Privacy-preserving Biomedical Data Sharing and Computation

by

Md Safiur Rahman Mahdi

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

Doctor of Philosophy

Department of Computer Science
The University of Manitoba
Winnipeg, Manitoba, Canada
August 2020

Thesis advisor

Noman Mohammed

Author

Md Safiur Rahman Mahdi

Privacy-preserving Biomedical Data Sharing and Computation

Abstract

Genomic data is being produced rapidly by both individuals and enterprises and needs to be outsourced from local machines to a cloud for better flexibility. Outsourcing also eliminates the local storage management problem for data owners. However, sensitive data must be encrypted by data owners before outsourcing in the cloud to protect data privacy and security. Since genomic data is huge in volume, it is challenging to execute researchers' queries securely and efficiently. In this thesis, I have developed various models for secure sharing and computation on genomic data in a third party cloud server. The security of the shared data is guaranteed through encryption while making the overall computation fast and scalable enough for real-life biomedical applications. In particular, I propose different methods for secure sharing and computation on genomic data such as secure count query, secure similar patients query, secure substring, and set-maximal search.

Acknowledgments

First of all, I would like to thank my supervisor, Dr. Noman Mohammed, for his help and advice for the entire duration of my Ph.D. program. I would not be able to complete my thesis without his continuous guidance and directions. I would also like to acknowledge the financial support I received from Dr. Mohammed. I also extend my thanks to Dr. Carson Kai-Sang Leung and Dr. Carl Ho for being my thesis committee members. I appreciate the time they have spent reading my thesis and thoughtful suggestions.

Secondly, I would like to thank my collaborators Zahidul Hasan, Md Momin Al Aziz, Nazmus Sadat, Dima Alhadidi, and Xiaoqian Jiang for their valuable times and efforts in my thesis through valuable suggestions and feedback.

Thirdly, I like to thank fellow lab-mates in the Data Security and Privacy lab, in the form of friendship and encouragement: Tasnia Faequa, Kazi Wasif Ahmed, Md Waliullah, Reza Ghesemi, Toufique Morshed, and Tanbir Ahmed. I also would like to thank the faculty and staff members of the Department of Computer Science who have helped me in various ways.

Last but not the least, I would like to acknowledge the unconditional love and countless support from my parents and siblings. I also like to thank my wife, Farhana Islam, for her continuous support and understanding during the difficult times. To my children, Arhaan and Arya, thank you for being the cause of joy and happiness in my life. I would also like to thank all of my friends who have made my stay in Winnipeg very pleasant.

This thesis is dedicated to my parents and my lovely wife.

Copyright Notices and Disclaimers

Sections of this thesis have been published in conference proceedings and journal publications, either previously or forthcoming at the time of publication.

The Majority of Chapter 4

- **MSR. Mahdi**, MN. Sadat, N. Mohammed, and X. Jiang – Secure Count Query on Encrypted Heterogeneous data. In *Proceedings of the 18th IEEE International Conference on Dependable, Autonomic and Secure Computing*, Calgary, Canada, 2020.

- Z. Hasan, **MSR. Mahdi**, MN. Sadat, and N. Mohammed – Secure Count Query on Encrypted Genomic Data. In *Journal of Biomedical Informatics (JBI)*, vol. 81, pp. 41-52, 2018.

The Majority of Chapter 5

- **MSR. Mahdi**, Z. Hasan, and N. Mohammed – Secure Sequence Similarity Search on Encrypted Genomic Data. In *Proceedings of the IEEE/ACM International Conference on Connected Health: Applications, Systems and Engineering Technologies (CHASE)*, Philadelphia, PA, USA, pp. 205-213, 2017.

The Majority of Chapter 6

- **MSR. Mahdi**, MMA. Aziz, D. Alhadidi, and N. Mohammed – Secure Similar Patients Query on Encrypted Genomic Data. In *IEEE Journal of Biomedical and Health Informatics (JBHI)*, vol. 23, pp. 2611-2618, 2019.

The Majority of Chapter 7

- **MSR. Mahdi**, MMA. Aziz, X. Jiang, and N. Mohammed – Privacy-Preserving String Search on Encrypted Genomic Data using a Generalized Suffix Tree. [Journal paper submitted].

Authors' Contributions

MSR. Mahdi formulated the problems, designed the solutions, and wrote the articles. Z. Hasan and MMA. Aziz assisted in analyzing and implementing sub-modules. MN. Sadat, D. Alhadidi, X. Jiang, and N. Mohammed provided feedback on the proposed solutions and edited the initial draft of the articles.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The volume of biomedical data is growing enormously due to the advancement of the current sequencing technologies. Simultaneously, it creates the opportunity to generate many healthcare applications such as personalized medicine, genetic compatibility testing, paternity testing, etc. However, human genome research may expose the private information of an individual, like susceptibility to a particular disease such as Alzheimer, breast cancer, or diabetes [4]. We can assess the susceptibility by querying an individual's genome across the common variations and then computing the predisposition [4]. This kind of analysis needs large number of genomic sequences to achieve better accuracy that sometimes a single organization cannot afford [5]. One possible solution can be granting access of the genomic data to outside of the organization. But releasing the data to a government organization or research institution, can cause significant risks of data privacy.

To store the shared data, cloud computing services are generally utilized because of cloud infrastructure's affordable low rate. However cloud services are unsafe and

may cause data security breaches. Publications clearly demonstrates that privacy should not be relied upon to be saved by cloud service providers [6]. Therefore, storing the huge biomedical data to the cloud could be a feasible solution if we can ensure data security and privacy.

1.1 Motivation

To release the data among multiple government organizations or research institutions, various privacy policies have been developed. Since every individual has a unique genome sequence, exposure of this sequence has major privacy risks. For example, an insurance company may deny the health coverage of a person who carries a particular gene that can lead to a specific disease. Thus the analysis procedure and storage of the genomic sequences should be done in a privacy-preserving manner.

My objective is to solve the security issues related to sharing and computation on outsourced genomic data. Overall, I address three potential security challenges. The first challenge is to provide ***data privacy***. The data stored in the cloud server, as well as the computation throughout the entire analysis process should be secured. Even if the cloud server gets compromised, the attacker should learn nothing about the data stored in the cloud. The second challenge is to achieve ***query privacy***. The institutions contributing the data, the cloud service provider or an adversary should not learn anything about a query executed by a researcher or an institution. The third challenge is to guarantee ***output privacy***. The result of the query will be disclosed to only the researcher who initiated the query.

Anonymization methods are inapplicable for protecting the genomic data [7] as

the generalization and suppression in these methods could change the meaning of the actual sequence. Cryptographic techniques can compute a predefined function on an encrypted dataset from multiple parties and return the function's result without exposing any information about the data from different parties [8]. For this reason, a number of privacy preserving techniques using cryptography have been developed to achieve the goal of sharing and computation on encrypted genomic data. However none of these techniques can overcome the three challenges mentioned above simultaneously or are feasible for practical implementation. In this thesis, I propose various methods to solve the problem of sharing and computing biomedical data by addressing all of the three challenges mentioned above.

1.2 Research Objectives

The primary research objectives of this thesis are as follows:

- Build a secure framework to outsource large volume of biomedical data to a untrusted cloud server.
- Perform secure computation on the outsourced data with better query execution time.
- Provide data privacy, query privacy, and output privacy. Note that I do not consider any privacy leakage through the output. Such inference attack can be abstained utilizing *differential privacy* and has been studied widely in the literature [9].

1.3 Contributions

I use a tree-based indexing to pre-filter the search result in all the proposed models that improves the performance significantly. My indexing techniques also provide an effective storage solution for large biomedical datasets. In addition, for the addition or removal of records, we do not need to regenerate the tree. New records can easily be added, deleted or modified to or from the nodes of the tree.

My proposed models also provide the security guarantee of: *data privacy, query privacy, and output privacy.* I execute the queries in my models by traversing the nodes of the tree. I use secure function evaluation (SFE) to take the decision which node to traverse. For SFE, I use Yao's garbled circuits [10]. Moreover, my 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, I demonstrate the effectiveness of my approaches in comparison with the previous approaches.

The contributions of this research are as follows:

1.3.1 Secure Count Query

I propose a secure method for performing count query operation on the encrypted data. Biomedical data is at first encrypted and then outsourced to a third party cloud server. Query execution is performed by traversing the encrypted tree, called *index tree* where the choice of navigating each node is made by checking whether a query predicate matches with a specific branch of the tree. Contingent upon the query, branches of the tree are traversed to figure the outcome from the data stored in the

nodes which match the query predicate. I propose two methods to perform secure count query using different datasets. The proposed methods for secure count query are as follows:

Secure Count Query on Encrypted Genotype Data

In my first method, I propose a method which can handle datasets containing genotype data only. This dataset contains only the genotypes and patients are divided into a case-control group based on with or without cancer.

Secure Count Query on Encrypted Heterogeneous Data

In the second method, I extend the previous work by integrating phenotype and numeric data in addition to genotype. In real-world applications, we see examples of this kind of dataset. Some clinical datasets contain the patients' genomic sequences, clinical profiles, and numeric data such as age or blood sugar level.

1.3.2 Secure Similar Patients Query

Similar Patients Query (SPQ) [1] is used to recognize similar patients from a large dataset of medical patients. The similarity is measured based on a patient's genomic sequence. By SPQ results, doctors can diagnose a patient better predicting the probability of having certain diseases. However, SPQ raises various security and privacy concerns. DNA sequences include health information about patients and their families. The exposure of such genomic sequences could harm patients by affecting education or employment opportunities. I present a secure method for determining

SPQ in a database of genomic sequences by proposing a prefix tree based indexing algorithm to pre-filter the search result. I utilize the following two different solutions to achieve SPQ:

Secure Hamming Distance

As the metric of the similarity measure of SPQ, I use hamming distance. 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 Top-k Query search using Edit Distance Approximation

I provide an alternative secure method to find similar patients by solving top-k query using edit distance approximation. I depend on the hamming distance to approximate the edit distance. Given a reference query sequence, my objective is to securely execute a SPQ. In particular, here SPQ retrieves the top-k sequence(s) from the aggregated databases based on the edit distance approximation.

1.3.3 Secure Substring and Set-maximal Search

Substring search and set-maximal search on genomic data is an indispensable problem. There are many real life applications of substring and set-maimal search such as ancestor search [11], identify similar patient, create personalized medicine etc. Besides genomic data, substring search and set-maximal search are also important for pattern matching or word searching that applicable to search any keyword in text

message, chat logs, documents etc. I utilized a *generalized suffix tree* to create an index of the genomic data, and secure query execution is ensured via the garbled circuit.

1.4 Thesis Organization

This thesis is organized as follows:

- **Chapter 2** provides the necessary biological and cryptographic background to understand the techniques I have adopted in my frameworks to provide the security of the biomedical data.
- **Chapter 3** presents a brief discussion on the related literature.
- **Chapter 4** addresses the secure count query operation on the encrypted data. The initial results of this chapter appear in [12]. The final results of this chapter is going to be appeared in [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 query using top-k query and edit distance approximation. The results of this chapter appear in [15].
- **Chapter 7** presents a method to privacy-preserving substring search and set-maximal search on encrypted genomic data using a generalized suffix tree. The results of this chapter have been submitted in a peer-reviewed journal [16].

- **Chapter** 8 concludes the thesis providing some future research directions.

Chapter 2

Background

In this section, I discuss biology background, various cryptographic techniques, and security requirement relevant to my thesis.

2.1 Biology Background

This section discusses various relevant biological background information and provides how security is involved while handling biological data.

2.1.1 Biomedical Data

Biomedical data represents the bio-medicine data or data related to biology and medicine [17]. All living organisms contain genome as hereditary material. The fundamental unit of genome is known as *deoxyribonucleic acid (DNA)* molecules. DNA is a linear molecule consisting of a sequence of four nucleotides or bases: adenine (A), cytosine (C), guanine (G) and thymine (T). In DNA, these nucleotides form base

pairs (bp) by making bonds with each other: A pairs with T and C pairs with G. DNA is double stranded and forms a helix structure. This structure of DNA was first described by James Watson and Francis Crick [18].

Genes are the portions of DNA coding sequences that hold the physical and functional unit of heredity [19]. Genes can span a few hundred DNA bases to more than 2 million bases [20]. The complete set of an organism's DNA is called the genome. In general, genes occupy a very small portion of the genome. For example, the human genome contains 30000 genes, which is only 1.5% of the entire genome. The rest of the genome consists of the repeated sequences and DNA that regulates genes [21].

The DNA of two different individuals are almost identical ($\sim$ 99%). In fact, Venter *et al.* [21] showed that DNA of two individuals differ no more than by 0.5%. This small amount of variations distinguish among different humans. Several types of genetic variations occur in human population, such as Copy-Number Variations (CNVs), Single Nucleotide Polymorphism (SNP), rearrangement etc [22].

Single Nucleotide Polymorphism

Single Nucleotide Polymorphism (SNP) is the most common form of DNA variation at a certain location in the genome. Each SNP represents a distinct single DNA building block, called a nucleotide. For instance, a SNP may supplant the nucleotide cytosine (C) with the nucleotide thymine (T) in a specific stretch of DNA [23]. SNPs occur regularly all through an individual's DNA. On average, they occur once in each 1,000 nucleotides, which implies there are approximately 4 to 5 million SNPs in an individual's genome. These varieties might be remarkable or happen in numerous

people; researchers have discovered more than 100 million SNPs in person's around the globe. In general, these varieties are found in the DNA between genes. They can perform as natural markers, helping researchers find genes that are related with disease [24].

Most of the SNPs do not have any effect on human health. Even though a specific SNP may not cause a disorder, a few SNPs are related with specific diseases. These affiliations permit researchers to search for SNPs to assess a person's hereditary inclination to build up a disease. Furthermore, if specific SNPs are known to be related with a trait, the researchers may look at stretches of DNA near these SNPs to recognize the gene or genes liable for the trait. Moreover, SNPs can be utilized to follow the inheritance of disease genes inside families [25].

Allele

Allele is a variation of a gene. A few genes have different structures, which are situated at a similar position on a chromosome. Human are called diploid creatures since they have two alleles at each hereditary locus (position), where one allele acquired from each parent. Each pair of alleles exhibits the genotype of a particular gene. Genotypes are depicted as homozygous if there are two indistinguishable alleles at a specific locus and as heterozygous if the two alleles vary. This is also called Alleles provide the creature's phenotype, which is the noticeable appearance of the organism.

Alleles can be either predominant or passive. In heterozygous organism, if a particular locus carries one predominant and one passive allele, the organism will

express the predominant phenotype [26]. If both allels are expressed, then it is called *bi-allelic*. The most common allele for a given SNP is known as *major allele* whereas the less common alele for a given SNP is called the *minor allele* [27].

Genotype

Genotype refers an organism's full set of genes. In other words, the term can be utilized to indicate the alleles, or variation types of a gene carried by an organism. Humans are diploid living beings, which implies that they have two alleles at each hereditary position, with one allele acquired from each parent. Each pair of alleles refers to the *genotype* of a particular gene [28].

Phenotype

Phenotype refers to the recognizable physical properties of a creature; these incorporate the living being's appearance, development, and behavior. A creature's phenotype is controlled by its genotype, as well as by environmental impact upon these genes. Because of the impact of natural components, creatures with indistinguishable genotypes, for example, indistinguishable twins, express nonidentical phenotypes. Instances of phenotypes include hair color, height, wing length etc [29].

2.1.2 Biomedical Data Security

Since biomedical data is broadly utilized in various applications, its security issues are surely known from their use cases. It is well established that change in the genomic sequence of an individual occurs since mutations can affect their health.

Some mutations can affect immediately whereas some affect at some point in future. Moreover, 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 [30].

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 [31].

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 [32]. 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 [33]. The security issue is generated largely from these collection and storage of genomic data for these purposes [34].

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 [35]. Ensuring the security of the SNPs in the GWAS is very important which has been clearly demonstrated by the work of Lin *et al.* [36] 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 [37; 38; 39].

2.2 Cryptographic Background

I utilize up-to-date cryptographic techniques to provide the security of my proposed technique. Therefore, I describe various useful cryptographic background necessary to understand those techniques.

2.2.1 Adversary

There are the following two main types of adversaries:

Semi-honest Adversary

In a semi-honest adversarial model, the parties accurately follow the protocol specification and do not wish to behave maliciously to generate the incorrect result. nevertheless, 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 [40; 41].

Malicious Adversary

In a malicious adversarial model, the corrupted parties 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. In general, 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 [42; 43].

2.2.2 Homomorphic Encryption

Fully homomorphic encryption (FHE) allows anybody to run any function over encrypted data, without first decrypting the data and any knowledge of secret key. A key generation algorithm generates a pair of keys: a public key, pk and a secret key, sk. Anyone can encrypt a value using this public key and anybody with the possession of this public key can perform computation over the encrypted data. The result of the computation is given in encrypted form and can only be decrypted by the person with the possession of the secret key. This scheme was first proposed by Craig Gentry [44].

A variation of FHE scheme is called partially homomorphic encryption (PHE). The main difference between PHE and FHE is that PHE only allows to perform specific functions to be computed over encrypted data. For example, Pailler cryptosystem [45] only allows addition and El Gamal [46] cryptosystem can compute multiplication. Both FHE and PHE provide strong security guarantee called semantic security. But computing arbitrary functions using fully homomorphic encryption are too slow to be practical [47].

FHE and PHE schemes are very popular and recorded its implementation extensively in literature. Cheon *et al.* [48] adopted somewhat homomorphic encryption scheme to compute edit distance. This scheme supports limited number of multiplications

besides addition. They adopted the edit distance algorithm suggested by Wagner *et al.* [49] and optimized it. It uses three different types of circuits for computing the edit distance: equality circuit, comparison circuit and addition circuits. For two encrypted DNA sequences of length 50, their algorithm would take one day to compute the result.

Lauter *et al.* [50] proposed a method where all data are encrypted using a single key and then hosted in a single cloud server. They used homomorphic encryption for calculating various statistical algorithms (e.g. Chi-Squared Test or Pearson Goodness-of-Fit, Linkage Disequilibrium [51], Estimation Maximization (EM) and Cochran-Armitage Test for Trend (CATT)) commonly used in genomic studies. They were successful in computing some polynomial functions using homomorphic encryption.

Gentry proposed a *bootstrappable* encryption system based on *ideal lattices* to lessen the noise after some calculation. This way calculation on ciphertext is possible but the bootstrapping process is computationally costly. Despite the fact that this finding was brought some light for homomorphic encryption but its utility was much questioned afterwards [52; 53]. Like the majority of the other *Randomized Encryption Protocol*, this will also expand the data size and the computational cost (because of bootstrapping). So here the research was divided in mostly in two parts:

- Improving the bootstrapping method based on lattices [54; 55].
- Scheme without bootstrapping which is named *Somewhat, Partial or Leveled* Homomorphic Encryption(SWHE or LHE) [56; 57].

Available Implementation

There are two mostly used implementations available for fully homomorphic encryption:

- Simple Encrypted Arithmetic Library *(SEAL) by Microsoft* [58; 59].
- Homomorphic Encryption software library *HElib* [60; 61].

2.2.3 Paillier Cryptosystem

Paillier cryptosystem is one of the outstanding homomorphic cryptosystem [45]. Homomorphic encryption permits to execute calculation on the encrypted data without decrypting it. Also, if we decrypt the result, it would be the equivalent if we perform the calculation on the plaintext. *Paillier cryptosystem* [45] supports addition and an adversary with the limited computational power and with the ownership of the ciphertext would not be able to extract any information about the plaintext. This is known as semantically secure. To ensure this security, this cryptosystem produces diverse ciphertexts when an equivalent message is encrypted on numerous times. This irregularity implies that this cryptosystem is a probabilistic encryption scheme.

I utilize *Paillier Cryptosystem* [45] to encrypt the data and use its homomorphic properties to perform count query. In the *Paillier Cryptosystem* [45], a key generation algorithm generates a pair of keys: a public key, pk and a secret key, sk. The secret and the public key are used for decryption and encryption purposes respectively. Therefore, after the encryption of a message m if we achieve 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, $\xi_{pk}(m)$ represents encryption of message m using the public key pk and $\xi_{sk}(c_1)$ and $\xi_{sk}(c_2)$ represents decryption of the ciphertexts c_1 and c_2 respectively using the secret key sk.

