j, \widetilde{b}_2^j, \ldots, \widetilde{b}_n^j)$ are the unencrypted Bloom filters. $j = 1$ and $j = 2$ denotes the Bloom filters from the left and right child respectively and $(q_1, q_2, \ldots, q_n)$ is from the query Bloom filter. Let $\mathbf{HD^1}$ denotes the distance between the Bloom filter at the left child and the query Bloom filter and $\mathbf{HD^2}$ denote the distance between the Bloom filter at the right child and the query Bloom filter. The final circuit will compare the values of $\mathbf{HD^1}$ and $\mathbf{HD^2}$. If the value of $\mathbf{HD^1}$ is less than or equal to $\mathbf{HD^2}$, the output of the circuit will be "true", otherwise the output will be "false".

Algorithm 6 Algorithm for searching on a *BF tree*, T

Input: Root node of T (*node*), and query Bloom filter, $\mathcal{B}_q$

Output: This algorithm will return the *Edit Distance*

```
1: function searchTree(node, B_q)
2:     if node.l = null then
3:         result = d(B_q, B_n)                    ▷ The final edit distance
4:     else
5:         if d(B_q, B̃_l) ≤ d(B_q, B̃_r) then
6:             searchTree(node.l, B_q)
7:         else
8:             searchTree(node.r, B_q)
9:         end if
10:    end if
11:    return result
12: end function
```

If the output is true, then the search process proceeds to the left child. So, the left child becomes the parent node and the process of comparing the Hamming distance continues the same way described earlier.

So, the final comparison program produces output which child to be traversed, left or right:

```
if d(B_q, B̃_l) ≤ d(B_q, B̃_r) then
    traverse 'left'
else
    traverse 'right'
end if
```

When we reach the leaf node through traversing, it means that we have found the Bloom filter which represents the genomic sequence of the matched patient. The edit distance between the query Bloom filter and the Bloom filter at the leaf node can

Researcher		**Common**		***CS/CI***
Input:		**Input:**		**Input:**
Genomic sequence, s		Public reference genome, Ref		n number of genomic sequences
		Hash function, $\mathcal{H}$		
Offline:				**Offline(CI):**
Generate edits using Ref				For each sequence, S_i
Convert to single character edits				i. Generate edits using Ref
Insert edits into BF, $\mathcal{B}_q$				ii. Convert to single character edits
				iii. Insert edits into BF, $\mathcal{B}_i$
				iv. Insert $\mathcal{B}_i$ into tree, T
Online:				**Online(CS):**
				For each node n in T
				1. If n has two children then
Input to circuit $\mathcal{B}_q$	$\xrightarrow{\mathcal{B}_q}$		$\xleftarrow{\mathcal{B}_l,\ \mathcal{B}_r}$	a) Input to circuit $\mathcal{B}_l$ and $\mathcal{B}_r$
		$d(\mathcal{B}_q,\ \mathcal{B}_l) \overset{?}{\leq} d(\mathcal{B}_q,\ \mathcal{B}_r)$		
			$\xrightarrow{yes/no}$	
				b) If yes, traverse left child; else, traverse right child
				c) For the child node, repeat step 1
				2) Else, leaf node found

Figure 6.4: The overall protocol of our proposed solution. Both the *CS* and *CI* have two common inputs, i) the public reference genome, **Ref** and ii) the hash function of Bloom filter, $\mathcal{H}$. The *CI* generates the tree T offline. The *CS* also generates the query Bloom filter, $\mathcal{B}_q$ offline. Then the server and the client engage in a secure computation protocol to find the appropriate leaf node containing the similar patient in the tree, T.

also be calculated the same way described above. Algorithm 6 describes the process of searching on tree $\widetilde{T}$.

Figure 6.4 describes all the steps involved in our model, including the protocol for query execution.

6.5.7 Runtime Complexity.

The search algorithm of the BF-tree resembles a binary search algorithm. If A is the database containing the records, then the worst-case and average case runtime complexity is $\mathcal{O}(|A|)$ and $\mathcal{O}(\log\ |A|)$ respectively. Computation and comparison of Hamming distance between two Bloom filters in line 5 of Algorithm 6 has the complexity $\mathcal{O}(m)$. So, the overall average runtime complexity of our search algorithm described in Algorithm 6 is $\mathcal{O}(m \cdot \log\ |A|)$

6.6 Experimental Result

We have implemented the BF-Tree in Java programming language as a binary tree. Each patients information was represented in a VCF file. All single character edits generated from a VCF file were inserted into a single Bloom filter. Each leaf node in the BF-Tree contains a Bloom filter. All nodes in the tree except the leaf node contains pointers to the left and right child. The tree is constructed recursively and the search algorithm is also recursive.

Calculating the edit distance using Equation 6.1 is approximated by calculating the Hamming distance between the Bloom filters [43]. So, in our method, we calculated Hamming distance instead of performing union and intersection operations of the Bloom filters.

The configuration of the computer where the tree is built and hosted is described in Table 6.1. For secure function evaluation we have used the garbled circuit implementation of *FlexSC* [60] library which is written in java.

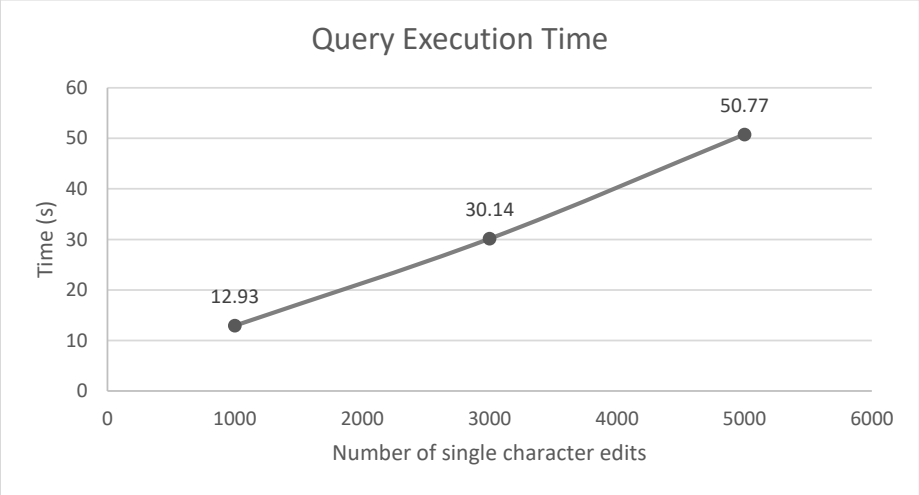


Figure 6.5: Average running time to execute a query. Note that, the reported time does not include the amount of time required to generate end encrypt the tree in the preprocessing stage.

	Cloud Server
Operating System	Ubuntu 16.04
Processor	Intel Core i5-4590
Memory	8 GB

Table 6.1: Configuration of the server.

We evaluated our system on a real life dataset from the Personal Genome Project (PGP) [65]. The dataset contains genomic sequences of 110 individuals. We ran three different experiments where we inserted 1000, 3000, and 5000 single-character edits into the Bloom filter. If we increase the length of the sequence, it will produce more single-character edits. To produce 5000 single-character edits, we had to consider the length of the sequence upto 15,102,400. The average time required to find a similar patient from tree $\widetilde{T}$ is shown in Figure 6.5. We can see that the running time increases linearly with the increase of single-character edits which means the run time increases

with the increase in the length of the sequence.

Table 6.2 lists the cost of using the garbled circuit for query execution for different number of single-character edits. It shows the amount of data transferred between the *researcher* and the *CS*, and the amount of time the garbled circuit runs in each end.

# of single character edits	Communication Cost (MB)	GC runtime (*researcher*)(s)	GC runtime (*CS*)(s)
1000	57.32	12.13	12.53
3000	112.2	28.94	29.55
5000	162.09	49.24	50.11

Table 6.2: The cost of garbled circuit per query.

6.7 Security Analysis

The security of the system is compromised if any part of the genomic sequence is revealed to the CS. On the other hand, researchers are only allowed to know the final result of their query. Following we discuss various leakages of our proposed model.

Leakage during the tree building and tree encryption phase. *CI* is only responsible for the generation and encryption of the *BF-Tree* and is considered as a trusted entity. So the leakage to the *CI* is none in this phase. As the *CS* and the *researchers* are not involved during this phase, the leakage to them is zero. The tree is encrypted using a non malleable encryption scheme. So, the *CS* cannot also infer any information from the encrypted *BF-Tree*, $\tilde{T}$ it receives from *CI*.

Leakage to *CI*. The *CI* is not involved at all during the query execution, it's only responsibility is to provide the secret key to the *researchers*. The leakage to *CI* during the query execution is none.