Homomorphic Properties: Suppose we encrypt two messages m_1 and m_2 utilizing the same public key pk which generates ciphertexts c_1 and c_2 respectively, and k is a consistent number. Then, *Paillier cryptosystem* [45] ensures the following homomorphic properties which can be used to perform count query:

- After decryption, we will achieve the sum of the corresponding plaintexts if we multiply two ciphertexts, after decryption .

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

- 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.

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

2.2.4 Oblivious-RAM

Oblivious RAM protocols were characterized by Goldreich and Ostrovsky[62] as *ORAM* which hides the access patterns made by the running project itself. This is progressively helpful today since we keep most of our data in the cloud and perform calculations there, but we often disregard what our programs are uncovering. The information or ram access can uncover what the program is doing even with encrypted data. So only encrypting the data is definitely not an adequate model now a days as we additionally need to randomize the access to it.

Usage

Many work have been done in ORAM, however we may take these two into considerations:

- Path-ORAM [63] (Security protocol).
- ObliviStore [64] (Secure Data Structure) as ObliviStore is the quickest ORAM implementation now (state of the art) which have *asynchronous and distributed* property making it benificial for any genomic computation framework. Path-ORAM comes into context since it has the lowest bandwidth cost and is the fundamental system of ObliviStore.

2.2.5 Secure Two-party Computation

Andrew Yao proposed the concept of secure computation in 1980s [10]. He presented 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 on the function f. The amount of work done by each party grows proportionally to the number of gates in the circuit

Table 2.1: Garbled truth table for an AND gate

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	$E_{A_1}(E_{B_1}(K_1))$	$E_{A_0}(E_{B_0}(K_0))$

evaluating function f. After executing the protocol, A and B learns the output of $f(x, y)$. However, they learn nothing about the input or any other information of the other party.

Construction and evaluation of Garbled Circuits (GC) 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 along with A's input. Suppose the input of A is $I(g)$. After receiving the circuit, the evaluator evaluates the circuit. He executes a 1-out-of-2 oblivious transfer protocol to obtain its private input, $I(e)$ [65] obliviously . Therefore, the evaluator can evaluate $f(x, y)$ from $I(g)$ and $I(e)$.

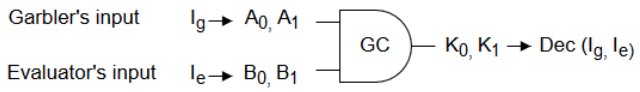


Figure 2.1: Garbled circuit for an AND gate.

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 [65], 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

Oblivious transfer (OT) was introduced by Michael O. Rabin [65] in 1981. It is a two-party (sender, receiver) cryptographic protocol where the sender sends a message to the receiver but remains oblivious whether the receiver receives the message or not. Later, Even *et al.* [66] 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.

Recursive Oblivious Transfer

In a recursive oblivious transfer, the sender repeats querying an element and sets the next query depending on the query result [67]. Not only the final result, but also the intermediate results sent from the receiver need to be hidden to protect the private information for the sender and the receiver [68].

Usage of Garbled Circuit

Since GC can calculate any arbitrary functions, it is the most utilized solution on secure genomic data computation [69]. The fundamental issue behind GC is the excessive data transfer needed for the key exchanges and bigger circuits. Moreover, this is a two party protocol meaning this works best when there are only two parties doing the calculation.

Available Implementation

There are various proficient implementations available for Garbled Circuits. These implementations are extensible and any calculation can be done safely utilizing these tools. Some of the notable tools of GC are—*FastGC* [70], *ObliVM-GC* [71], *JustGarble* [72], *FlexSC (Java)* [73], and *FlexSC (C/C++)* [74].

2.2.6 Bloom Filter

The objective of the bloom filter is to check if an element is present in a set with a small probability of false positive. In 1970, Burton H. Bloom [75] proposed it. It contains an array of fixed length m bits. Let, the Bloom filter, $\mathcal{B}$ represent 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:

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

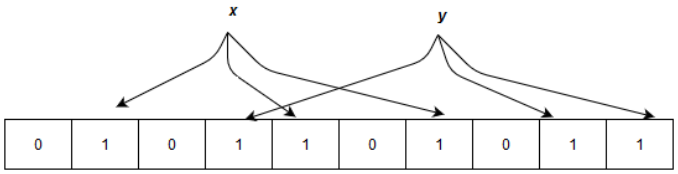


Figure 2.2: Bloom filter containing three hash functions and two strings (x and y).

The hash values are utilized as an offset into the bit array, $\mathcal{B}$ and the equivalent bits are set to 1. To check if a query, q is in $\mathcal{B}$, similarly k hashes are processed 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 relating bit is not 1, then that element is absent in $\mathcal{B}$. In this way, Bloom filter decides if an element is absolutely not in the set. Else, we accept that the element is available in the Bloom filter. Figure 2.2 shows a case 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 the length, the false positive probability, and the number of hash function of the Bloom filter respectively.

The principle leverage of Bloom filter is that it is quick. 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 guarantees better security, however the array rapidly fills up and the Bloom Filter turns out slower. Less hash functions make it quicker but results in chance of getting more false positive results [76].

2.2.7 Advanced Encryption Standard

Advanced Encryption Standard (AES) is an encryption technique which utilizes a block (the fixed-length group of bits) for encryption or decryption. I utilize a variation of AES known as *Counter Mode* (AES-CTR) [77] to encrypt the data. Counter mode turns a block cipher into a stream cipher. AES-CTR generates a keystream by encrypting *initialization vector (IV)* and consecutive values of a *counter*. The IV is an arbitrary number that can be used along with a secret key for data encryption. The *counter* is a n bit string which is non-repeating. Hence, AES-CTR produces a different cipher each time for the same plain text due to the unique counter (*padding IV and counter value*). 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 $x_i,\ y_i,\ \mathcal{ENC}_k,\ CTR,\ iv$ are i. plaintext, ii. ciphertext, iii. encryption key, iv. counter value, and v. initialization vector, respectively. Notably, AES-CTR is proven to be indistinguishable under chosen-plaintext attack or *IND-CPA* secure [78].

2.3 Security Requirement

In this thesis, I focus three security requirements while designing my proposed models. Ensuring these security requirements of the following three parameters are of paramount importance while designing a secure biomedical data outsourcing and computation mechanism:

1. ***Data privacy.*** The data stored in the cloud server, along with the computation, should be secured and should not release any information about the data. Regardless of whether the cloud server gets compromised, the confidentiality of the data should be guaranteed.

2. ***Query privacy.*** The institutions that share the data, the cloud service provider or an adversary in the ownership of a compromised server should learn nothing about a query executed by an institution or a researcher.

3. ***Output privacy.*** The query result should be concealed to all party other than the researcher who originated the query. Here, output privacy means not to prevent any inference attack caused by the results of the query. Such inference attack can be abstained utilizing *differential privacy* and has been studied widely in the literature [9].

2.4 Burrows-Wheeler Transform

The Burrows–Wheeler Transform (BWT) adjusts a character string into runs of comparable characters. This is helpful for compression since it packs a string that

has runs of repeated characters [79]. BWT is also utilized in pattern matching where all the occurrences of a pattern p can be discovered as a prefix of sequential rows of the BWT matrix, and these rows are found using a backward search process [80].

Chapter 3

Related Work

In this chapter, I provide an overview of the existing secure solutions related to my proposed methods for privacy-preserving biomedical data sharing and computation.

3.1 Secure Count Query

In recent years, various 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, I present an overview of the two proposed methods for secure count query operation on outsourced genomic data. To the best of my knowledge, these are the only two research work that has particularly addressed this problem. I also present some other relevant works that are closely related to this problem.

3.1.1 Using Homomorphic Encryption

In 2008, Kantarcioglu *et al.* [81] 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: Although this model provided the first solution for the addressed problem, it is not liberated from imperfections. This methodology has several drawbacks. *First*, colluding third parties might result in the exposure of sensitive information. *Second*, huge bandwidth is required for the correspondence between the DS and KHS. Also, the KHS needs to be online during the query execution for decrypting the final outcome. *Third,* the searching process during the query execution is linear to the number of records and the homomorphic encryption scheme utilized is very expensive. This resulted in a framework that is not practical for a large database (see

Section 4.4 for experimental results). *Fourth*, all data owners utilize 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 technique uncovers the data access pattern to the cloud server.

3.1.2 Using Cryptographic Hardware

In the subsequent work, Canim *et al.* [82] 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 [81]. Thus, this model overcomes the limitations of [81]. 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 [83] 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 [84] which impedes the processing of larger queries.

3.2 Secure Sequence Similarity Search using Hamming Distance

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.* [85] 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.

Zhang et al. [86] utilized *secret sharing* and *secure multi-party computation* for computing Edit distance between two sequences. Perl *et al.* [87] 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 2010, Rheinländer *et al.* [88] 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.* [89] applied a prefix tree based searching index for secure similarity search using edit distance as the similarity metric.

3.3 Secure Top-K Similar Patients Query Search

Existing techniques for secure edit distance computation on genomic data can be classified into outsourced and federated models. In the outsourced model, data owners encrypt their entire databases and then outsource these databases to a cloud server. The required computation will be conducted on the encrypted data stored on the cloud. Once outsourced, data owners can go offline. Hence, this approach has a communication overhead of transferring the encrypted databases to the centralized cloud server. In the federated model, data owners do not outsource their databases to a cloud server. Data owners compute on their plaintext data and only share partial aggregate information with the cloud server or other parties to compute the final output. Therefore, the communication overhead for such a model is relatively low. However, the federated model requires all data owners to remain online during the computation. Almost all existing techniques [1; 90; 91; 92; 2] adopt the federated model.

Jha *et al.* [90] conducted one of the initial works in privacy-preserving genomic sequence similarity. They presented three protocols which execute the original edit distance algorithm over garbled circuit. Nonetheless, due to the performance of the garbled circuit available that time, it took ~ 40 seconds to compute the edit distance between two sequences with a length of 25. Wang *et al.* [1] addressed the problem of approximating the original edit distance in a practical setting. The procedure used a

public reference genomic sequence to calculate an approximated edit distance between two strings. Nonetheless, the selection of a public reference leaks few information about the underlying data distribution. Furthermore, it affects the accuracy as the computation is done in accordance with a reference. Concurrently, Aziz *et al.* [2] also utilized two different approximation methods (comprising shingling, private set intersection, banded alignment, and garbled circuit) to compute the edit distance among genomic sequences.

To the best of our knowledge, there is only one contribution [48] that addresses the problem of edit distance computation by adopting the centralized model. Following the proposal of the fully homomorphic encryption (FHE) by Gentry, Cheon *et al.* [48] also suggested edit distance to be homomorphically computed via lattice encryption. Nevertheless, because of the current state of FHE, the scheme is still ineffective as it takes 27.5 seconds to calculate a 8×8 block of dynamic programming. Since the crypto behind the FHE improves and advances, in future we might observe a better usage of this. There are also some other related studies [91] which address approximating or securely computing the edit distance. Table 3.1 presents the summary of them. We can notice from the table that our method is the only one that overcomes the three privacy challenges together and it is feasible for practical implementation at the same time.

In addition, there are some related work that addressed computing kNN on encrypted outsourced data to the cloud [93; 94] and outsourcing the biometric identification [95]. Although related, these articles do not address the problem of secure SPQ on genomic data.

Table 3.1: Secure genomic data similarity methods where n, m are record and SNP size respectively.

Authors	**Year**	**Data** ($n \times m$)	**Time** (s)	**Principal Method**	**Privacy**			**Architecture**
					Data	**Query**	**Output**	
Jha *et al.* [90]	2008	25×25	< 40	Smith-Waterman	✓			Federated
Wang *et al.* [91]	2009	400×400	28.5	Custom protocols	✓			Federated
Wang *et al.* [1]	2015	2000×9000	2800	Private set difference on a reference sequence		✓		Federated
Cheon *et al.* [48]	2015	8×8	27.5	Homomorphic encryption	✓			Outsourced
Asharov *et al.* [92]	2017	500×3500	2	Custom protocols		✓		Federated
Aziz *et al.* [2]	2017	50×3500	5	Custom protocols		✓		Federated
Our method	2017	50×3500	30	Compressed prefix tree	✓	✓	✓	Outsourced

3.4 Secure Substring and Set-maximal Search

We can divide the existing works of string search into four classes according to the length of the query sequence and database sequence:

- Query sequence length is smaller than the database sequence length.
- Query sequence length is smaller than the database sequence length and the query initiator is interested in exact match from a particular position.
- Query sequence length is smaller than the database sequence length and the query initiator is interested in maximum match from a start position.
- Query sequence length is equal to the database sequence length and query initiator is interested in maximum matches along with the match start and end position (*set-maximal matches* [96]).

In this chapter, we are interested in the second and third aforementioned problems.

Table 3.2: Previous research works in secure and privacy-preserving string search in genomic sequence.

Existing techniques	Year	Query solved	Query start position	Principal Method
Atallah *et al.* [97]	2003	Substring search	✗	Dinamic programming with HE
Troncoso *et al.* [98]	2007	Substring search	✗	Finite automata with ROT
Katz *et al.* [99]	2010	Substring search	✗	GC
Sudo *et al.* [100]	2015	Substring search	✗	Wavelet matrix with AHE
Shimizu *et al.* [3]	2016	Set-maximal search	✓	BWT with ROT
Ishimaki *et al.* [101]	2016	Substring & Set-maximal search	✓	BWT with FHE
Our method	2019	Substring & set-maximal search	✓	GST with GC

3.4.1 Substring Search without a Query Start Position

Secure substring matching is a familiar problem and a popular research area. Katz *et al.* [99] proposed a secure DNA sequence search (exact match) using garbled circuit [10]. Atallah *et al.* [97] utilized dynamic programming to compare between two sequences securely using homomorphic encryption. Other researchers utilized finite automata with `recursive oblivious transfer (ROT)` [67] to solve DNA searching problem [102; 98]. Sudo *et al.* [100] proposed a unique algorithm for privacy-preserving substring search. They utilized secure `wavelet matrix` and `additively homomorphic encryption` [100] to search a string in logarithmic time.

3.4.2 Substring Search

Although related, the aforementioned works do not address the problem of secure substring search (exact match) with a starting position. In this chapter, we proposed a secure model to find a string match between query and database sequences

where query contains a starting position. Moreover, we are interested finding the exact match count rather than the maximal matches. To the best of our knowledge, Ishimaki *et al.* [101] only addressed this problem. To solve the problem, Ishimaki *et al.* [101] created a look-up table from `positional Burrows-Wheeler transform (PBWT)` [11] as preprocessing and utilized `fully homomorphic encryption (FHE)` with bootstrapping optimization. Moreover, to reduce the incorrect decryption (due to random noise inside ciphertext), the authors proposed two solutions:

- Set a threshold noise to ensure correct decryption.
- Reset the noise by the bootstrapping method.

3.4.3 Set-maximal Search

Privacy-preserving set-maximal search is another indispensable problem that help identification of distant relatives. Shimizu *et al.* [3] introduced a secure variable length prefix/suffix match on SNP sequences or regular text. For preprocessing, the authors utilized PBWT proposed by Durbin *et al.* [11] as the data structure whereas we utilized a generalized suffix tree to preprocess the data. To assure the privacy, we utilized garbled circuit whereas the authors used `ROT` method of the additive homomorphic encryption as the secure protocol. Ishimaki *et al.* [101] extended this work to support statistical analysis. For this, they utilized `FHE` to support both addition and multiplication in encrypted form. In their protocol, a server holds some aligned genome sequences and a user holds a genome sequence. After secure computation, the user can obtain only the count of prefix match larger than a given threshold without

knowing the server sequences.

Table 3.2 summarizes the previous works in privacy-preserving string search. We categorize the previous works into different query solved and appearance of query start position in the query as well.

3.5 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, or secure similarity search, they target closely related problems and use different cryptographic techniques to ensure the security of the genomic data. Here, I mention some of these works.

Ayday *et al.* [103] 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.* [104] 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.

Statistical Analysis

To protect the privacy of the genome database, Lauter *et al.* [50] 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. [105] 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. [106] 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) [107]). 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.* [108] 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 Disequilib-*

rium using this model. Zhang *et al.* [109] used homomorphic encryption to compute *Chi-Squared Test* in untrusted public cloud.

Choi *et al.* [110] 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.* [111] proposed a scheme for securely performing *meta-analysis* for genetic association study. Instead of storing data to multiple cloud storage (like Kamm *et al.* [105] and Zhang *et al.* [86]), they kept the data in the corresponding data owner's premises. Wang *et al.* [112] 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.

Chapter 4

Secure Count Query

Human genome research may expose an individual's private information, like susceptibility to a particular disease, for example, Alzheimer, diabetes, and breast cancer [4]. We can assess this by querying an individual's genome across the common variations and then predicting the predisposition [4]. This kind of analyses needs pervasive genomic sequences to achieve better accuracy that sometimes a single organization cannot afford [5]. One conceivable arrangement can be allowing access of the genomic data to outside of the organization. But releasing the data to the government organization or research institution, can cause major risks of data privacy.

To share data among multiple government organizations or research institutions, various privacy policies have been developed. Since each individual has unique genome sequence, exposure of this sequence has major privacy risks. For instance, an insurance agency may preclude the health coverage of an individual who holds a specific gene that lead to a specific disease. Therefore, privacy-preserving way should be enforced to store and analyze genomic sequences.

Figure 4.1: Data from various sources is used in biomedical informatics.

To store the data, cloud computing services are generally utilized because of cloud infrastructure's affordable low rate. But cloud services are unsafe and may cause data security breach. We should not expect privacy from cloud service providers [6].

Our objective is to share and secure computation of outsourced genomic data. Overall, we guarantee three important security concerns. They are: data privacy, output privacy, and query privacy. `Data privacy`: The data uploaded to the cloud server should be secured. `Query privacy`: Data owners, cloud or any adversary should not know anything about a researcher's query. `Output privacy`: Only the researcher, who initiated the query, will know the result of the query.

Since suppression and generalization can alter the content of the original sequence, anonymization methods are inapplicable for preserving the genomic data. Cryptographic techniques can calculate a predefined function and send the function's result without exposing sensitive information from multiple parties [8]. Therefore, different cryptographic privacy-preserving methods have been developed to acquire the objective of encrypted genomic data sharing and computation. In particular, Kantarcioglu

Table 4.1: Existing techniques

Algorithms	**Method**	**Trusted Entity**	**Privacy**		
			Data	**Query**	**Output**
Kantarcioglu *et al.* [81]	Paillier	Online	✓		
Canim *et al.* [82]	Cryptographic Hardware, AES	N/A	✓		✓
Our method	Paillier, GC, AES	Offline	✓	✓	✓

et al.[81] and Canim *et al.*[82] utilized genomic data to solve secure count query problem. However, their solution can not guarantee the security prerequisites together: *data privacy, output privacy, and query privacy.* Our proposed method solves the secure sharing and computation problem on heterogeneous data by preserving aforementioned challenges.

We utilize a heterogeneous dataset that includes genotype, phenotype, and numeric data. The addition of the numerical data is an interesting challenge because of its usability in real-world applications. Currently, data from numerous sources are being used in clinical informatics: data from wearable sensors [113], cell phones [114], participatory sensing applications [115] (Figure 4.1). Datasets containing clinical information include clinical profiles along with the genomic sequences and numeric data such as age or blood sugar level. We consider a single sequence (S) comprises of multiple SNPs, $S = \{a_1, a_2, ..., a_n\}$, where a_i represents a SNP. Table 4.2.1 describes the data format. Each row displays genomic sequence(genotype data), diagnoses(phenotype data), and age(numeric data) of a single patient. SNPs $a_1, a_2, ..., a_n$ is represented in a single column. In general, A SNP, a_i is presented as a pair of nucleotides [82]. The last column represents the patients' age as the numeric data.

In this chapter, we propose a secure framework that enables outsourcing genomic data in a cloud server and then execute count query on it. Since we have mentioned earlier, to the best of our knowledge, [81] and [82] 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 propose scalable solution for real-life applications. Table 4.1 represents a comparison of the two existing solutions with our method. This table summarizes various aspects of the solutions such as methodology, involvement of trusted entity during query execution and various kinds of security requirements.

4.1 Problem Definition

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: $|\ \{\forall i,\ d_i \in D\ \ |\ d_i \text{ satisfies } \theta\}\ \ |$.

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
AND Age between 35 to 65
```

Query 4.1: Example of a query

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

Table 4.2: Dataset representation

	Sequence							
Patient id	**SNP_1**	**SNP_2**	**SNP_3**	**SNP_4**	**SNP_5**	...	**Diagnoses**	**Age**
1	AG	CC	TT	AG	CT	...	Headache, High cholesterol	55
2	AA	CC	CT	AG	CT	...	Arthritis	77
3	AG	CT	CC	AA	TT	...	Hair Loss, Mumps	81
4	AG	CC	TT	AG	CT	...	Nausea, Asthma, Cold	69
5	GG	CT	TT	GG	CC	...	Acid reflux	40
6	AA	CC	TT	GG	CC	...	High blood pressure	35
7	AG	CT	CT	AG	CT	...	Migraine	24
8	AA	CC	TT	GG	CC	...	High blood pressure	65
9	GG	CT	CT	AG	CT	...	Obesity	44
10	AG	CT	CT	AG	CT	...	Hypertension	47

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 Preliminaries

4.2.1 Data Representation

In this section, we present the format of the data used in this research. The database contains not only genomic but also 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 three types of datasets

- datasets with genotype only, datasets with genotype and phenotype information, and lastly datasets with genotype, phenotype, and numeric information. We observe examples of these kinds of datasets in real-life applications. In the dataset which contains only genotypes, all the patients are separated into a case-control group based on their medical condition (e.g., individuals with or without a particular disease). Some clinical datasets contain both the patients' genomic sequences and the diagnoses associated with the SNPs. Moreover, some heterogeneous dataset contains the numeric data such as age or sugar level along with the genotypes and phenotypes data.

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 4.2.2). 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 [82; 116]. For each patient, phenotypes are listed as the diagnoses or genomic conditions. The last column in Table 4.2 represents the age as the numeic value.

4.2.2 System Architecture

Figure 4.2 represents a general architecture of our proposed framework. As depicted in the figure, it incorporates four principle members: *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 useful. The

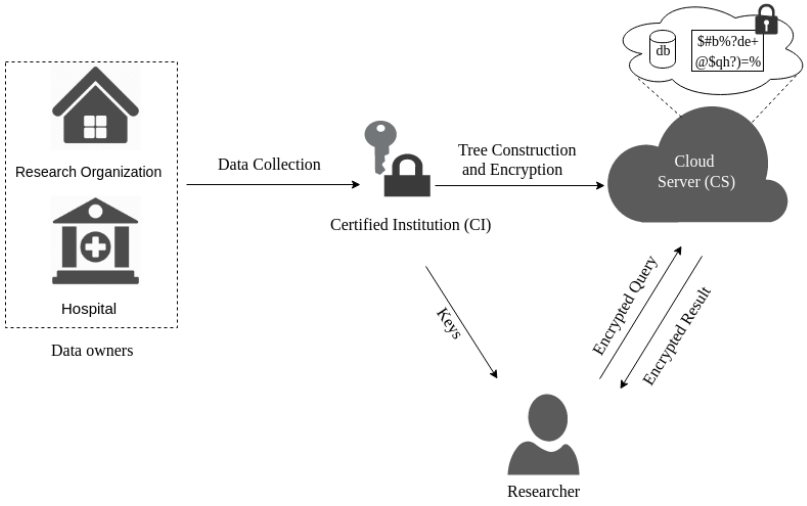


Figure 4.2: System architecture.

jobs performed by each of the entities are discussed below –

1. ***Data Owners.*** *Data owners* comprises of the institutions who consented to share the genomic data they have. 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 [117]. They send the genomic data to the *CI* in plaintext. Preceding sending the data to the *CI*, *data owners* process their data in a formerly agreed format.

2. ***Certified Institution (CI).*** The data shared by various *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 principle duties performed by *CI*:

   (a) ***Generation of Index Tree.*** After getting the data from the contributing *data owners*, *CI* constructs an encrypted searchable version of the

aggregate shared data and sends it to the *CS*. The search operation is fundamentally performed on an encrypted *index tree*. In our proposed framework, the *CI* constructs 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 deal the keys used for encryption and decryption. It supplies 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* provides the essential 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 performed on this tree. *CS* is liable for taking care of all the communications with the *researchers*. The *researchers* send their encrypted query to *CS*, *CS* then performs this query and sends back the encrypted result to the *researchers*.

4. ***Researchers.*** *Researchers* can be any organization or individual who is keen on 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* utilize the public key to encrypt their query and then send it to the *CS*. *CS* assesses this query on the encrypted tree and sends back the encrypted outcome to the *researchers*. After decrypting this

result utilizing the secret key, the *researchers* achieve the final outcome.

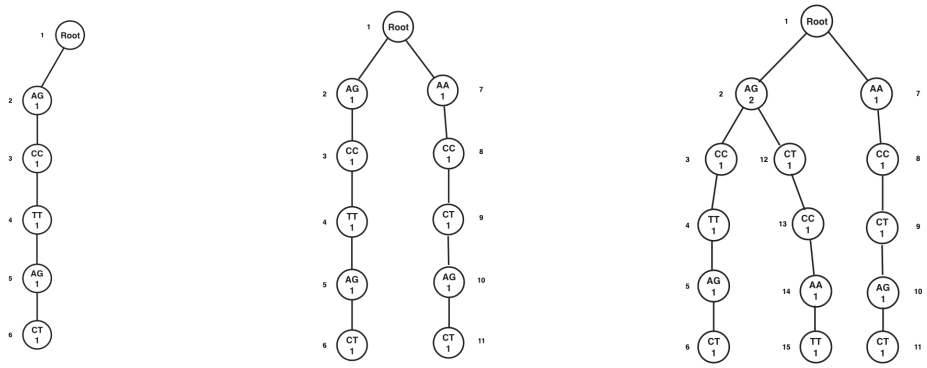
4.2.3 Threat Model

Our objective is that the *CS* does not learn anything about the shared genomic data and both the *CI* and *CS* learn nothing about the query executed by the *researchers*. Besides, we need to guarantee that the *researchers* do not gather any information from the data. We expect the *CI* to be trusted entity as it is responsible for the generation and encryption of the *index tree*. The *CI* can check the identity of the organizations or individuals 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 equivalent as the *Data Access Committee (DAC)* of NIH [117].

In our proposed framework design, we expect that the *CS* to be *semi-honest*, otherwise called *honest but curious* adversary [118]. This adversary effectively follows the protocol and does not have the intention to behave maliciously to produce the incorrect result. Nevertheless, they may try to assemble more information than expected during or after the protocol execution. Hence, we require each party does not unveil any information during protocol execution. Accordingly, we accept that none of the parties *data owner*, *CI*, or the *CS* has any expectation to act maliciously in the desire of creating incorrect outcome.

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

- We expect that the *CI* does not collude with the *CS* and *CS* additionally does not collude with the *researchers*. This is a basic necessity for ensuring data and query privacy.



(a) (b) (c)

Figure 4.3: Various states of the *index tree* generation. Figure 4.3a, 4.3b, and 4.3c illustrates the tree after the first, second, and third record insertion respectively.

- We expect that the keys received by the *researchers* from the *CI* are accurate.

4.3 Methodology

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

We initially present our model with genotype data and then we discuss how to coordinate the phenotype and numeric information into our index tree.

4.3.1 Genomic Data

We will initially discuss how to build the index tree with the genotype data only and then the search technique on this tree.

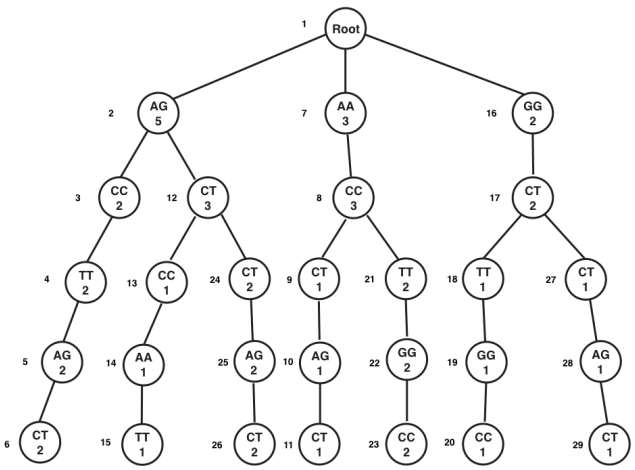


Figure 4.4: *Index tree* for Table 4.2 (genotypes information only).

Generation of *Index Tree*

When the *CI* receives the data from the *data owners*, it initially builds a search tree, T which we call the *index tree*, utilizing the SNPs from the database D. There is only a single such tree in our framework. After the building 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 4.3 and 4.4, 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 4.3 and 4.4).

We denote a node as σ and represent as, $\sigma(sid,\ val,\ count,\ list)$. 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 4.3a. 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 4.3b. 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 4.4 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 4.4, 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 count 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
```

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* [45]) 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}
```

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 [119].

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:

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

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 implies 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 procedure proceeds until *CS* discovers all the matched nodes for the corresponding query or *CS* looked through all the nodes of $\widetilde{T}$.

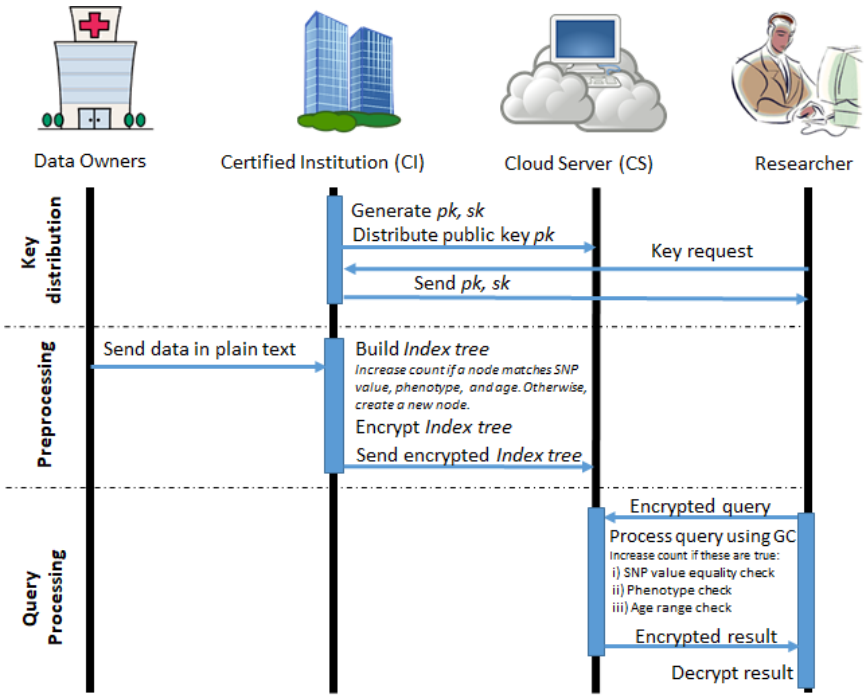


Figure 4.5: Workflow of our proposed method. It consists of three processes: key distribution, preprocessing, and query execution.

Let q be the query consisting of the SNPs the *researchers* are interested in and r 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 4.5 summarizes each of the steps of our proposed technique.

4.3.2 Heterogeneous Data

The tree depicted in Figure 4.4 is only capable of counting the number of genotypes at a specific position of the genome. To decide the impact of heterogeneous data, we need to incorporate the phenotype and numeric information at every node of the tree.

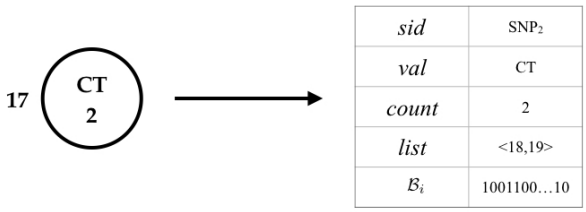


Figure 4.6: Information stored in a single node.

Integration of Phenotype Information to the *Index Tree*

We utilize Bloom filter to incorporate the phenotype information to the *Index Tree*. Bloom filter can be utilized to check the membership of an element in a set. We use this to facilitate search operation over the *index tree*. Specifically, we include a Bloom filter as the fifth component of each of the node in the tree to consolidate the phenotype information [120].

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 inclusion of the relating phenotypes in the Bloom filter. Each phenotype from $\sum$ is mapped to a unique number and that number will be embedded 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 4.6 represents all the information stored in a single

node.

Example 3. Figure 4.3a 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.

While inserting record # 3 (see Figure 4.3c), 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).

Now, we discuss the integration process of the numerical data along with the genotype and phenotype data. Next, we explain how to encrypt phenotype and numeric data, and count query search on encrypted heterogeneous data.

Integration of Numeric Information to the *Index Tree*

Similar to phenotypes insertion, we add numeric information such as `age` as the sixth component of each of the node in the tree. Each node of the tree now consists of `age.low` and `age.high` that represents the age range of the current node and its children. For example, for a particular node, `age.low` and `age.high` values are 55 and 81, respectively represents that the node itself and its children contain age which lie in between 55 and 81. Figure 4.7 and 4.8 represents the updated version of the *index tree* (Figure 4.3 and 4.4) with the integration of the numeric information.

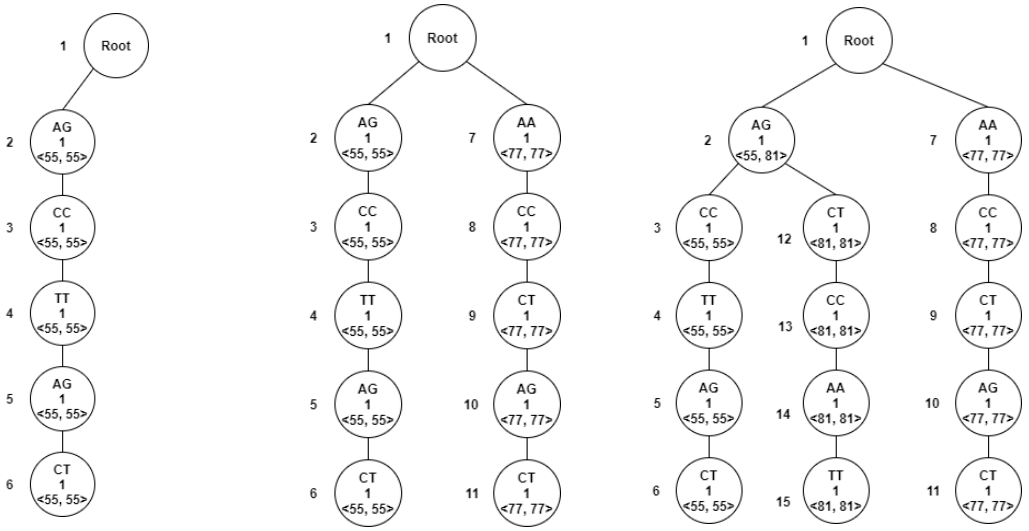


Figure 4.7: Updated states during the generation of *index tree* with integration of the numeric data. Figure 4.7a, 4.7b, and 4.7c represents the tree after the insertion of the first, second, and third record, respectively.

The overall procedure to generate the *index tree* including the genotype, phenotype, and numeric information is described in Algorithm 1.

Encryption of the Bloom Filter and the Age

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 $\widetilde{\mathcal{B}}_i = \mathcal{B}_i \oplus s$.

To encrypt the `age` information in each node, *CI* also uses the public key, *pk* generated by the *CI*. Therefor, after the encryption of the heterogeneous data (includes genotype, phenotype, and numeric information), each node of $\widetilde{T}$ becomes $\sigma(sid,$

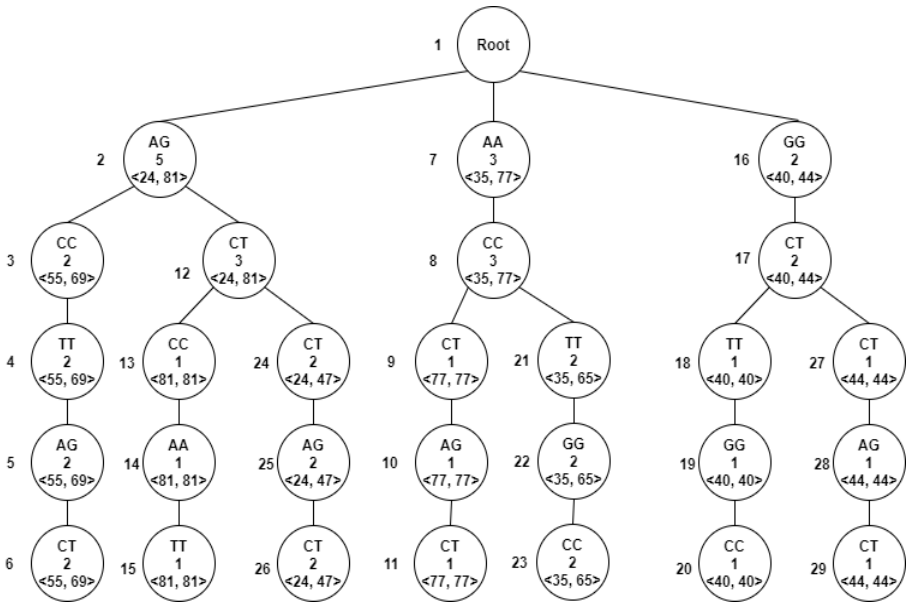


Figure 4.8: Updated *Index tree* for Table 4.2 (with numeric information).

$\xi_{\text{pk}}(val)$, $\xi_{\text{pk}}(count)$, $\xi_{\text{pk}}(\mathcal{B}_i)$, $\xi_{\text{pk}}(age.low)$, $\xi_{\text{pk}}(age.high)$, $list$).

These encryption is done at the same time when other data at each node are encrypted as discussed in Section 4.3.1. The encrypted Bloom filter can efficiently be decrypted inside secure function evaluation (SFE) by garbled circuit during the search on the tree. After the encryption, *CI* sends the encrypted *prefix tree* to the *CS*.

Query Processing of the Phenotype and the Numeric Data

Besides providing the *Researchers* the public key and secret key (pk, sk), 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

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)
2:     for each record D_i do
3:         parent ← r
4:         for each column D_i^j do
5:             Create a new node, a with the corresponding sid, val, phenotypes, age
6:             for each node, n ∈ T do                          ▷ T is the tree being generated
7:                 if sid and val of a matches with n and Bloom filter of n contains the phenotypes of a then
8:                     n.count++
9:                     parent ← n
10:                    if a.age < n.age.low then
11:                        n.age.low ← a.age
12:                    end if
13:                    if a.age > n.age.high then
14:                        n.age.high ← a.age
15:                    end if
16:                    Delete node a
17:                else
18:                    a.count ← 1
19:                    a.age.low ← age
20:                    a.age.high ← age
21:                    parent.addChild(a)
22:                    parent ← a
23:                    Insert phenotypes of a in all the parent's Bloom filter
24:                end if
25:            end for
26:        end for
27:    end for
28:    return T
29: end function
```

filter generated at the *researcher's* end as $\mathcal{B}_q$. The *researchers* does not need to encrypt the phenotype and the numeric information since the *researchers* sends those directly to the garbled circuit.

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 garbled 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 the *researcher*. The encrypted Bloom filter $\widetilde{\mathcal{B}}_i$ is decrypted using the SFE inside the circuit as

$$\mathcal{B}_i = \widetilde{\mathcal{B}}_i \oplus s$$

Then each of the positions from Z will be checked inside the garbled circuit.