Leakage to *researchers*. The *researchers* only knows the result of the query. Note that, it is also possible to extract information about the data by observing the output which is known as inference attack [26]. One way to avoid this attack is to use *differential privacy* [61] which adds noise to the output. Preventing inference attack is completely the different domain of research and studied extensively in the literature.

Leakage to *CS*. The *CS* will be able to know the tree traversal pattern. It knows which nodes are being traversed from the root to the leaf while executing the query. As $\tilde{T}$ is encrypted and *CS* does not receive the query from the *researchers*, the *CS* does not reveal any information about the data or the query. The output to the circuit we used in our method is a boolean value (either "left" or "right"). So, the *CS* will know only which nodes are traversed during query execution.

6.8 Summary

- We have developed a method for similar patient matching which is able to preserve the data, query and output privacy.
- We have pre-filtered the searched result using a data structure called BF-Tree.
- Our experiment and analysis show that our method achieves reasonable accuracy to find similar sequence using the genomic data.

Chapter 7

Conclusion

In this thesis, we have presented three different algorithms for secure outsourcing of genomic data in a central repository (cloud server) and performed three different computations (i.e., count query, hamming distance, and edit distance).

7.1 Summary

Firstly, in Chapter 4, we have presented a secure and efficient method for outsourcing genomic data. The proposed method constructs an *index tree* from the aggregate genomic data and then outsources it to the third party cloud server. By employing a secure interactive protocol, the cloud server can traverse the nodes of the tree and execute count query operation. We have demonstrated that our model does not reveal any sensitive genomic data during the data processing as well as query execution phase. We have also taken into account the phenotype information using a data structure similar to Bloom filter search tree.

In the next chapter (Chapter 5), we propose another model for similar sequence search on encrypted data. We incorporate a prefix tree-based indexing technique in our proposed model which not only provides an effective storage solution for large genomic datasets but also facilitates efficient query execution by traversing the nodes of the tree. In this model, we have used Hamming distance to calculate the similarity measure.

Finally, in Chapter 6, we introduce another method for similar patient matching (SPM) operation. We generated a BF-tree using all the available data and all the computation for SPM is done on this tree. To calculate the similarity between two sequences we have used edit distance. Experimental results on real life dataset demonstrates the efficiency of our model.

7.2 Looking ahead

We have used garbled circuit to provide the security of the computation which require the interaction between the *cloud server* and the *researcher*. It would be interesting to see the performance of a non-interactive solution leveraging the recent development of *fully homomorphic encryption* schemes [66]. Recently, Intel has introduced Software Guard Extensions (Intel SGX) [67] architecture for secure computation. SGX provides a protected area in the memory called enclaves which protects the code and data during the computation. Though the previous research using cryptographic hardware [28] had some limitations, the performance of SGX is expected to be better. Designing a cloud-based database management system using the SGX is an interesting research challenge. Another promising area of improvement could

be the use of differential privacy techniques which provides the privacy of the output by adding noise to answers of the queries [61], thus preventing the inference attack.

Secure query execution on encrypted data is an active area of research. In the next few years, we expect that existing open problems will be addressed and new innovative technologies will be introduced to reduce the computational and communication overhead. However, the problems of genomic data privacy cannot be solved only by technology. There is an urgent need to bridge the gap between privacy technologies and current policies. We believe that cross-disciplinary research among computer science, biomedical and public policy community is needed to bring social and legal regulations that will complement the best practices of privacy-preserving technology.

Bibliography

[1] M. Akgün, A. O. Bayrak, B. Ozer, and M. Ş. Sağıroğlu, "Privacy preserving processing of genomic data: A survey," *Journal of Biomedical Informatics*, vol. 56, pp. 103–111, 2015. 1

[2] F. Yu, S. E. Fienberg, A. B. Slavković, and C. Uhler, "Scalable privacy-preserving data sharing methodology for genome-wide association studies," *Journal of biomedical informatics*, vol. 50, pp. 133–141, 2014. 1, 20

[3] P. R. Burton *et al.*, "Size matters: just how big is big?: Quantifying realistic sample size requirements for human genome epidemiology," *International Journal of Epidemiology*, vol. 38, no. 1, pp. 263–273, 2009. 2

[4] M. Naveed *et al.*, "Privacy in the genomic era," *ACM Computing Surveys (CSUR)*, vol. 48, no. 1, 2015. 2, 26