Protocol 4.3.2 *checkEquality()*

1 : *CS* Sends $\widetilde{\delta}$ to *researcher*

2 : *Researcher* decrypt $\widetilde{\delta}$ as $\delta = \xi_{sk}(\widetilde{\delta}) \ = \ val_n + \ \mu$

3 : *Researcher* calculate noise as $\mu' \ = \ \delta \ - \ val_q$

4 : $a \leftarrow$ *CS* and *researcher* check the equality of μ and μ' using SFE (**true** or **false**)

5 : $b \leftarrow$ *CS* and *researcher* check the indexes of Bloom filters $\mathcal{B}_n$ and $\mathcal{B}_q$ using SFE

6 : **return** $a \cap b$

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

function `SEARCH(node, index)`
  $count \leftarrow 0$
  Get the sid, (s) from list of SNP identifiers
  for each node, $n \in \widetilde{T}$ **do** ▷ $\widetilde{T}$ is the encrypted `index tree`
    if $n.sid = s$ **then**
      Generate a random number μ
      $\widetilde{\delta} \leftarrow \xi_{pk}(val_n) + \xi_{pk}(\mu)$ ▷ Encryption using the public key
      $circuitOutput \leftarrow$ Protocol 4.3.2
      if `circuitOutput` = **true then**
        Generate two random numbers μ_1 & μ_2
        $\widetilde{\delta_1} \leftarrow \xi_{pk}(n.age.low) + \xi_{pk}(\mu_1)$
        $\widetilde{\delta_2} \leftarrow \xi_{pk}(n.age.high) + \xi_{pk}(\mu_2)$
        CS Sends $\widetilde{\delta_1}, \widetilde{\delta_2}$ to *researchers*
        Researchers decrypt $\widetilde{\delta_1}$ as $\delta_1 = \xi_{sk}(\widetilde{\delta_1}) = (n.age.low + \mu_1)$
        Researchers decrypt $\widetilde{\delta_2}$ as $\delta_2 = \xi_{sk}(\widetilde{\delta_2}) = (n.age.high + \mu_2)$
        $decryptedAge.lowInGC \leftarrow (\delta_1 - \mu_1), decryptedAge.highInGC \leftarrow (\delta_2 - \mu_2)$
        $lowRangeCheckInGC \leftarrow (decryptedAge.lowInGC \geq query.age.low)$
        $highRangeCheckInGC \leftarrow (query.age.high \leq decryptedAge.highInGC)$
        if $(lowRangeCheckInGC \cap highRangeCheckInGC)$ **then**
          $\widetilde{count} \leftarrow$ `n.getCount()` ▷ $\widetilde{count}$ is the encrypted count value
        end if
      end if
    end if
  end for
  `children` $\leftarrow$ `node.getChildren()`
  for each `child` in `children` **do**
    `result` = `Search(child, index)`
    if `count` $>$ 0 **then**
      `count` = `count` + `result`
    else
      `count` = `result`
    end if
  end for
  return `count`
end function

Search with Numeric Information in the *Index Tree*

To search numeric information in *index tree*, we need to check the age range of the query whether it lies between the `node.age.low` and `node.age.high`. For this, we

need to check two conditions: `query.age.low >= node.age.low` and `query.age.high <= node.age.high`. Again, `node.age.low`,`node.age.high` are encrypted. Therefore, we use the random mask to avoid the decryption inside the garbled circuit. The detailed process of checking `query.age.low >= node.age.low` is the following:

We refer the additive mask we use as *noise* and denote it as μ_1. After the addition of the *noise*, the encrypted masked value of *age.low* is:

$$\widetilde{\delta_1} \;=\; \xi_{pk}(age.low) \;+\; \xi_{\mathrm{pk}}(\mu_1) \tag{4.4}$$

Here, $\mu_1 \in \mathcal{M}$ where $\mathcal{M}$ is the message space and μ_1 is random. *CS* then sends the resulting obscure value $\widetilde{\delta_1}$ to the *researcher*. *Researchers* get the masked value after the decryption as:

$$\delta_1 \;=\; \xi_{sk}(\widetilde{\delta_1}) \;=\; age.low + \;\mu_1 \tag{4.5}$$

As μ_1 is random, the *researchers* will get random values for δ_1 after decryption from Equation 4.5. As a result, though δ_1 is revealed to the *researchers* from Equation 4.5, they will not be able to infer age information from it.

The *researchers* is the garbler of the circuit through which we check `query.age.low >= node.age.low`. The input of the *researchers* is δ_1 and *query.age.low*. The input from *CS* (evaluator) to this circuit is the actual noise it added, μ_1. Inside the GC, at first, following subtract operation will reveal the *node.age.low*:

$$\delta_1 \;-\; \mu_1 \tag{4.6}$$

Then, using GC, we can perform the `>=` operation with the *query.age.low* with *node.age.low* to check whether `query.age.low >= node.age.low` or not. Similarly,

Table 4.3: Configuration.

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

we check `query.age.high <= node.age.high`. Only *CS* knows these output and *CS* then continues traversing the children of that node if both `query.age.low >= node.age.low` and `query.age.high <= node.age.high` are $True$. This process continues until *CS* finds all the matched nodes for the corresponding query or *CS* searched all the nodes of $\widetilde{T}$.

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

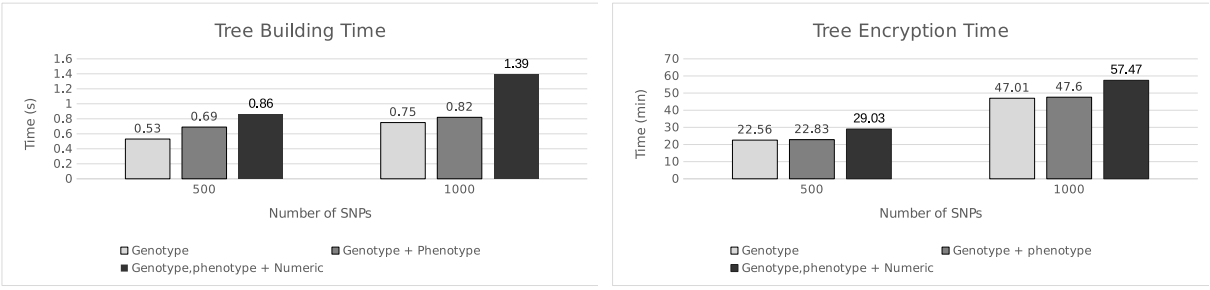
4.4 Performance Analysis

We have constructed a model of our privacy preserving framework to assess its practicality and tested its performance on real datasets. The *CS* and the *CI* run on two different machines. The configuration of the *CS* and *CI* is depicted in Table 4.3. The source code is written in JAVA programming language.

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

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

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



(a) Tree Building Time.

(b) Tree Encryption Time

Figure 4.9: Tree building and encryption time with various number of SNPs.

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* [70], *ObliVM-GC* [71], *JustGarble* [72] and used the *FlexSC* [73] library to implement the garbled circuits. We also used the *Paillier Cryptosystem* [45] to implement the *homomorphic encryption*.

We assessed our framework on real life dataset with 500 and 1000 SNPs. For real life dataset, we utilized the Personal Genome Project (PGP) [121] data which contains genotype, phenotype, and numeric data. We experimented with three different query sizes that included 10, 50 and 100 randomly selected SNP sequences. We also experimented with different number of records (500, 1000, and 1500) 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. The is because the structure of the index tree remains same for a fixed number of SNPs. For each experiment, we executed 10 runs and averaged the result over the runs.

A. Tree Building Time. We analyzed the time required to construct the *index tree* for various 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 4.9a plots the time required for building the tree with heterogeneous data. 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 4.9b 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` and `age` 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 various query of different sizes on the encrypted tree. The queries we utilized were controlled 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, the execution time decreases. This is because the increase in query size also

Table 4.4: Query execution time (seconds).

# of SNPs	**500**			**1000**		
Query size	10	50	100	10	50	100
Genotype	9.8	5.1	4.2	9.3	6.9	5.9
Genotype, Phenotype	67.23	54.33	10.77	126.61	113.86	10.52
Genotype, Phenotype, Numeric	103.96	83.69	36.85	163.92	143.9	36.81

increases the probability of finding a matched node adjacent to the root. We observe that the query execution time is higher for the numeric information. This is because of the range comparison via SFE in particular nodes.

Table 4.5 presents the comparison of the query execution time of our method with [81] and [82]. We report the times of [81] and [82] directly from the original articles. However, we have estimated the running time of [81] using the same platform as our model. The query execution time for [81] 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 [82] 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 technique is linear to the number of SNPs whereas it is linear to the number of records for both [81] and [82].

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

Query size	**10**	**20**	**30**	**40**
Kantarcioglu *et al.* [81]	25 min	27 min	28 min	30 min
Canim *et al.* [82]	20 sec	40 sec	60 sec	80 sec
Our method	2.4 sec	2.7 sec	1.5 sec	1.4 sec

Table 4.6: Communication overhead in MB.

# of SNPs	**500**			**1000**		
Query size	10	50	100	10	50	100
Genotype	0.09	0.05	0.04	0.08	0.05	0.05
Genotype, Phenotype	5.60	10.28	11.16	11.06	21.06	21.95
Genotype, Phenotype, Numeric	5.72	10.41	11.35	11.11	21.18	22.13

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.

5. Scalability. The proposed method is scalable for large datasets. Figure 4.10 shows a scalability graph where it needs around 3.5 minutes to process a query with 1500 sequences.

6. Storage Analysis. Table 6.4 lists the amount of spaces required to represent

Figure 4.10: Count query execution time by varying number of records. Each record contains 500 SNPs, whereas query contains 100 SNPs.

Table 4.7: Comparison of sizes (MB).

# of SNPs	**500**			**1000**		
Database	Original	Unencrypted Tree	Encrypted Tree	Original	Unencrypted Tree	Encrypted Tree
Genotype	0.45	13.98	64.7	0.89	25.9	129.8
Genotype, Phenotype	0.61	16.13	95.94	1.1	31.11	188.2
Genotype, Phenotype, Numeric	1.1	65	97	1.52	150	190

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* [45].

4.5 Security Discussions

We expect that the security of our proposed framework 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 construe information in various phases of the framework. 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 liable for the generation and encryption of the *index tree* and is considered as a trusted entity. Therefore the leakage to the *CI* is none in this stage. The *CS* cannot construe 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 outcome 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. Therefore, the *researchers* cannot construe anything from the value of μ'. Note that we do not consider here any privacy leakage through the output. Such inference attack can be abstained utilizing *differential privacy* and has been studied widely in the literature [9].

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.

4.6 Summary

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

- Our technique can successfully preserve both data privacy and data utility supporting large datasets by constructing an *index tree*. We notice that the time required to read the data from the database and construct *index tree* using this data is linear. Moreover, the tree encryption time does not have direct effect on the number of records.

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

- We incorporate numeric information with the genotypes and phenotypes data by adding `age` as the node attribute. By similar way, we can also handle multiple numeric attribute.

- Our proposed model is also scalable for large datasets.

These attributes make our proposed technique a promising one for executing count query securely on encrypted heterogeneous data.

Chapter 5

Secure Sequence Similarity Search

Medication compatibility is important to determine the appropriate and specific medication for a patient. For this, the user needs to share their genome with the hospital or medical unit (i.e. pharmaceutical companies). Moreover, the medical units need checking the compatibility of their developed medication by conducting medical tests on the user's genomic information. Because of the privacy considerations, the users do not need want to share their genomic data with the medical units. The medical units are also reluctant to reveal the properties of their medication under development.

To bridge the gap between the contradictory goals of the users and the medical unit, a secured personalized medical solution is required. One of the steps to provide the personalized medical solution is to identify similar patients from a genome database [122]. This is called similar sequence/patient query. In this chapter, we provide a secure technique for the similar sequence search.

5.1 Similar Patient Query

Similar Patient Query (SPQ) is the recognition of similar patients from a large number of structured Electronic Medical Records (EMRs). The major focus of SPQ 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 [123]. 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 SPQ it is required to execute a query on the encrypted data which preserves the privacy of the patients.

5.2 Problem Definition

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:

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

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 $u = v$

Table 5.1: Sample Genomic data representation

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

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 Data Representation

Table 5.1 presents 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 Similar Sequence Search Query

Our goal is to securely perform 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: Example of a query

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 Architecture

The proposed system architecture is the same as presented in Figure 4.2 in Chapter 4. So, again there are four entities: *Data Owners*, *Certified Institution (CI)*, *Cloud Server (CS)* and *Researchers*.

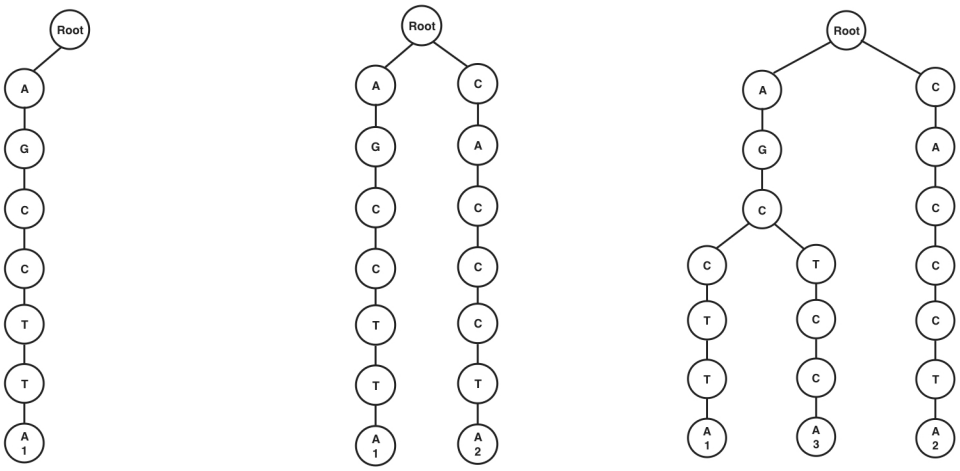
The *Data Owners* shares the genomic data with the *CI*. The *CI* builds an encrypted searchable version of the aggregated shared data which we call *prefix tree* and sends it to the *CS*. The search operation is basically executed on this encrypted *prefix tree*. *CI* also manages the encryption keys. 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 the SPQ problem is also similar to the threat mode described in Section 4.2.3. Our goal 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 Methodology

In this section, we describe 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*

At first, *CI* creates a search tree, T (which we call *prefix tree*) after receiving the data from the *data owners*. To build the tree, *CI* utilizes 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 2.1.1, there are four nucleotides $\{A, C, G, T\}$. We used 3-bit encoding (A= 000, C= 011, G= 101, and T= 110) so that the hamming distance between any two dissimilar nucleotides becomes similar (2). Thus, we divide the final distance by 2 to achieve the actual hamming distance. Instead of using 2-bit encoding to encode A, T, G, C : {00, 01, 10, 11}, we use 3-bit encoding since we want similar hamming distance between any two dissimilar nucleotides. If we use 2-bit encoding ($A = 00$, $C = 01$, $G = 10$, and $T = 11$), the hamming distance between A and T becomes 2, and the others become 1. 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 $\{000, 011, 101, 110\}$ 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 $\{000, 011, 101, 110\}$ for each of the 4 possible sequences.

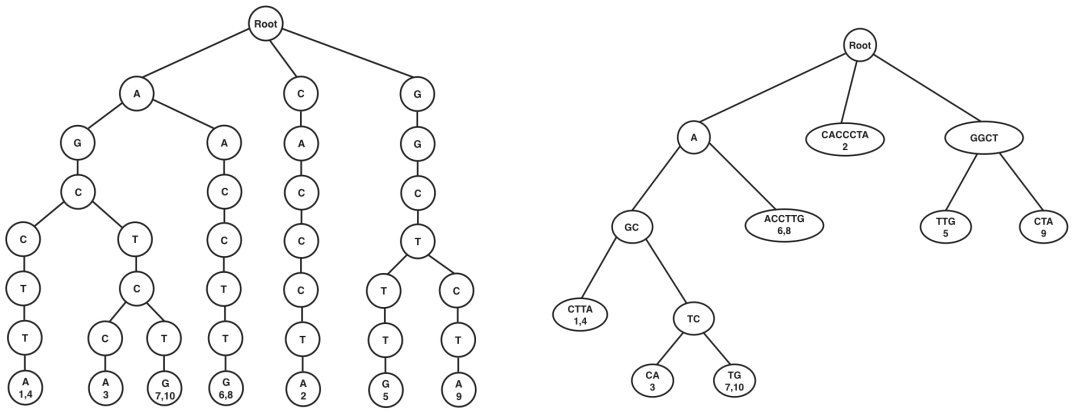
b) *id*: we keep a unique identifier to each record inserted in tree T so that we can reference the original record using this identifier later. If several records in the database D contain the same sequence, the corresponding leaf node will contain multiple *id*s.

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 $\sigma(val, id)$. 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 $\sigma_1(A, \langle G \rangle)$.*

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 corre-



(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.

sponding 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 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 Algorithm for building prefix tree

Input: Root node and the database (D)

Output: A prefix tree, T

```
1: for i ← 1 to n do
2:     InsertSequence(T.root, D_i);
3: end for
4: function InsertSequence(T.root, D_i)
5:     currNode ← T.root;
6:     l ← D_i.length                          ▷ length of the current sequence
7:     for j ← 1 to l do
8:         currNucleotide ← D_i^j;
9:         flag ← 0;
10:        for k ← 1 to currNode.numOfChildren do
11:            if currNode.getChild(k) == currNucleotide then
12:                currNode = currNode.getChild(k)
13:                flag ← 1;
14:            end if
15:            if flag == 0 then
16:                a = π(D_i^j)
17:                currNode.addChild(a)
18:                currNode = a
19:            end if
20:        end for
21:    end for
22:    return T
23: end function
```

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 sequentially starting from the root node

to a leaf node, we get different sequences belonging to the same record in the database.

- 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 constructing the *prefix tree* T from the database, CI encrypts the *prefix tree* using AES-CTR method 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 $\xi_k(IV||\ CTR_i), where\ i \geq 1$. 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*

After 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 $\{000,\ 011,\ 101,\ 110\}$. 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 values 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 `000` *after the decryption. Now, if the query is AGCCTGT (ϕ = 000101011011110101110), only the corresponding length of the decrypted value (length of* $000 = 3$*) will be considered as the query length. The hamming distance between the updated query (000) and decrypted value (000) 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'* (101011011110101110) *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 the database sequences. Our search algorithm takes r, q, k as input and returns the similar sequences (S) that satisfy $D_{hd}(q, s) \leq k$.

5.8 Performance Analysis

We have implemented our proposed technique for secure similar sequence search problem and assessed 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 separately.

Algorithm 4 Searching similar sequence in the tree

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

Output: Resulting similar sequences (S)

$a \leftarrow r.getChildren()$
while $a \neq \{\phi\}$ **do**
    $b \leftarrow a.pop()$
    if $b.getChildren() \neq \phi$ **then**
        Update query (q') up to length of $b.val$
        if $D_{\text{hd}}(q', b.val) <= k$ **then**
            $S \leftarrow S + b.val$
            Update $k' = k - D_{\text{hd}}(q', b.val)$
            Algorithm 4 (b, q', k')
        end if
    else if $b.getChildren() = \phi$ **and** $D_{\text{hd}}(q, b.val) \leq k$ **then**
        $S \leftarrow S + b.val$
    end if
end while
return S

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.

We have implemented the cryptography building blocks of garbled circuit and

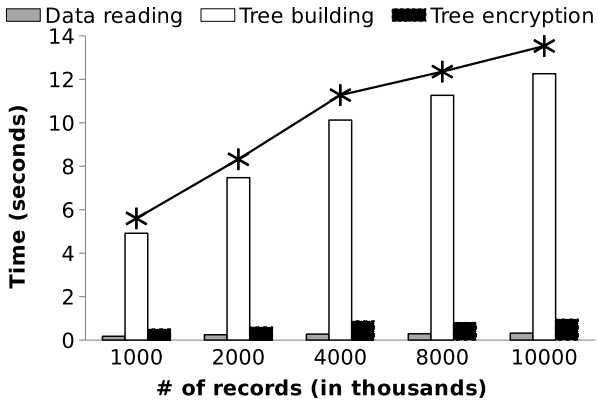
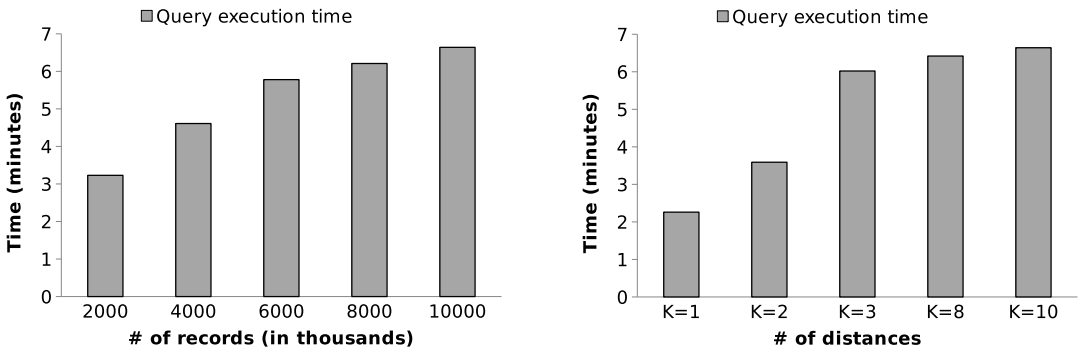


Figure 5.3: Data read and *prefix tree* building time.

AES in CTR mode described in Chapter 2. We investigated different garbled circuit libraries and used the *FlexSC* [73] 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 [124] 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 [124] 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 5.3 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 1K to 10K, where each record contains 500 nucleotides. As expected, the time increases linearly with the



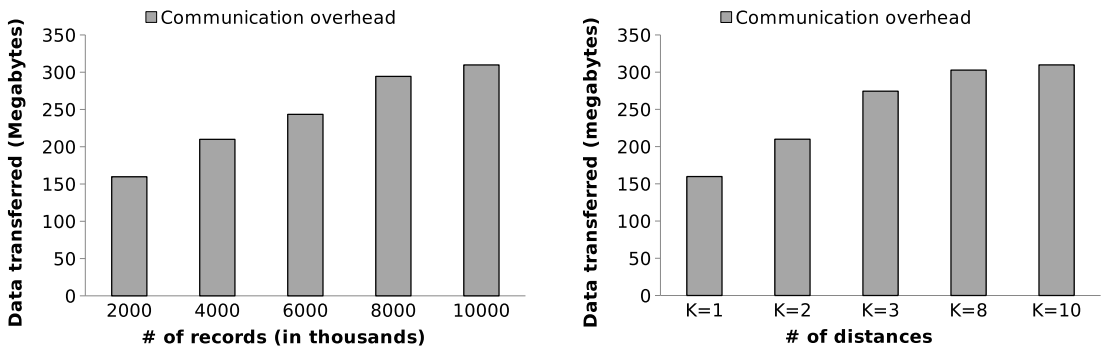
(a) (b)

Figure 5.4: Figure 5.4a shows the query execution time on different datasets with different number of records and a fixed hamming distance $k = 10$. Figure 5.4b shows the query execution time on a dataset of 10000 records and different hamming distances $k \in \{1, 2, 3, 8, 10\}$.

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 5.4 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 5.5 shows the amount of data transferred between the *researchers* and the *CS* during the evaluation of the garbled circuits.



(a) (b)

Figure 5.5: Figure 5.5a shows the communication overhead on different datasets with different number of records and a fixed hamming distance $k = 10$. Figure 5.4b 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 Discussions

The security of the system is compromised if *CS* can access any sequence. Moreover, 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 [9].

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 technique 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 compute Hamming distance. We have demonstrated that our model does not reveal any sensitive genomic data during the query execution phase as well as data processing phase.

Chapter 6

Secure Top-K Similar Patients Query Search

In this chapter, we provide a method to preserve the privacy of genomic data sharing in similar patients query using top-k query search. In the previous chapter, we proposed a method to find the similar sequence based on the value of k, where k represents a distance. However, in this chapter, we are proposing a method that retrieves all the similar sequences of a dataset in a descending order of similarity and presents only the top-k sequence(s). Our proposed method addresses the problem of data privacy, query privacy, and output privacy in a feasible time. It is noteworthy that prior contributions [1; 125; 92] did not consider outsourcing their datasets to the cloud. They conducted the required computation on plaintext genomic data and shared partial results with other parties according to defined system models to compute final results. In contrast to this, our proposed approach encrypts datasets before outsourcing to an untrusted cloud and then conducts the required computation

on encrypted data.

Our objective is to design an efficient framework for outsourcing the genomic data while securely computing SPQ. The proposed method depends on the hamming distance. However, we compared with edit distance since it is a popular similarity metric to compare the accuracy of similar patient queries [92; 1]. This metric allows health care professionals or researchers to fetch similar genomic sequences based on a query sequence. For example, when a new patient is admitted, the medical doctor may want to retrieve data from previous patients with similar genomic sequences. The history of the previous patients may help the medical doctor determine whether the patient has a predisposition to a specific disease [4].

6.1 System Design Overview

The architecture of our proposed system is the same as presented in Figure 4.2 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 compressed prefix 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.

6.2 Threat Model

CI is a *trusted* entity as it is responsible for the generation and encryption of prefix trees. It can authenticate the identity of the researchers who apply to access the data before sending them the keys. This role of authentication performed by *CI* can be considered similar to *Data Access Committee (DAC)* of NIH [117]. We consider that *CS* is *semi-honest* or *honest but curious* [118]. Thus, CS follows the protocol as defined and does not try to misinterpret any information about the contents. Nevertheless, it can gather any statistical or other information regarding input, output or the computation during/after the protocol execution. We assume that the *data owner*, *CI* and *CS* have no interest to behave maliciously in the wish of generating incorrect output. Our method is also designed based on the the following assumptions:

- We consider that the system works appropriately, that is *researchers* receive the correct keys from the *CI*.
- We consider that the *CS* does not collude with the *researchers* and *CI* does not collude with the *CS*. This is a necessary requirement to guarantee query privacy.

6.3 Genomic Data Representation

We consider a dataset of n genomic sequences where each sequence is comprised of $C_1, C_2, \ldots, C_m$ nucleotides. Here, each row represents an individual and the last column is utilized to store the phenotype of that particular individual. This phenotype can reveal any associated diseases and certain physical traits of that individual. In

Table 6.1: Sample genomic data representation where $C_i \in \{A, T, G, C\}$ are the different positions on the same sequence and the phenotype, *Cancer* $\in \{0, 1\}$

#	C_1	C_2	C_3	C_4	C_5	...	C_m	Cancer
1	A	G	C	C	T	...	C	1
2	C	A	C	C	C	...	G	0
3	A	G	C	T	C	...	G	1
2	C	A	C	C	C	...	G	0
				⋮				
n	A	G	T	C	T	...	C	0

Table 6.1, we show an example of such a dataset where the phenotype is defined as a diagnosis of a cancer.

6.4 Query Types

Given a reference query sequence, our objective is to securely execute a similar patient query. In particular, researchers want to retrieve the top-k sequence(s) based on the edit distance approximation from the aggregated database. For example, consider the following query is submitted by a researcher.

```
SELECT (*) FROM Sequences
WHERE s LIKE AGGTC ... C AND k = 1
```

Query 6.1: Example of a query

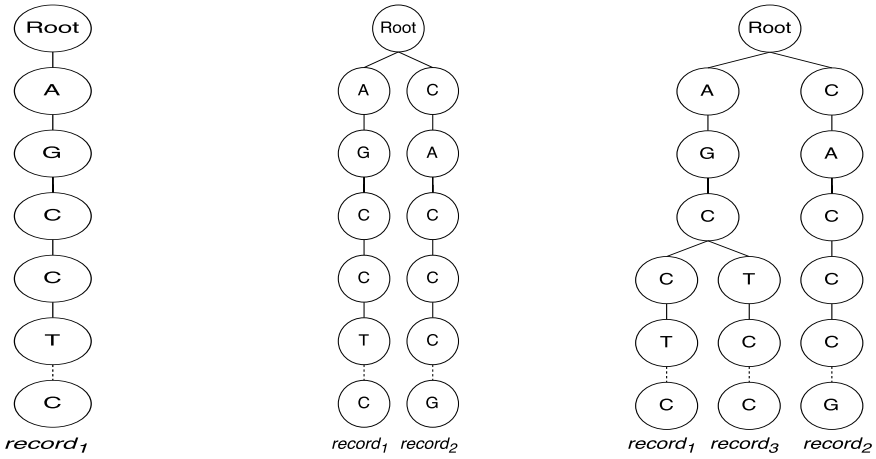
Table 6.2: Notations

Notation	Meaning
CI	Certified Institution who does the preprocessing
CS	Cloud Server where the data is stored in
D	Genomic dataset
n	number of records in the dataset D
D_i	i^{th} record of D
D_i^j	j^{th} nucleotide on i^{th} record of D
$\pi(D_i^j)$	create a new node using j^{th} nucleotide on i^{th} record of D
m_i	number of nucleotides in the record D_i
T	Constructed prefix tree
$\mathcal{E}_{CTR}$	AES encryption in Counter mode
$\tilde{T}$	Compressed prefix tree
$\mathcal{E}_{\tilde{T}}$	Encrypted compressed prefix tree

If we execute the above query (q) on Table 6.1, the researcher will end up with the record number 3 since #3 is the most similar sequence among the n records.

6.5 Methodology

In this section, we present the two main phases of the defined framework: data preprocessing and query execution. All the notations used are summarized in Table 6.2.



(a) (b) (c)

Figure 6.1: Different steps during the generation of the prefix tree from Table 6.1. Figure 6.1a, 6.1b, and 6.1c represents the tree after the insertion of the first, second, and third record, respectively.

6.5.1 Preprocessing

Before outsourcing the genomic data to the cloud, we do a one-time preprocessing to convert the data from a tabular format (Table 6.1) to a tree based representation. We detail the steps below.

Building a Prefix Tree

When CI receives the sequences from data owners, it constructs a prefix tree T with specific properties. As a human genomic sequence is comprised of only four nucleotides $\{A,\ C,\ G,\ T\}$, each letter can be represented as three bits. For example,

we can map {*A, C, G, T*} to {*000, 011, 101, 110*}, respectively. We depend on the hamming distance to approximate the edit distance. We used 3-bit encoding instead of a 2-bit encoding to encode A, T, G, C since we want 1 hamming distance between any two dissimilar nucleotides. If we use 2-bit encoding ($A = 00$, $C = 01$, $G = 10$, and $T = 11$), the hamming distance between A and T becomes 2. To solve this, we used 3-bit encoding ($A = 000$, $C = 011$, $G = 101$, and $T = 110$) so that the hamming distance between any two dissimilar nucleotides becomes 2. Therefore, we divide the final distance by 2 to achieve the actual hamming distance.

Algorithm 5 presents the pseudo-code for building the prefix tree. The steps are clarified in Fig. 6.1. The sequences, that have similar prefixes, share the same nodes. New branches will be created when dissimilarities appear. The cost of inserting a sequence is $\mathcal{O}(m)$ (m is the length of the sequence) as the algorithm performs constant-time sequential steps for each nucleotide of the sequence. Hence, the total runtime of building the prefic tree is $\mathcal{O}(mn)$, where n is the number of records in the database. The features of the prefix tree can be summarized as follows:

- If traversed sequentially from the root node to the leaf node, the levels on each branch represent different positions of a genomic sequence.
- The original genomic sequence can be reconstructed by recursively traversing from leaf node to the root node.
- Only a unique nucleotide value at a particular position creates a new branch and the following nucleotides are added as the children of that node.

Algorithm 5 Algorithm for building prefix tree

Input: Root node and the database (D)

Output: A prefix tree, T

```
1: for i ← 1 to n do
2:     InsertSequence(T.root, D_i);
3: end for
4: function InsertSequence(T.root, D_i)
5:     currNode ← T.root;
6:     l ← D_i.length                          ▷ length of the current sequence
7:     for j ← 1 to l do
8:         currNucleotide ← D_i^j;
9:         flag ← 0;
10:        for k ← 1 to currNode.numOfChildren do
11:            if currNode.getChild(k) == currNucleotide then
12:                currNode = currNode.getChild(k)
13:                flag ← 1;
14:            end if
15:            if flag == 0 then
16:                a = π(D_i^j)
17:                currNode.addChild(a)
18:                currNode = a
19:            end if
20:        end for
21:    end for
22:    return T
23: end function
```

Compression of the Prefix Tree

As each node in the prefix tree T represents one individual nucleotide, this tree will result in higher storage requirement for longer genomic sequences. Hence, T is compressed by aggregating different nodes sequentially. The aggregation of nodes is done on nodes which have only one child. For example, in Fig. 6.1c, the rightmost

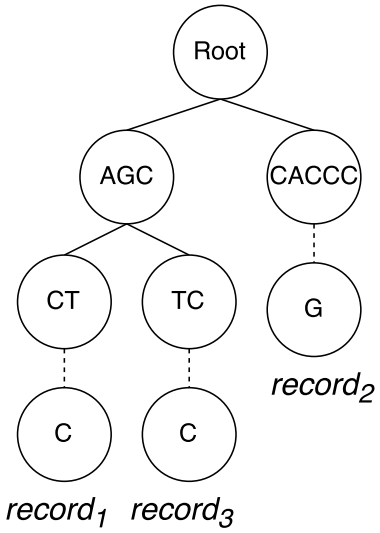


Figure 6.2: Compressed Prefix tree for 3 records from Table 6.1.

branch contains the sequence $C \rightarrow A \rightarrow C \rightarrow C \rightarrow C \ldots G$. Since each node in the path has only one child node, these nodes can be aggregated to one node $CACCC \ldots G$ (as shown in Fig. 6.2). This will reduce the storage requirement of the prefix tree T because we no longer need to store one node per nucleotide. Comparing the prefix tree size (19MB) with the compressed prefix tree size (968 KB) of the real data demonstrates the cost associated with each node in the prefix tree.

Encrypting the Compressed Prefix Tree

After building the compressed prefix tree $\tilde{T}$ from the database, CI encrypts $\tilde{T}$ with Advanced Encryption Standard (AES) in Counter Mode [77]. The encryption is performed on the individual node value. As AES is a symmetric encryption, the ciphertext $\mathcal{E}(A)$ will be the same for each encryption of the nucleotide value A. Hence, we use AES in Counter Mode (CTR) to achieve randomized encryption. In other words, the ciphertext $\mathcal{E}_{CTR}(A)$ will be different for each encryption of the nucleotide

value A. Finally, *CI* outsources the encrypted compressed prefix tree $\mathcal{E}_{\tilde{T}}$ to the cloud server *CS*, where the targeted query is executed. *CI* also shares the secret key with researchers.

6.5.2 Query Execution

The execution of a query q on the encrypted compressed tree $\mathcal{E}_{\tilde{T}}$ for approximating the edit distance is detailed in Algorithm 6. This algorithm is executed using a garbled circuit. We called the GC with the parameters: the encoded researcher query's sequence, decryption key, and root node. The researcher first encoded his/her query ($q \rightarrow \tilde{q}$) such that $A = 000$, $C = 011$, $G = 101$, and $T = 110$. The researcher then gave this encoded query $\tilde{q}$ and the secret key as an input for a garbled circuit for execution. On the other side, *CS* holds the preprocessed tree, $\mathcal{E}_{\tilde{T}}$. Initially *CS* makes a map d to store the values of the approximated edit distance for each record in $\mathcal{E}_{\tilde{T}}$. The main function *RecursiveQueryExec* of Algorithm 6 (line 6) starts with the root node of the tree $\mathcal{E}_{\tilde{T}}$. Since each node has a different number of nucleotides after compression, the length (l) of the data (*node.value*) is taken into account. The GC decrypts the *node.value* using the key provided by the researcher (line 9) and specifies the length of each node (l) (line 10). The GC uses both the *node.value* and a substring of $\tilde{q}$ from *pos* to $pos + l$ ($\tilde{q}_{(pos,l)}$), where *pos* is the current position of the query's sequence currently being evaluated (line 11). The position *pos* will be changing when we continue traversing the tree depending on the length of each node. The underlying function that is being computed by GC is a simple XOR that counts the number of ones between both substrings. This can also be seen as a

Algorithm 6 Algorithm for query execution

Input: Root node of the compressed prefix tree $\tilde{T}$, Encoded query $\tilde{q}$

Output: Approximated edit distances d of the n records from a query sequence $\tilde{q}$

1: **for** $i \leftarrow 1$ to n **do**
2: $d[i] \leftarrow int.max$ ▷ Assigning all records a fixed distance
3: **end for**
4: **for** each $cnode \in T.child$ **do**
5: $0 \leftarrow pos$
6: `RecursiveQueryExec(cnode,pos,int.max)`
7: **end for**
8: **function** $RecursiveQueryExec(node, pos, distance)$
9: `GC decrypts` $node.value$
10: `l` $\leftarrow$ `node.length` ▷ length of the node
11: `currDist` $\leftarrow$ `retrieves number of ones in`$(node.value \oplus \tilde{q}_{(pos,l)})$ ▷ $\tilde{q}_{(pos,l)}$= substring of query from pos to $pos + l$
12: `distance` $\leftarrow$ `distance - currDist`
13: `pos` $\leftarrow$ `pos+l`
14: **if** $node == leaf$ **then**
15: `d[node.recordId]` $\leftarrow$ `distance`
16: $return$
17: **end if**
18: **for** each $cnode \in node.child$ **do**
19: `RecursiveQueryExec(cnode,pos,distance)`
20: **end for**
21: **return** `SortDescennding(`d`)`
22: **end function**

hamming distance value between two substrings (line 11) with respect to the encoding mentioned before. The function *RecursiveQueryExec* has another variable (*distance*) as a third input. Its value is reduced by the value retrieved from the GC function (line 12). The child nodes are then traversed using the recursive function (line 19). The termination criteria for the recursion relies on the considered node being a leaf

node (line 14). For a leaf node, the distance is stored in the distance map d. However, if the length of the query and the data record does not match, the smaller length will be considered and the computation will be conducted accordingly.

Example 1: *Suppose the query is AGCCT...G. We are considering the compressed prefix tree (Fig. 6.2) as an encrypted tree stored in a cloud server. For simplicity, we are demonstrating the example by presenting the node values as nucleotides for better understanding (not as encoded data). The very first node of the first level of Fig. 6.2 contains AGC and accordingly node.value = AGC and* $l = 3$. *Since* $l = 3$, *so we take only the first three nucleotides from the query (AGC from AGCCT...G). The input for the GC will be the first node value from the CS (AGC) and the first 3 nucleotides from the query provided by the researcher. We calculate the approximated edit distance between the updated query and node.value using the GC by XORing the encoded values from both parties to count the number of ones between both substrings. We initially assign a fixed b distance to all records where* $b = int.max$. *After computing the approximated edit distance for each node, the value of b will be decremented by the number of ones. For the first node, b will remain int.max as the query substring and the node value are similar. After that, the updated query will be CT...G and the next pre-order node is CT. We take same* $l = 2$ *and the query substring (CT). Again b will not change as the approximated edit distance is* 0. *According to the pre-order traversal, we reach to a leaf-node with node.value = C, and we have G as the updated query. So the approximated edit distance will be 1. The value of b will be decremented by 1. As we are in a leaf node, we will store this result. Leaf node contains the sequence id (sequence* 1*). Thus we will store the result as* $<$ *(sequence); (distance)* $>$.

Similarly we will traverse the whole tree, and store each leaf node's updated edit distance. The Researcher will get the result (encrypted sequences) in a decremented order according to the b value ($b = int.max$ means the query and the sequence is exactly the same and the edit distance is zero). Finally, the researcher decrypts the sequences using the keys given by CI and gets the query result.

6.6 Security Analysis

The security of the system is compromised if CS gets idea about any sequence. Otherwise, researchers are only permitted to know their query result. Various scenarios of our proposed model are as follows.

- *Leakage to CI in each query.* there is no leakage to *CI* during the query execution phase since *CI* is not involved during the query execution. Its only task is to provide the secret key to researchers.

- *Leakage to researchers.* The leakage to researchers is the result of the query. We do not consider here any privacy leakage through the output.

- *Leakage to CS.* The *CS* can know the tree traversal pattern which means all the nodes in $\tilde{T}$ which are accessed during the query execution. The tree traversal pattern includes either the paths stopping at some internal nodes or reaching the leaves, depending on the result of the query. Since the nucleotide value of each node is encrypted, the tree traversal pattern does not leak any sequence to *CS*.

6.7 Experimental Results

We implemented our proposed method for secure similar patient query problem. To evaluate its performance, we used both real and synthetic datasets. We used two different machines to run the *CS* and the *CI*. Both of them were Intel Core i5 (3.3 GHz processors) and 8 GB memory, running Ubuntu Linux 16.04. We used JAVA programming language to write the source code.

We considered the following aspects to assess the efficiency of our proposed method:

1. *Data read and tree building time.* Time needed to process the genomic database and build the corresponding 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 a researcher.

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

5. *Storage analysis.* Amount of spaces required to represent the original data, unencrypted tree and the encrypted tree.

We implemented the cryptography building blocks: Yao's garbled circuit (Yao's protocol [10]) for secure computation of the edit distance and an encryption scheme with Block Ciphers called *Counter Mode* [77] to encrypt the data. We investigated various garbled circuit libraries and implemented the garbled circuits using the *FlexSC* [73] library.