[5] J. Chen, W. Geyer, C. Dugan, M. Muller, and I. Guy, "Make new friends, but keep the old: recommending people on social networking sites," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2009, pp. 201–210. 3

[6] M. Gymrek, A. L. McGuire, D. Golan, E. Halperin, and Y. Erlich, "Identifying personal genomes by surname inference," *Science*, vol. 339, no. 6117, pp. 321–324, 2013. 4

[7] X. Zhou, B. Peng, Y. F. Li, Y. Chen, H. Tang, and X. Wang, "To release or not to release: Evaluating information leaks in aggregate human-genome data," in *Proceedings of the 16th European Conference on Research in Computer Security*. Springer-Verlag, 2011, pp. 607–627. 4

[8] B. Malin and L. Sweeney, "How (not) to protect genomic data privacy in a distributed network: Using trail re-identification to evaluate and design anonymity protection systems," *Journal of Biomedical Informatics*, vol. 37, no. 3, pp. 179–192, 2004. 4

[9] Y. Erlich *et al.*, "Redefining genomic privacy: Trust and empowerment," *PLoS Biol*, vol. 12, 11 2014. 4

[10] A. Yao, "How to generate and exchange secrets," in *Foundations of Computer Science, 1986., 27th Annual Symposium on*. IEEE, 1986, pp. 162–167. 4, 9, 11, 30

[11] Z. Hasan, M. S. R. Mahdi, and N. Mohammed, "Secure count query on encrypted genomic data: A survey," *IEEE Internet Computing, to appear*. 6

[12] M. Z. Hasan, M. S. R. Mahdi, and N. Mohammed, "Secure count query on encrypted genomic data," in *Proceedings 3rd International Workshop on Genome*

Privacy and Security (GenoPri) in conjunction with the AMIA Annual Symposium, 2016. 6

[13] Z. Hasan, M. S. R. Mahdi, M. N. Sadat, and N. Mohammed, "Secure count query on encrypted genomic data," *Journal of Biomedical Informatics (JBI), under minor review*. 6

[14] M. S. R. Mahdi, M. Z. Hasan, and N. Mohammed, "Secure sequence similarity search on encrypted genomic data," in *IEEE/ACM International Conference on Connected Health: Applications, Systems and Engineering Technologies (CHASE)*, 2017, pp. 205–213. 6

[15] Z. Hasan and N. Mohammed, "Identification of similar patients with edit distance approximation," *Submitted*. 7

[16] M. O. Rabin, "How To Exchange Secrets with Oblivious Transfer." *IACR Cryptology ePrint Archive*, vol. 2005, p. 187, 2005. 10, 11

[17] S. Even, O. Goldreich, and A. Lempel, "A randomized protocol for signing contracts," *Communications of the ACM*, vol. 28, no. 6, pp. 637–647, 1985. 11

[18] C. Hazay and Y. Lindell, *Efficient secure two-party protocols: Techniques and constructions*. Springer Science & Business Media, 2010. 12, 13, 38

[19] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Proceedings of the 17th International Conference on Theory and Application of Cryptographic Techniques*, 1999, pp. 223–238. 14, 15, 44, 54, 58

[20] C. Paar and J. Pelzl, *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009. 15

[21] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970. 16

[22] J. C. Venter *et al.*, "The sequence of the human genome," *Science*, vol. 291, no. 5507, pp. 1304–1351, 2001. 19

[23] Z. Lin, A. B. Owen, and R. B. Altman, "Genomic research and human subject privacy," *Science*, vol. 305, no. 5681, pp. 183–183, 2004. 20

[24] N. Homer, S. Szelinger, M. Redman, D. Duggan, W. Tembe, J. Muehling, J. V. Pearson, D. A. Stephan, S. F. Nelson, and D. W. Craig, "Resolving individuals contributing trace amounts of dna to highly complex mixtures using high-density snp genotyping microarrays," *PLoS Genet*, vol. 4, no. 8, pp. 1–9, 2008. 20

[25] R. Wang, Y. F. Li, X. Wang, H. Tang, and X. Zhou, "Learning your identity and disease from research papers: Information leaks in genome wide association study," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*. ACM, 2009, pp. 534–544. 20

[26] N. A. Erlich, Yaniv, "Routes for breaching and protecting genetic privacy," *Nat Rev Genet*, vol. 15, no. 6, pp. 409–421, 2014. 20, 100