Table 6.3: Description of the real-life dataset

Parameters	**Dataset**
Number of records (n)	50
Sequence length (l)	3400-3500
Number of queries	10
Query sequence length	3400-3500
Data size (KB)	734
Data source	iDash 2016 [126]

To evaluate our system, we used real-life and synthetic datasets. The real-life dataset is taken from the iDASH competition 2016 [126], where there are approximately 3400-4000 different SNPs from 50 different individuals. We generated various synthetic datasets varying the number of records (between 2000 to 10000) by randomly adding records to the dataset of iDASH competition 2016 [126] since the real-life dataset was not large enough to evaluate the scalability of our proposed model. We executed 10 runs for each experiment and averaged the result over the runs. Corresponding details about the dataset is presented in Table 6.3.

Tree Building and Encryption Time. We analyzed the time required for our method containing different number of records to determine the scalability of our system. Fig. 6.3 plots the time required for reading the data from the database, building the compressed prefix tree ($\tilde{T}$) using this data, and encrypting the compressed prefix tree. Here we vary the number of records from 2000 to 10000, where each record contains 3500 nucleotides. As expected, the operation time increases when we increase

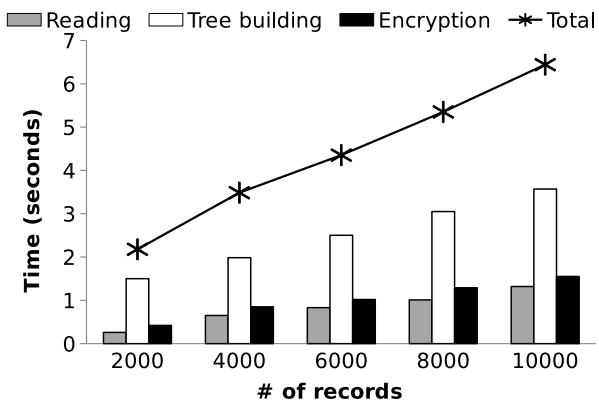


Figure 6.3: Tree building and encryption time.

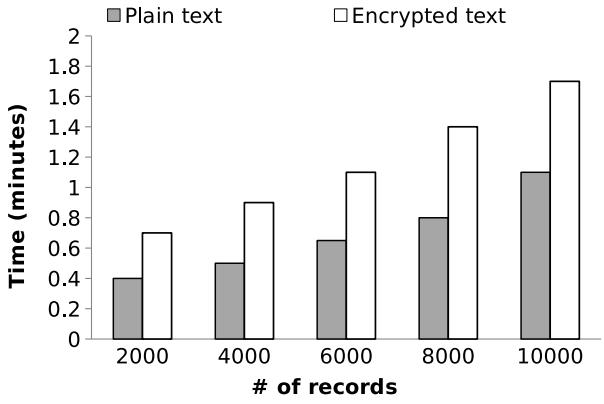


Figure 6.4: Query execution time

the number of records. We observe that, compressed prefix tree building takes significant time than the data read and tree encryption time. Moreover, the encryption time slightly changes, as shown in Fig. 6.3, because encryption time depends on the depth of the compressed prefix tree.

Query Execution Time. Fig. 6.4 shows the query execution time based on the number of records in the dataset (plaintext and encrypted text). The size of the dataset has an effect on the query execution time. As expected, the time increases

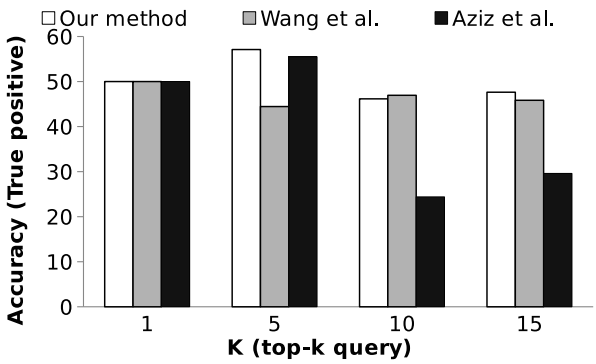


Figure 6.5: Accuracy for different top-k query using our method compared with of Wang *et al.* [1] and Aziz *et al.* [2].

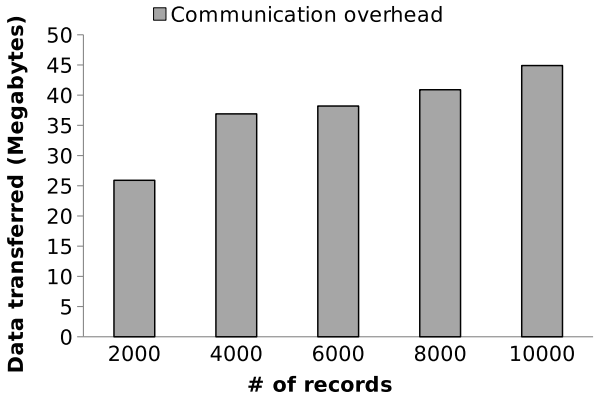


Figure 6.6: Communication overhead of our method.

linearly with the increase of the number of records. We also observe that secure computation requires more time than the plaintext computation.

Communication Overhead. Fig. 6.6 shows the amount of network communication required in megabytes between the researcher and *CS* during the evaluation of the garbled circuit. As expected, the overhead increases linearly with the increase of number of records.

Storage Analysis. Table 6.4 lists the amount of spaces required to represent

Table 6.4: Size of original database, unencrypted tree, and encrypted tree (in KB).

# of records	2000	4000	6000	8000	10000
Raw data	6900	13900	20600	27700	33500
Unencrypted tree	370	409	423	432	455
Encrypted tree	1100	1300	1352	1400	1465

the original data, unencrypted tree and the encrypted tree. As expected, unencypted prefix tree consumes less storage than the raw database.

Accuracy Analysis. We analyzed and compared our accuracy for Similar Patient Query (SPQ) with recent works [1; 2]. Here the accuracy is defined as true positive rate or sensitivity $(N_{TP}/(N_{TP} + N_{FN}))$ where TP, FN are true positives and false negatives, respectively. Fig. 6.5 represents the accuracy of SPQ for different k values (top-k) using our method along with the accuracy of Wang *et al.* [1] and Aziz *et al.* [2]. We can see that our results achieve similar accuracy in comparison with these related works (federated approach) that run on plaintext. We consider this comparison because it shows the strength of our approach. We are able to achieve similar accuracy (and sometimes better) while our algorithm runs on encrypted data. It will be more logical to compare our approach with Cheon et al.'s approach (outsourced approach) [48]. However, their proposed parameters, utilized for the homomorphic encryption scheme, need to be extended to fit our targeted data size (3500 nucleotides).

6.8 Summary

In this chapter, we domonstrated a secure and efficient method for similar patient searches on encrypted data. The proposed method constructed a compressed *prefix tree* from the aggregated genomic data and then outsourced it to a third party cloud server. We demonstrated that this model does not reveal any sensitive genomic data during the data processing nor during the query execution phase. In fact, this model preserved data privacy, query privacy, and output privacy. Additionally, this model achieved an efficient run time with high accuracy compared to the current state of the art. However, one of the limitation of the proposed model is considering equal-size sequences. This is a limitation as the original edit distance algorithm considers variable-length sequences.

Chapter 7

Secure Subtring and Set-maximal Search

The volume of genomic data is growing enormously due to the advancement of the current sequencing technologies. Storing these huge data to cloud could be a feasible solution if we can ensure the data security and privacy. Substring search on genomic data is an indispensable problem in sequence analysis. There are many real life applications of substring search such as ancestor search, identify similar patient, create personalized medicine etc. Besides genomic data, substring search is also important for pattern matching or word searching that applicable to search any keyword in text message, chat logs, documents etc.

Genomic data is more sensitive since it contain heredity information. For example, if a person's genomic data is leaked, we can easily predict his/her offspring's heredity information such as eye color. Thus, we have to preserve the security and privacy on the uploaded genomic data in the cloud. Genomic data breach of an individual

may reveal the susceptibility of certain disease that can effect the denial of the health insurance.

Our objective is to outsource genomic data and compute substring and set-maximal search on the outsourced data in a privacy-preserving way. In our proposed model, we utilize cloud to store the data because of the extensive volume of the genomic data. We provide *data privacy* on the outsourced cloud data. Therefore, the cloud cannot infer any information from the uploaded data. Moreover, researcher executes query on the cloud in a way that preserve *query privacy* and *output privacy*. Thus, adversary or cloud learn nothing about a researcher's query and the query output is only available to the researcher.

Privacy-preserving substring search and set-maximal search is a fundamental problem and many researchers proposed various solutions. Existing works can be divided into many sub-problems according to number of the search sequence, length of the query, starting position, exact match, maximum match etc. We compared our proposed method for substing search with Ishimaki *et al.*'s [101] work where researcher's query sequence length is smaller than the database sequence and researcher is interested in exact match from a particular position. We also compared our proposed method for set-maximal search with Shimizu *et al.*'s [3] work where researcher is interested in maximum match between query sequence and database sequence from a start position. We utilize *generalized suffix tree* data structure that stores the suffixes of a sequence. Ukkonen's proposed a linear time and space algorithm for suffix tree construction [127]. For multiple sequences, *generalized suffix tree* can store common substring.

Contributions. The contributions of this chapter are summarized as follows:

- We propose a privacy-preserving protocol to solve two problems based on genomic data: substring search query and set-maximal matches. We utilize a generalized suffix tree to create an index of the genomic data.
- We present different algorithms to execute secure substring search and secure set-maximal search on encrypted tree via garbled circuit [10]. Our proposed method preserves *data privacy*, *query privacy*, and *output privacy*.
- We implement our proposed method and evaluated secure substring search by comparing with Ishimaki *et al.* [101]. Experimental results shows that on a dataset of *2184* records (each containing *10000* SNPs) it requires approximately *2.3* seconds to execute secure substring search, while Ishimaki *et al.* [101] required *16* minutes.
- Moreover, we compare secure set-maximal search with Shimizu *et al.*'s [3] work. Experimental results shows that on a dataset of *2184* records (each containing *10000* SNPs) it requires approximately *2* seconds to execute secure set-maximal search, while Shimizu *et al.* [3] required *15.5* seconds.

In this section, we describe our proposed system model, kind of data we utilized, query types, encryption method, and threat model.

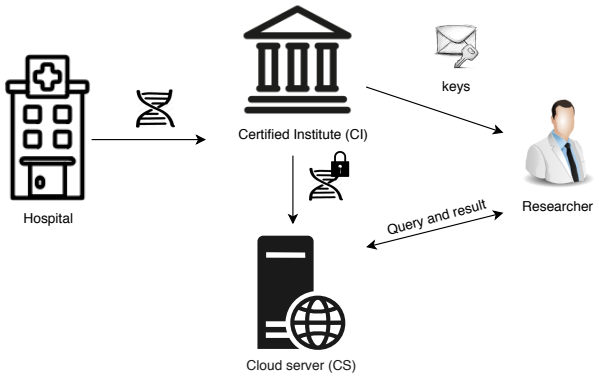


Figure 7.1: System architecture.

7.1 System Model

Figure 7.1 summarizes our proposed system model. Our proposed model has four entities: *Hospital*, *Certified Institute* (CI), *Cloud Server* (CS), and *researchers*. The hospital entity is responsible for possessing and managing single-nucleotide polymorphism data. Though we name the entity as *hospital*, it could be any organization that possess or collects genomic data. The second entity in our protocol is the *CI* who collects data from the *hospital* and builds the generalized suffix tree. It also encrypts the tree, sends the encrypted tree to the *CS*, and manages the keys necessary for encryption and decryption. Moreover, *CI* sends the necessary secret key for decryption to the *researchers*. Here, we could combine these two entities (*hospital* and *CI*) into one entity (*hospital*) by granting *CI*'s responsibilities (building and encrypting the tree) to the *hospital*. *CS* stores the encrypted tree and communicates with the *researcher* while executing query.

Table 7.1: Sample haplotype SNP data representation.

#	SNP_1	SNP_2	SNP_3	SNP_4	SNP_5	SNP_6
1	1	0	0	0	1	0
2	1	1	1	0	1	0
3	1	1	0	0	0	1
4	0	1	0	1	1	0
			⋮			
n	0	1	0	1	0	1

7.2 Data Representation

Human genome can only differ $\approx 0.1\%$ of the positions between two individuals. This location difference in genomes is called single-nucleotide polymorphism (SNP). In this chapter, we consider a database of n human haplotype SNP sequences. The SNP data was simulated using Markovian Coalescent Simulator as collected from iDASH competition 2018 [96]. Here, SNPs are bi-allelic either 0 or 1. The most frequent value is known as the major allele, and the less frequent value is called the minor allele. Major alleles are coded as 0 whereas minor alleles are coded as 1. Notably, we only have the binary representation for simplicity as our proposed method can work on any dataset with fixed character set. Table 7.1 shows an example of the dataset.

7.3 Query Types

In this study, we present two different string queries resulting in similar records performed against a genomic dataset. Let $\mathcal{D}$ be a database of genotypes, $\mathcal{D}_j = d_1, d_2, ..., d_m$ where $\mathcal{D}_j$ represents a particular record in the dataset ($1 \leq j \leq \|D\|$) and d_i represents the vector of genotypes for i^{th} genome for a vector of SNPs, where SNPs are sorted with respect to their locations on the genome. We address two different problems, as described below:

***Definition 1 (Substring search)**: Given dataset $\mathcal{D}$, a query sequence* $\mathbf{q}$*, and a search start position* $\mathbf{s}$*, there might be record (s) $\mathcal{D}_j$ (zero or more) such that:*

$$q = D_j[d_s, d_{s+\mathsf{q}-1}] \text{ where } 1 \leq j \leq \|D\|$$

Example 3.3.1. Consider *researcher's* submitted query, $q = 010$ with a start position, $s = 4$. If we execute this query q on Table 7.1, the researcher will get the sequence $1: 1\ 0\ 0\ \mathbf{0\ 1\ 0}$ and sequence $2: 1\ 1\ 1\ \mathbf{0\ 1\ 0}$ since they matches the substring 010 from position 4. The details of substring search process are described in section 7.5.3.

***Definition 2 (Set-maximal search)**: Given $\mathcal{D}$,* $\mathbf{q}$*, and* $\mathbf{s}$ *as mentioned above, then there exist* $\mathsf{j}\prime$ *(zero or more) in $\mathcal{D}$ such that:*

1. $q[i_1, i_{max}] = D'_j[d_s, d_s + d_{max} - 1]$
2. $q[i_{max} + 1] \neq D'_j[d_s + d_{max}]$
3. $1 \leq j' \leq \|D\|$

Example 3.3.2. Consider *researcher's* submitted query, $q = 00011$ with a start position $s = 2$. If we execute this query q on Table 7.1, the researcher will get the *0001* from sequence *1 : 1* ***0 0 0 1*** *0* containing *4* matches. Therefore, $q[i_1] = 1$ and $q[i_{max}] = 4$. The details of set-maximal search process are described in section 7.5.3.

7.4 Threat Model

In this section, we introduce our proposed privacy model, assumptions, and security goals.

7.4.1 Privacy Model:

In this chapter, we define our three privacy models as following:

- ***Data Privacy:*** The data stored in the *CS* should be encrypted. If the *CS* gets compromised, the attacker should not learn anything about the data stored in the cloud.
- ***Query Privacy:*** The *hospital* (institutions contributing the data), the *CS* or an adversary should learn nothing about a query executed by *researchers*.
- ***Output Privacy:*** The result of the query should not be disclosed to anybody except the *researchers* who initiated the query.

7.4.2 Assumptions:

We rely on the following four assumptions for our proposed system:

- We assume the *CS* to be a semi-honest entity (also known as honest-but-curious) [128] where it follows the protocol but may attempt to derive additional information while executing *researcher's* query.

- We consider *CI* is to be a trusted entity since it is responsible to create the suffix tree and share necessary keys to the *researcher* to decrypt the encrypted result.

- We consider neither the *hospital*, *CI* nor the *CS* behaves maliciously to generate incorrect output.

- Moreover, we consider that the *CS* does not collude with *researchers* and *CI*, and *researchers* does not collude with the *CS*.

7.4.3 Goals:

In our proposed model, we wish to achieve the following goals:

- ***Goal 1:*** Our objective is to ensure the confidentiality of the genomic data so that *CS* does not learn anything about the outsourced data.

- ***Goal 2:*** Our second objective is that both *CI* and *CS* learn nothing about the *researcher* query. *CI* also create and store a profile for each *researcher* and authenticate before sending keys to the *researcher*.

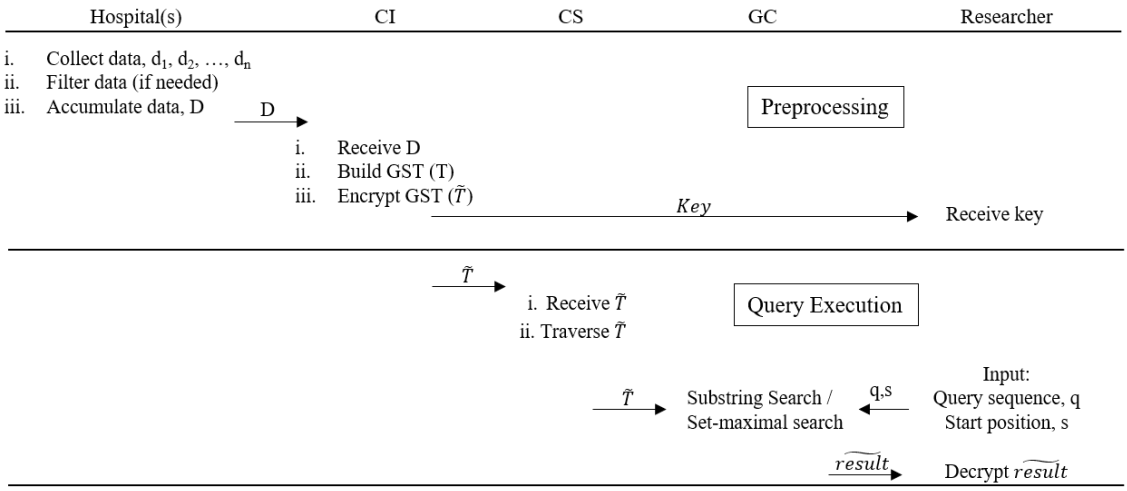


Figure 7.2: The overall protocol of our proposed solution.

7.5 Methods

In this section, we present our proposed method of secure substring/set-maximal search. We divide the proposed method into three subprocesses: 1) Generalized suffix tree building, 2) Tree encryption, and 3) Secure search on the encrypted tree. We require the same preprocessing (tree building and encryption) for both substring and set-maximal search. We divide the secure search into two subsections to address the secure substring search and set-maximal search. We summarize the notations in Table 7.2 and Figure 7.2 represents the overall protocol of our proposed solution.

7.5.1 Generalized Suffix Tree (GST) Building

Let $\mathcal{S}$ be a sequence over an alphabet $\sum \in \{0, 1\}$, $\|\mathcal{S}\|$ the length of $\mathcal{S}$, and $\mathcal{S}_i$ with $i \in [1 : \|\mathcal{S}\|]$ the suffix of $\mathcal{S}$ starting at position i. The suffix tree $\mathcal{T}$ of a sequence $\mathcal{S}$ is a rooted tree whose edges are labeled with substrings of $\mathcal{S}$. The properties of

Table 7.2: Notations

Notation	**Meaning**
R	Root node of GST
q	Query sequence
pos	Query position
$matched_{seq}$	Matched sequence
enc_{pos}	Encrypted suffix position
$eqGC$	Equality check by GC
$cNode$	Child node
enc_{label}	Encrypted label sequence
$\|\|label\|\|$	Length of edge label
$\|\|q\|\|$	Length of query string

the suffix tree are the followings:

- The *path label* of a node v is the addition of all edge label from the root to v.
- Each substring $\mathcal{S}'$ of $\mathcal{S}$ contains only one subtree $\mathcal{T}'$ of $\mathcal{T}$.
- Every node of the suffix tree is either a leaf node or has two child nodes at most starting with either *0* or *1*.
- The labels on the edges can have any length greater than zero and no two edges going out from the same node can start with the same character. Therefore, a given (startNode, suffixString) pair can denote a unique path within the tree.

- Another aspect of the suffix tree is *suffix link* [127] that helps linear time suffix tree construction. A suffix link is a link, from node v with path label αw, to node v' with path label w, where $\alpha \in \sum$ and $w \in \sum^{+}$.

- Every inner vertex has a suffix link. Moreover, Ukkonen utilizes *edge-label compression* and *skip/count speed up* which are described in [129].

Example 4: Figure 7.3 presents a suffix tree using the sequence 1 of the table 7.1. If we traverse the nodes in Figure 7.3, we achieve *0, 10, 010, 0010, 00010, 100010*, which are actually the suffixes of the sequence *100010*.

The *generalized suffix tree (GST)* presents multiple sequences $\mathcal{S}_1, \ldots, \mathcal{S}_n$ over $\sum \in \{0, 1\}$. For this, we can build a suffix tree for $\mathcal{S}_1$ and then extending the tree by incorporating the sequences $\mathcal{S}_2, \ldots, \mathcal{S}_n$.

The naive implementation for constructing a suffix tree requires $\mathcal{O}(n^2)$ time complexity, where n is the sequence length [129]. However, Esko Ukkonen [127] reduced this to $\mathcal{O}(n)$ (linear) time using suffix link. The Ukkonen algorithm [127] proceeded forward from the first character to the last one from the longest to the shortest suffix. However, Ukkonen's suffix tree operates on a single sequence only. Since our dataset contain multiple sequences (Table 7.1), we construct the Generalized Suffix Tree (*GST*) exampled below:

Example 5: Figure 7.4 presents a GST using the sequence *1* and *2* of the table 7.1. It shows the SNP characters in each edge label. Each filled circle (vertex) in the Fig. 7.4 represents suffix(es) including the suffix position and the sequence number that contains the suffix(ex). For example, if we traverse up to node *5*, we get the suffix *010* and node attributes $[1;\ 4, 2;\ 4]$ which represents both sequence *1* and *2*

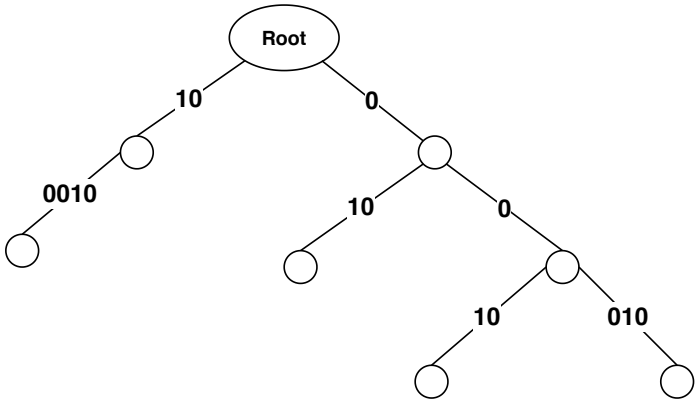


Figure 7.3: Suffix tree for a single sequence, $S = 100010$.

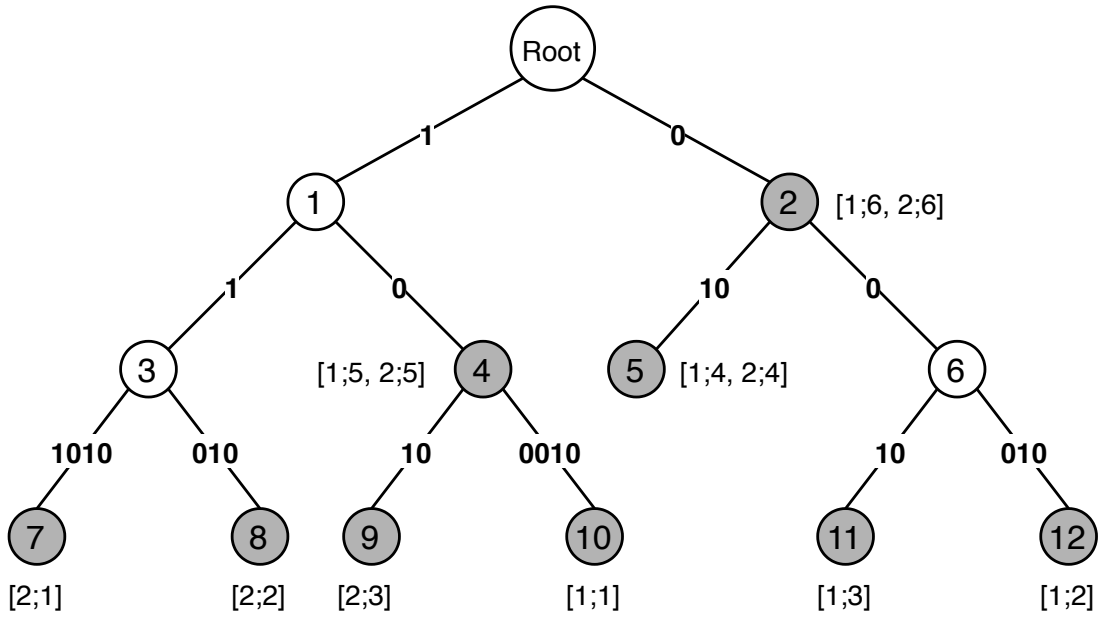


Figure 7.4: Generalized Suffix Tree with suffix positions.

have the suffix *010* starting from position *4*.

7.5.2 Tree Encryption

CI utilizes *AES-CTR* [77] to encrypt the contents in the GST to ensure the *data privacy*. The detailed process of GST encryption are as following:

- ***Key Generation:*** *CI* generates the key ($\mathcal{ENC}_k$), Initialization Vector ($\mathcal{IV}$) and a counter ($\mathcal{CTR}$) to encrypt each edge label and node attributes (if any) of the GST.

- ***Encryption:*** At first, *CI* builds a *keystream* by $\mathcal{ENC}_k(\mathcal{IV} \mid\mid \mathcal{CTR}_i), where\ i \geq 1$. Then it encrypts each edge label and node attribute (suffix position) by performing XOR operation between the edge label/suffix position and *keystream*. We applied two different encryption settings: one with only GST label encryption, and another with both the GST label and the suffix position encryption. We denote the encrypted GST as $\tilde{\mathcal{GST}}$.

- ***Share:*** *CI* sends the $\tilde{\mathcal{GST}}$ to *CS*. It also sends the keystream to the *researcher*.

7.5.3 Secure Search on the Encrypted Tree

Once the *researchers* send his/her query, search process starts with the $\tilde{\mathcal{GST}}$ stored in the *CS*. In our proposed method, to preserve the *query privacy*, we send the *researcher's* query and start position as an input to *GC*. We describe the detailed procedure of secure substring and set-maximal search below:

Algorithm 7 Secure Substring search on GST

Input: `R`, `q`, `pos`

Output: $matched_{seq}$

function SECURESEARCH(R, q, pos)
  $node_{matched} \leftarrow$ *Algorithm 8*(q, R)
  if $node_{matched}$! = $Null$ **then**
    $enc_{pos} \leftarrow retrieve\ suffix\ position$;
    $flag \leftarrow eqGC(enc_{pos}, pos)$
    if $flag == True$ **then**
      add matched sequence to $matched_{seq}$;
    end if
    for $cNode \in node_{matched}.child$ **do**
      $getLeafInfo(node_{matched}, pos)$
    end for
  end if
end function
function GETLEAFINFO($node_{matched}, pos$)
  $enc_{pos} \leftarrow retrieve\ suffix\ position$;
  $flag \leftarrow eqGC(enc_{pos}, pos)$
  if $flag == True$ **then**
    add matched sequence to $matched_{seq}$;
  end if
  for $cNode \in node_{matched}.child$ **do**
    $getLeafInfo(node_{matched}, pos)$
  end for
end function
return $matched_{seq}$

Secure Substring Search

We describe the secure substring search process according to the example from Section 7.3 where *researchers* send his query sequence *010* and position *4* to the *CS*. Algorithm 7 describes the secure substring search process. Initially, Algorithm 8

finds the particular node that matches the query sequence and the encrypted $\mathcal{G}\tilde{\mathcal{S}}\mathcal{T}$. In particular, while traversing the $\mathcal{G}\tilde{\mathcal{S}}\mathcal{T}$, GC execute an equality test (line no. 7, 10 in Algorithm 8) with the common sub-sequence between the query and the encrypted tree label comparing their minimum length. The detailed procedure is described in the following example:

Example 6: *Researchers* send the query sequence $q = 010$ and start position $pos = 4$ to *GC*. At first, we find a matched node between the query sequence and the suffix sequences in the $\mathcal{G}\tilde{\mathcal{S}}\mathcal{T}$ by Algorithm 8. Algorithm 8 starts with finding the edge that matches with the first character of the query and the first character of the encrypted label (line no. 3). This equality checking is performed by GC. Since the SNPs data is bi-allelic (either 0 or 1), in worst case two *GC* equality checking is necessary to find the matching edge. Moreover, each encrypted edge can have different length of SNPs. Therefore, line no. 6, 9 of Algorithm 8 finds equal length of query sequence and encrypted edge label. The *GC* decrypts the encrypted edge label and perform an equality test with the corresponding query sequence (line no. 7, 10 in Algorithm 8). The algorithm proceed to the next node if the query and tree label matches (line no. 12). Therefore, for the query sequence $q = 010$, the algorithm 8 will match encrypted edge label *0, 10* and return node #5. For simplicity, we demonstrate the example by presenting the edge label and node attributes as plaintext for better understanding. After matching the query sequence, we need to check the start position of the query and suffix position of the matched node as well. This equality checking is also performed via the secure protocol *GC* (Algorithm 7, line no. 5). We observe that

matched node #5 contains the attributes [1; 4, 2; 4] which matches the query start position *pos =4*. *CS* sends the encrypted matched sequence to the *researchers* and *researchers* obtains the desired result by decrypting the encrypted value. Therefore, it preserves the *output privacy* since no one (except CI) else can decrypt the result(s) due to unavailability of the key.

The Algorithm 7 also checks the suffix positions of every child node of the matched node since the child nodes could also match the query sequence and given positions (*getLeafInfo*, lines 10-16). For example, in Figure 7.4 for a query sequence *q =10* with start position *1*, node #4 will be returned by Algorithm 8. While checking the attributes of node #4 (1; 5, 2; 5), Algorithm 7, line no. 5 will not match the query start position *pos =1*. However, node #4 has child node #10 which matches the suffix position 1 with the query start position *pos =1*.

Secure Set-maximal Search

We demonstrate the secure set-maximal search process from the example in Section 7.3 where *researchers* send his/her query *q =00011* and *pos =2* to the *CS*. Again, to preserve the *query privacy*, we send the *researcher's* query sequence and position as an input of the *GC*. Algorithm 9 describes the secure set-maximal search process. The difference between the secure substring search and set-maximal search is that in the set-maximal search, GC checks the number of common prefix between the query sequence and the encrypted edge label instead of the equality testing. The detailed procedure is described in the following example:

Example 7: *Researchers* send the query sequence q =*00011* and start position *pos* =*2* to *GC*. Similar to the previous example, we find a matched node between the query sequence and the suffix sequences in the $\mathcal{GS\tilde{T}}$ by Algorithm 10. Again, Algorithm 10 starts with finding the edge that matches with the first character of the query and the first character of the encrypted label by GC (line no. 4). Moreover, line no. 8, 11 of Algorithm 10 finds equal length of query sequence and encrypted edge label. The *GC* decrypts the encrypted edge label and achieve number of common prefix with the corresponding query sequence (line no. 9, 12 of Algorithm 10). The algorithm proceed to the next node if the query and tree label matches (line no. 14). Therefore, for the query sequence q =*00011*, the algorithm 10 will traverse nodes: *2, 6, 12* and return node #12 as the matched node and *4 (1+1+2)* as the match count. For simplicity, we demonstrate the example by presenting the edge label and node attributes as plaintext for better understanding. Also, we need to check the start position of the query and suffix position of the matched node by GC as well (line no. 5, Algorithm 9). We observe that matched node #12 contains the attributes [1; 2] which matches the query start position *pos* =*2*.

While traversing $\mathcal{GS\tilde{T}}$, number of common prefix between a particular query and encrypted tree label actually determines the maximum matches between them (line no. 9, 12 of Algorithm 10: noOfMatch). Since the description of the *getLeafInfo* is similar to the Algorithm 7, we do not describe it in Algorithm 9. Moreover, we do not require to execute GC equality test in line no. 5 in both Algorithm 7 and 9 if we keep the suffix positions unencrypted in secure substring search and set-maximal

Algorithm 8 Match node (substring search) on GST

Input: `q`, `node`

Output: $matched_{node}$

for $i \leftarrow 1$ to $\|q\|$ **do**
  $q \leftarrow q_i$ *to* q_{end}
  $enc_{label} \leftarrow$ *get label of* $eqGC(q[i], enc_{label}[1])$;
  $lenToMatch \leftarrow min(\|q\|, \|label\|)$;
  if $\|label\| > \|q\|$ **then**
    $enc_{label} \leftarrow enc_{label}.substring(0, len_q)$;
    $flag \leftarrow eqGC(q, enc_{label})$;
  else
    $q\prime \leftarrow q.substring(i, len_{label})$;
    $flag \leftarrow eqGC(q\prime, enc_{label})$;
  end if
  if $flag == True$ **then**
    if $\|label\| >= \|q\|$ **then**
      return $edge.Node()$;
    else
      $node = edge.Node()$;
      $i += lenToMatch - 1$;
    end if
  end if
end for

search.

Complexity Analysis

Algorithm 8 and 10 represents search complexity of the secure substring and set-maximal search procedure, respectively. In both algorithms, the worst case time complexity of searching a query sequence depends on $\mathcal{O}(\|q\|)$, where $\|q\|$ is the length of the query sequence. Since we are traversing the $\|q\|$ characters using GC on the

Algorithm 9 Secure set-maximal search on GST

Input: `R`, `q`, `pos`

Output: $matched_{maxseq}, match_{count}$

function SECURESEARCH(R, q, pos)
  $node_{matched}, match_{count} \leftarrow Algorithm\ 10(q, R)$;
  if $node_{matched}\ != Null$ **then**
    $enc_{pos} \leftarrow retrieve\ suffix\ position$;
    $flag \leftarrow eqGC(enc_{pos}, pos)$;
    if $flag == True$ **then**
      $add\ matched\ sequence\ to\ matched_{maxseq}$;
    end if
    for $cNode \in node_{matched}.child$ **do**
      $getLeafInfo(node_{matched}, pos)$
    end for
  end if
end function
return $matched_{maxseq}, match_{count}$

$\mathcal{GS\tilde{S}T}$, it will result in $\mathcal{O}(\|q\|)$. This is also experimentally presented in Section 7.6 on Figure 7.6 (data encrypted only). Moreover, if we encrypt both the tree label and the suffix position of each node, then the worst case time complexity of searching a query sequence depends on $\mathcal{O}(\|q\| * t)$, where $\|q\|$ is the length of the query sequence and t is the number of records in the dataset that contains the matched sequence.

7.6 Experimental Results

We implemented and evaluated our proposed method of secure substring and set-maximal search. We utilized a synthetic database from the iDASH competition 2018 [96], where *1000* individuals records were given. We ran *CS* and the *CI* in two dif-

Algorithm 10 Match node (set-maximal search) on GST

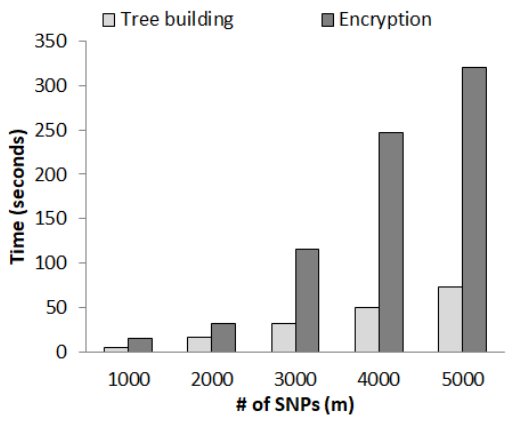
Input: q, node

Output: $matched_{node}, match_{count}$

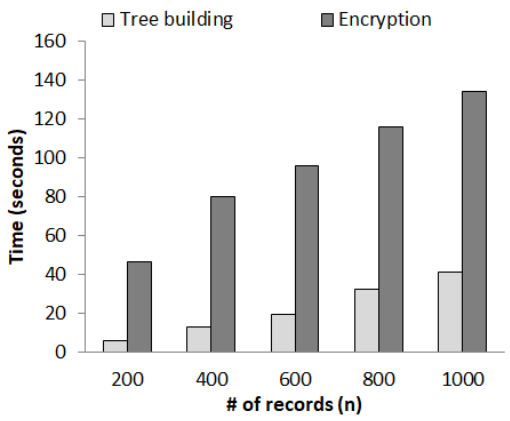
1: $match_{count} \leftarrow null$;
2: **for** $i \leftarrow 1$ to $\|q\|$ **do**
3: $q \leftarrow q_i$ *to* q_{end}
4: $enc_{label} \leftarrow$ *get label of* $eqGC(q[i], enc_{label}[1])$;
5: $lenToMatch \leftarrow min(\|q\|, \|label\|)$;
6: $noOfMatch \leftarrow 0$;
7: **if** $\|label\| > \|q\|$ **then**
8: $enc_{label} \leftarrow enc_{label}.substring(0, len_q)$;
9: $noOfMatch \leftarrow matchPrefix(q, enc_{label})$;
10: **else**
11: $q\prime \leftarrow q.substring(i, len_{label})$;
12: $noOfMatch \leftarrow matchPrefix(q\prime, enc_{label})$;
13: **end if**
14: **if** $noOfMatch > 0$ **then**
15: $match_{count} += noOfMatch$;
16: **if** $\|label\| >= \|q\|$ **then**
17: **return** $edge.Node(), match_{count}$;
18: **else**
19: $node = edge.Node()$;
20: $i += lenToMatch - 1$;
21: **end if**
22: **end if**
23: **end for**

ferent machines. Both of the machine's configuration were Intel Core i7 (*3.41* GHz processors), and *16* GB memory, running Windows 10. We implemented the garbled circuits using the *FlexSC* [73] library.

There are four parameters that we experimented with: m, the SNP size of the



(a) 1000 records with different SNPs (1000-5000).



(b) 3000 SNPs with different number of records (200-1000).

Figure 7.5: Tree building and encryption time.

Table 7.3: Query Execution Time (QET) and Communication Overhead (CO) for substring search and set-maximal search on 1000 records with different SNPs, m (1000-5000).

Query type	Substring search				Set-maximal search			
Encryption	Data		Data & Suffix position		Data		Data & Suffix position	
$n * m$/evaluation criteria	QET(sec)	CO(KB)	QET(sec)	CO(KB)	QET(sec)	CO(KB)	QET(sec)	CO(KB)
1000*1000	2.2	64	21.3	68	1.4	40	6.8	84
1000*2000	2.3	64	23.2	77	1.7	48	6.7	84
1000*3000	2.4	64	25.6	82	1.6	47	7.1	87
1000*4000	2.2	64	24.1	78	1.5	41	6.9	87
1000*5000	2.1	64	22.8	77	1.8	50	6.7	84

genomic sequence, n, the total number of records in the database $\mathcal{D}$, s, the query start position, and l, the length of the query. Moreover, we execute our experiments according to two different encryption settings: only GST label encryption, and both

Table 7.4: QET and CO for substring search and set-maximal search on 3000 SNPs with different number of records, n (200-1000).

Query type	Substring search				Set-maximal search			
Encryption	Data		Data & Suffix position		Data		Data & Suffix position	
$n * m$/evaluation criteria	QET(sec)	CO(KB)	QET(sec)	CO(KB)	QET(sec)	CO(KB)	QET(sec)	CO(KB)
200*3000	2.28	6	7.4	22	1.08	3.7	2.8	8
400*3000	2.05	5.46	11.7	37	1.07	3.7	3.4	11
600*3000	2.03	5.46	16.1	51	1.3	3.7	4.8	16
800*3000	2.25	6.3	22.3	71	1.14	4.01	5.7	19
1000*3000	2.4	6.4	25.6	82	1.6	4.7	7.1	23

the GST label and the suffix position encryption. These encryption settings have direct impact on the data privacy. If we only encrypt the GST label, then adversary can predict the matched sequences by tree traversal pattern and suffix positions.