[27] M. Kantarcioglu, W. Jiang, Y. Liu, and B. Malin, "A cryptographic approach to securely share and query genomic sequences," *IEEE Transactions on information technology in biomedicine*, vol. 12, no. 5, pp. 606–617, 2008. 23, 25, 33, 56, 57

[28] M. Canim, M. Kantarcioglu, and B. Malin, "Secure management of biomedical data with cryptographic hardware," *IEEE Transactions on Information Technology in Biomedicine*, vol. 16, no. 1, pp. 166–175, 2012. 25, 33, 35, 56, 57, 102

[29] R. Agrawal, D. Asonov, M. Kantarcioglu, and Y. Li, "Sovereign joins," in *22nd International Conference on Data Engineering (ICDE'06)*, April 2006, pp. 26–26. 26

[30] K. Lauter, A. López-Alt, and M. Naehrig, "Private computation on encrypted genomic data," in *International Conference on Cryptology and Information Security in Latin America*. Springer, 2014, pp. 3–27. 26, 32

[31] L. Kamm, D. Bogdanov, S. Laur, and J. Vilo, "A new way to protect privacy in large-scale genome-wide association studies," *Bioinformatics*, vol. 29, no. 7, pp. 886–893, 2013. 27, 28

[32] F. Chen, S. Wang, X. Jiang, S. Ding, Y. Lu, J. Kim, S. C. Sahinalp, C. Shimizu, J. C. Burns, V. J. Wright *et al.*, "PRINCESS: Privacy-protecting rare disease international network collaboration via encryption through software guard extensions," *Bioinformatics*, 2017. 27, 32

[33] C. C. Khor *et al.*, "Genome-wide association study identifies FCGR2A as a susceptibility locus for kawasaki disease," *Nat Genet*, vol. 43, no. 12, pp. 1241–1246, Dec 2011. 27

[34] H. Zhicong, A. Erman, F. Jacques, H. Jean-Pierre, and J. Ari, "Genoguard:

Protecting genomic data against brute-force attacks," in *2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 447–462. 27

[35] Y. Zhang, W. Dai, X. Jiang, H. Xiong, and S. Wang, "FORESEE: Fully Outsourced secuRe gEnome Study basEd on homomorphic Encryption," *BMC medical informatics and decision making*, vol. 15, no. 5, pp. 1–11, 2015. 27

[36] G. Choi, J. L. Raisaro, S. Pradervand, R. Colsenet, N. Jacquemont, N. Rosat, and J.-P. Hubaux, "Privacy-preserving exploration of genetic cohorts with i2b2 at lausanne university hospital," in *Proceedings of the 3rd International Workshop on Genome Privacy and Security (GenoPri'16)*, 2016. 27

[37] W. Xie *et al.*, "SecureMA: Protecting participant privacy in genetic association meta-analysis," *Bioinformatics*, pp. 3334–3341, 2014. 28

[38] Y. Zhang, M. Blanton, and G. Almashaqbeh, "Secure distributed genome analysis for GWAS and sequence comparison computation," *BMC Medical Informatics and Decision Making*, 2015. 28, 29

[39] S. Wang *et al.*, "HEALER: Homomorphic computation of ExAct Logistic rEgRession for secure rare disease variants analysis in GWAS," *Bioinformatics*, vol. 32, no. 2, pp. 211–218, 2016. 28

[40] T. Dugan and X. Zou, "A Survey of Secure Multiparty Computation Protocols for Privacy Preserving Genetic Tests," in *Connected Health: Applications, Systems and Engineering Technologies (CHASE), 2016 IEEE First International Conference on*. IEEE, 2016, pp. 173–182. 28

[41] J. H. Cheon, M. Kim, and K. Lauter, "Homomorphic computation of edit distance," in *Financial Cryptography and Data Security*. Springer Berlin Heidelberg, 2015, pp. 194–212. 28

[42] R. A. Wagner and M. J. Fischer, "The String-to-String Correction Problem," *Journal of the ACM (JACM)*, vol. 21, no. 1, pp. 168–173, Jan. 1974. 28

[43] M. Beck and F. Kerschbaum, "Approximate two-party privacy-preserving string matching with linear complexity," in *2013 IEEE International Congress on Big Data*, June 2013, pp. 31–37. 29, 88, 97

[44] O. Papapetrou, W. Siberski, and W. Nejdl, "Cardinality estimation and dynamic length adaptation for bloom filters," *Distrib. Parallel Databases*, vol. 28, no. 2-3, pp. 119–156, Dec. 2010. 29, 88

[45] X. S. Wang, Y. Huang, Y. Zhao, H. Tang, X. Wang, and D. Bu, "Efficient genome-wide, privacy-preserving similar patient query based on private edit distance," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 492–503. 29, 85, 86

[46] H. Perl, Y. Mohammed, M. Brenner, and M. Smith, "Fast confidential search for bio-medical data using bloom filters and homomorphic cryptography," in *E-Science (e-Science), 2012 IEEE 8th International Conference on*, 2012, pp. 1–8. 30

[47] S. Jha, L. Kruger, and V. Shmatikov, "Towards practical privacy for genomic

computation," in *2008 IEEE Symposium on Security and Privacy (SP 2008)*, 2008, pp. 216–230. 30

[48] T. Smith and M. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195 – 197, 1981. 30

[49] A. Rheinländer, M. Knobloch, N. Hochmuth, and U. Leser, "Prefix tree indexing for similarity search and similarity joins on genomic data," in *Scientific and Statistical Database Management*. Springer, 2010, pp. 519–536. 30

[50] C. Wang, K. Ren, S. Yu, and K. M. R. Urs, "Achieving usable and privacy-assured similarity search over outsourced cloud data," in *INFOCOM, 2012 Proceedings IEEE*. IEEE, 2012, pp. 451–459. 31

[51] E. Ayday, J. L. Raisaro, J.-P. Hubaux, and J. Rougemont, "Protecting and Evaluating Genomic Privacy in Medical Tests and Personalized Medicine," in *Proceedings of the 12th ACM Workshop on Workshop on Privacy in the Electronic Society*. ACM, 2013, pp. 95–106. 31

[52] J.-J. Yang, J.-Q. Li, and Y. Niu, "A hybrid solution for privacy preserving medical data sharing in the cloud environment," *Future Generation Computer Systems*, vol. 43, pp. 74–86, 2015. 31

[53] S. Purcell *et al.*, "PLINK: a tool set for whole-genome association and population-based linkage analyses," *The American Journal of Human Genetics*, vol. 81, no. 3, pp. 559–575, 2007. 35

[54] D. N. Paltoo *et al.*, "Data use under the NIH GWAS Data Sharing Policy and future directions," *Nature genetics*, vol. 46, no. 9, pp. 934–938, 2014. 36, 38

[55] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft, "Privacy-preserving ridge regression on hundreds of millions of records," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 334–348. 47

[56] V. Pappas *et al.*, "Blind seer: A scalable private dbms," in *Security and Privacy (SP), 2014 IEEE Symposium on*, May 2014, pp. 359–374. 49

[57] Y. Huang, D. Evans, J. Katz, and L. Malka, "Faster secure two-party computation using garbled circuits." in *USENIX Security Symposium*, 2011. 54

[58] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, "Oblivm: A programming framework for secure computation," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 359–376. 54

[59] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway, "Efficient garbling from a fixed-key blockcipher," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 478–492. 54

[60] X. Wang, H. Chan, and E. Shi, "Circuit oram: On tightness of the goldreich-ostrovsky lower bound," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 850–861. 54, 76, 97

[61] C. Dwork, A. Roth *et al.*, "The algorithmic foundations of differential privacy,"

Foundations and Trends in Theoretical Computer Science, vol. 9, no. 3-4, pp. 211–407, 2014. 59, 79, 100, 103

[62] N. V. Chawla and D. A. Davis, "Bringing big data to personalized healthcare: a patient-centered framework," *Journal of general internal medicine*, vol. 28, no. 3, pp. 660–665, 2013. 61

[63] D. Vatsalan and P. Christen, "Privacy-preserving matching of similar patients," *J. of Biomedical Informatics*, vol. 59, no. C, pp. 285–298, Feb. 2016. 62

[64] "Idash-privacy and security workshop on genomic data." http://www.humangenomeprivacy.org/2015/competition-tasks.html, 2015. 76

[65] G. M. Church, "The personal genome project," *Molecular Systems Biology*, vol. 1, no. 1, 2005. 98

[66] C. Gentry, *A fully homomorphic encryption scheme*. Stanford University, 2009. 102

[67] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, "Using innovative instructions to create trustworthy software solutions," in *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '13. ACM, 2013, pp. 11:1–11:1. 102