Figure 7.5a shows the GST building and encryption time with both the edge label and the suffix position encrypted. We notice that the data preprocessing time (tree building and encryption) increases linearly with the increase of the m since increasing m actually increases the total number of suffixes. On the other hand, Fig 7.5b shows the GST building and encryption time for different number of records n={*200, 400, 600, 800, 1000*} where we fix the total number of SNPs to *3000*. As expected, preprocessing time increases linearly with the increases of n. Moreover, we observe that, tree building and encryption time is also linear if we only encrypt the edge label. However, it requires fewer time than encrypting both the edge label and the suffix position.

Table 7.3 represents the query execution time and the communication overhead

(fixed n and variable m). In both experiments (substring/set-maximal search), we execute five different values of m={*1000, 2000, 3000, 4000, 5000*}, l=*25*, and n=*1000*. We note that, we achieve less query execution time and communication overhead in set-maximal search compared to the substring search. This is because, query execution time is dependent on the total number of matches.

In substring search, we are interested in results with the exact match. On the other hand, in set-maximal search, we are interested in maximum matches between the query and the encrypted tree labels. Appearance of early mismatch between the query and tree labels causes into limited query execution time. Therefore, it takes less processing for the set-maximal search compared to the substring search. Moreover, we observe that increasing m neither affect the query execution time nor the communication overhead. The reason is query execution time and communication overhead is independent of the m. Also, as expected, query execution time and communication overhead increases if we encrypt both the data (tree edge label) and suffix position.

On the other hand, table 7.4 presents the query execution time and communication overhead for different *number of records* n={*200, 400, 600, 800, 1000*}, l=*25* with fixed m=*3000*. We observe that both query execution time and communication overhead increases linearly with the increases of n since the total match count increases if we increase the n.

Moreover, we note that the query execution time and communication overhead are almost similar for both substring and set-maximal search if only the data is encrypted. Query execution time and communication overhead increase linearly for

both substring and set-maximal search if data and suffix position both are encrypted. This is because, increasing n causes more matches along with the suffix positions. If the suffix position is encrypted, GC equality testing (line no. 5 in Algorithm 7 and 9) consumes extra query processing time. On the other hand, unencrypted suffix position checking requires almost same query processing time with the increment of n according to the table 7.4.

Figure 7.6 shows the substring search query execution time for *different query length*, l=$\{$*50, 100, 150, 200, 250*$\}$ with a fixed query start position (s=*1*), n=*1000* records, and m=*3000* SNPs. We also execute this experiment according to two different encryption settings: only GST label (data) encryption, and both the data and the suffix position encryption. While both the data and the suffix position are encrypted, we report that different query length with a fixed query start position affects the query execution time.

We notice that the query execution time decreases with the increase of the l because the total match count decreases if we increase the l. For example, it requires less time to find a match with query sequence length 50 rather than query sequence length 150. However, the matched node with query sequence length 50 contains more suffix positions to check (line no. 5 of Algorithm 7 and 9) than the matched node with query sequence length 150. On the other hand, if we execute the same queries while only the data is encrypted, query execution time increases linearly with the increment of the query length. Moreover, set-maximal search query execution time is independent of the query length and dependent on the number of matches (Table 7.3 and 7.4). For example, even though we execute a secure set-maximal search

with query sequence length 300, it could require very small query execution time if mismatch occurs at the beginning of the sequence.

7.6.1 Improvement Over Prior Approaches

We evaluate our proposed method by comparing with Ishimaki *et al.* [101] (substring search) and Shimizu *et al.* [3] (set-maximal search). Table 7.5 shows those comparisons. To solve the secure substring search, Ishimaki *et al.* [101] requires 16 minutes to process a secure query consisting of 10 SNPs. In contrast, our proposed method costs approximately 2.3 seconds to process the same query. Therefore, to process a secure substring search query, our method is at least *400* times faster than Ishimaki *et al.* [101]. Shimizu *et al.* [3] required around 4 seconds to solve the secure set-maximal search processing a secure query consisting of 25 SNPs. In contrast, our proposed method costs approximately 2 seconds to process the same query. Notably, the implementation was not available for Ishimaki *et al.* [101] compared to Shimizu *et al.* [3]'s work as we could not benchmark it on the same machine. However, Ishimaki *et al.* employed a far superior computational server to perform their experiments in comparison to ours.

7.7 Discussion

In this section, we explain the pros, cons, and future work of our proposed model.

Table 7.5: Comparison of secure substring/set-maximal search execution time on 2184 records, each containing 10000 SNPs [3].

Problem	Comparison	Query execution time
Secure Substring Search	Ishimaki *et al.* [101]	16 minutes
	Our work	2.3 seconds
Secure Set-maximal Search	Shimizu *et al.* [3]	3.97 seconds
	Our work	2 seconds

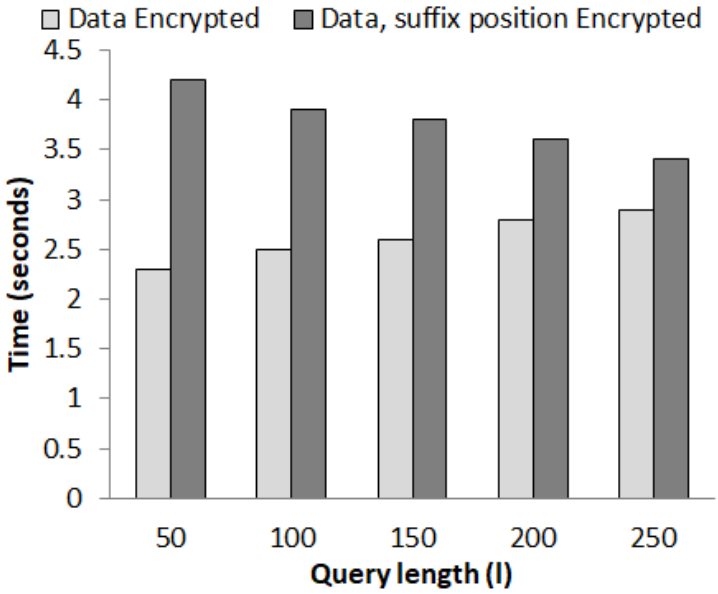


Figure 7.6: Substring search query execution time for different query lengths with 1000 records, each containing 3000 SNPs.

7.7.1 Advantages of Our Approach

Generalized Suffix Tree

In our proposed method, *CI* builds a generalized suffix tree from the accumulated

data submitted by the *Hospital*. This is a preprocessing step in our proposed method where we store all suffixes of each sequence of the data in the GST along with the suffix position and sequence number. The main advantage of building the GST is two-fold. Firstly, it categorize the similar sequence according to the suffix position and sequence number. Secondly, increasing the number of SNPs in the database does not affect the query execution time. We can observe this in Table 7.3. Moreover, we only need one-time pre-processing to build and encrypt the GST.

Secure Centralized Model

We utilize a centralized (outsourced) model where the *Hospital(s)* outsource their databases to the *CS* via trusted entity, *CI*. Once outsourced, the *Hospital(s) and CI*, afterwards can remain offline. On the other hand, the hospitals/data owners do not outsource their databases to a cloud server in the distributed (federated model) model. Therefore, the distributed model requires the data owners to remain online during the computation [130].

Privacy Guaranty

- ***Data privacy.*** The data is encrypted and stored on the cloud server. The semantically secure encryption scheme ensures the privacy of the data even if the cloud server gets compromised.
- ***Query privacy.*** As the query is submitted to GC directly, it is only known to the researcher, and not to any of the other participating entities in the system.
- ***Output privacy.*** The researcher is only aware of the results after decrypting

the results whereas the cloud server only handles the encrypted counterpart. However, we do not prevent any inference attack followed by the outputs of a query as we assume the researchers to be honest.

7.7.2 Limitations

Leakage of Search Pattern in CS

In our proposed method, during the query execution, *CS* learns the tree traversal pattern since it depends on the *researcher* query input and GC output. However, the *CS* will not learn/infer any sensitive information since SNPs value are encrypted and we assume *CS* does not collude with the *researcher*. We could solve this leakage by utilizing oblivious-RAM [131; 63] which will accumulate further computational complexity.

Malicious Researcher

As described in Section 7.5.2, *researcher* receives the keys $(\mathcal{ENC}_k, \mathcal{IV})$ from *CI* to decrypt the encrypted result. Therefore, we do not consider any dishonest *researcher* in this chapter. The *CI* should create and store a profile for each *researcher* and authenticate before sending keys to the *researcher*. Then, as the *researcher* is not anonymous while querying the *CS*, the *CS* monitors the *researcher*'s action to detect any abnormality. If the *CI* catches any misbehaviour, it can block the *researcher*.

7.8 Security Analysis

We analyze the security of our protocols in this section. Based on our assumption that the cloud server does not collude with *researchers* and *CI*, we can prove that the

proposed framework is secure. The security of the proposed framework depends on the security of two primitives: AES in Counter mode and garbled circuits. The security of CTR mode [77] is well-analyzed and well-understood. This cryptosystem provides IND-CPA (indistinguishability under chosen plain text) security guarantee [78] since we are utilizing a different IV in each iteration of the AES-CTR. Garbled circuits supports secure two-party computation against semi-honest adversaries [10]. After running garbled circuits, both parties learn the output of a function but neither of them learn about the input of each other. The certified institution (CI) encrypts the tree and sends it to the cloud server. The cloud server will not be able to learn and decrypt the tree because it does not have the corresponding keys. The researcher sends the query to GC directly and accordingly the cloud server will not be able to learn it. The researcher and cloud server use garbled circuits to execute query such that the researcher and the cloud server will not be able to learn the input of each other.

7.9 Summary

In this chapter, we presented a secure protocol for efficient substring search and set-maximal search. We utilized a *generalized suffix tree* to create an index of the genomic data and secure query execution is ensured via garbled circuit. Our results show that, we achieved significant improvement in query execution time compared to Ishimaki *et al.* [101]. Our proposed method could also solve the secure substring search without a given query start position. In such case, query execution time would be faster since we do not need to check encrypted position using GC after the query

sequence matches with the $G\tilde{S}T$.

Chapter 8

Conclusion

In this thesis, I have presented four different algorithms for secure outsourcing of genomic data in a central repository (cloud server) and performed three different computations (i.e., count query, similar patient query, and substring search query).

8.1 Summary

In Chapter 4, I have introduced a secure and proficient strategy for outsourcing genomic data. The proposed technique develops an *index tree* from the aggregate genomic data and afterward outsources it to the third party cloud server. By utilizing a secure interactive protocol, the cloud server can navigate the nodes of the tree and perform count query operation. I have demonstrated that our model does not uncover any sensitive genomic data during the data processing as well as query execution stage. Moreover, I integrate the phenotype and numeric information along with the genomic data to securely perform the count query along with the range query for the numeric

data.

In the next chapter (Chapter 5), I propose a model for similar sequence search on encrypted data. I incorporate a prefix tree-based indexing technique in my 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, I have used Hamming distance to calculate the similarity measure.

In Chapter 6, I introduce another method of SPQ search using edit distance approximation. I generated a compressed prefix tree using genomic data and all the computation for SPQ is done on this tree. Experimental results on real life dataset demonstrates the efficiency of my model. The main difference between Chapter 5 and Chapter 6 is that in Chapter 5, the sequence similarity result depends on the value of the threshold k, but in Chapter 6 no threshold k is given by the researcher and all the similar sequences are listed as a result of top k query search.

In Chapter 7, I present a secure protocol for efficient substring and set-maximal search. I utilize a generalized suffix tree to create an index of the genomic data, and secure query execution is ensured via the garbled circuit. My results show a significant improvement in query execution time compared to earlier work from Ishimaki *et al.* [101] (substring search) and Shimizu *et al.* [3] (set-maximal search). My proposed method can also be extended to perform other variants of substring searches (i.e., without a predefined query start position).

In summary, the proposed methods satisfy the research objectives presented in the Introduction. In particular, the data owner can outsource its data to an untrusted cloud service provider, and researchers can query on encrypted data. The proposed

framework also provides three different privacy guarantees.

8.2 Future work

8.2.1 Secure Count Query

I 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 [132]. Recently, Intel has introduced Software Guard Extensions (Intel SGX) [133] 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 [82] 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 [9], thus preventing the inference attack.

8.2.2 Secure Similar Patient Query

To achieve secure similar patient query, both in Chapter 5 and 6, I utilized the hamming distance as a similarity measure. In Chapter 5 similar patient are filtered by the threshold value k given by the *researcher*, whereas in Chapter 6 similar patient are filtered by the top-k query search. In future, the similarity matrix could be change

to the edit distance. In the preprocessing method such as utilizing the compressed prefix tree, could also be replaced by other data structure with the edit distance as the similarity measure.

8.2.3 Secure Substring Search and Set-maximal Search

My proposed methods for secure substring search and secure set-maximal search require a query string as an input whose length is smaller than the database sequence length, and a starting position. In the future, this work could certainly be extended to support *set-maximal matches* [96] where the query length will be same as the data sequence length and no start position will be given. Moreover, a threshold can be added along with the substring search query so that *researchers* can achieve results with a certain number of match between the query sequence and any given dataset.

Secure query execution on encrypted data is an active area of research. In the next few years, I assume that current open problems will be addressed and new innovative technologies will be acquainted 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 approaches. 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 method.

Bibliography

[1] 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*, 2015, pp. 492–503. xi, 5, 33, 35, 96, 97, 112, 113

[2] M. M. Al Aziz, D. Alhadidi, and N. Mohammed, "Secure approximation of edit distance on genomic data," *BMC Medical Genomics*, vol. 10, no. 2, 2017. xi, 33, 34, 35, 112, 113

[3] K. Shimizu, K. Nuida, and G. Rätsch, "Efficient privacy-preserving string search and an application in genomics," *Bioinformatics*, vol. 32, no. 11, pp. 1652–1661, 2016. xiii, 36, 37, 116, 117, 139, 140, 146

[4] 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, 41, 97

[5] P. R. Burton, A. L. Hansell, I. Fortier, T. A. Manolio, M. J. Khoury, J. Little, and P. Elliott, "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. 1, 41

[6] 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*, 2009, pp. 201–210. 2, 42

[7] 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. 2

[8] Y. Erlich, J. B. Williams, D. Glazer, K. Yocum, N. Farahany, M. Olson, A. Narayanan, L. D. Stein, J. A. Witkowski, and R. C. Kain, "Redefining genomic privacy: Trust and Empowerment," *Public Library of Science biology*, vol. 12, no. 11, 2014. 3, 42

[9] 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. 3, 25, 74, 94, 147

[10] A. Yao, "How to generate and exchange secrets," in *27th Annual Symposium on Foundations of Computer Science (SFCS)*, 1986, pp. 162–167. 4, 19, 36, 109, 117, 143

[11] R. Durbin, "Efficient haplotype matching and storage using the positional

Burrows–Wheeler transform (PBWT)," *Bioinformatics*, vol. 30, no. 9, pp. 1266–1272, 2014. 6, 37

[12] M. Z. Hasan, M. S. R. Mahdi, M. N. Sadat, and N. Mohammed, "Secure count query on encrypted genomic data," *Journal of Biomedical Informatics*, vol. 81, pp. 41–52, 2018. 7

[13] M. S. R. Mahdi, M. N. Sadat, N. Mohammed, and X. Jian, "Secure count query on encrypted heterogeneous data," in *18th IEEE International Conference on Dependable, Autonomic and Secure Computing*, 2020. 7

[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. 7

[15] M. S. R. Mahdi, M. M. Al Aziz, D. Alhadidi, and N. Mohammed, "Secure similar patients query on encrypted genomic data," *IEEE Journal of Biomedical and Health Informatics*, vol. 23, no. 6, pp. 2611–2618, 2019. 7

[16] M. S. R. Mahdi, M. M. Al Aziz, D. Alhadidi, X. Jian, and N. Mohammed, "Privacy-preserving string search on encrypted genomic data using a generalized suffix tree," 2019. 7

[17] R. F. Woolson and W. R. Clarke, *Statistical methods for the analysis of biomedical data*. John Wiley & Sons, 2011, vol. 371. 9

[18] J. D. Watson and F. Crick, "Molecular structure of nucleic acids: A structure for deoxyribose nucleic acid." *Nature*, vol. 171, pp. 737–738, 1953. 10

[19] H. Pearson, "Genetics: what is a gene?" *Nature*, vol. 441, no. 7092, pp. 398–401, 2006. 10

[20] U. N. L. Medicine, "Genes," http://ghr.nlm.nih.gov/handbook/basics/gene, 2015, online; accessed 18-May-2020. 10

[21] J. C. Venter *et al.*, "The sequence of the human genome," *Science*, vol. 291, no. 5507, pp. 1304–1351, 2001. 10

[22] Y. Nakamura, "DNA variations in human and medical genetics: 25 years of my experience," *Journal of human genetics*, vol. 54, no. 1, pp. 1–8, 2009. 10

[23] A. Ahmadian, B. Gharizadeh, A. C. Gustafsson, F. Sterky, P. Nyrén, M. Uhlén, and J. Lundeberg, "Single-nucleotide polymorphism analysis by pyrosequencing," *Analytical biochemistry*, vol. 280, no. 1, pp. 103–110, 2000. 10

[24] G. T. Marth, I. Korf, M. D. Yandell, R. T. Yeh, Z. Gu, H. Zakeri, N. O. Stitziel, L. Hillier, P.-Y. Kwok, and W. R. Gish, "A general approach to single-nucleotide polymorphism discovery," *Nature genetics*, vol. 23, no. 4, pp. 452–456, 1999. 11

[25] B. W. Kirk, M. Feinsod, R. Favis, R. M. Kliman, and F. Barany, "Single nucleotide polymorphism seeking long term association with complex disease," *Nucleic acids research*, vol. 30, no. 15, pp. 3295–3311, 2002. 11

[26] H. Montgomery, R. Marshall, H. Hemingway, S. Myerson, P. Clarkson, C. Dollery, M. Hayward, D. Holliman, M. Jubb, M. World *et al.*, "Human

gene for physical performance," *Nature*, vol. 393, no. 6682, pp. 221–222, 1998. 12

[27] E. P. Noble, "The D2 dopamine receptor gene: a review of association studies in alcoholism and phenotypes," *Alcohol*, vol. 16, no. 1, pp. 33–45, 1998. 12

[28] D. White and M. Rabago-Smith, "Genotype–phenotype associations and human eye color," *Journal of human genetics*, vol. 56, no. 1, pp. 5–7, 2011. 12

[29] S. J. Chanock, T. Manolio, M. Boehnke, E. Boerwinkle, D. J. Hunter, G. Thomas, J. N. Hirschhorn, G. Abecasis, D. Altshuler, J. E. Bailey-Wilson *et al.*, "Replicating genotype–phenotype associations," *Nature*, vol. 447, no. 7145, p. 655, 2007. 12

[30] S. D. Kahn, "On the future of genomic data," *science*, vol. 331, no. 6018, pp. 728–729, 2011. 13

[31] H. K. Patil and R. Seshadri, "Big data security and privacy issues in healthcare," in *2014 IEEE international congress on big data*. IEEE, 2014, pp. 762–765. 13

[32] G. Danezis and E. De Cristofaro, "Fast and private genomic testing for disease susceptibility," in *Proceedings of the 13th Workshop on Privacy in the Electronic Society*, 2014, pp. 31–34. 13

[33] F. R. Bieber, C. H. Brenner, and D. Lazer, "Finding criminals through DNA of their relatives," 2006. 13

[34] J. M. Butler, "The future of forensic DNA analysis," *Philosophical transactions*

of the royal society B: biological sciences, vol. 370, no. 1674, p. 20140252, 2015. 13

[35] 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. 13

[36] Z. Lin, A. B. Owen, and R. B. Altman, "Genomic research and human subject privacy," *Science*, vol. 305, no. 5681, pp. 183–183, 2004. 13

[37] 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," *Public Library of Science Genetics*, vol. 4, no. 8, 2008. 14

[38] 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*, 2009, pp. 534–544. 14

[39] Y. Erlich and A. Narayanan, "Routes for breaching and protecting genetic privacy," *Nature Reviews Genetics*, vol. 15, no. 6, 2014. 14

[40] C. Hazay and Y. Lindell, *Efficient secure two-party protocols: Techniques and constructions*. Springer Science & Business Media, 2010. 14

[41] T. Araki, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara, "High-throughput

semi-honest secure three-party computation with an honest majority," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 805–817. 14

[42] T. Dimitriou and A. Michalas, "Multi-party trust computation in decentralized environments in the presence of malicious adversaries," *Ad Hoc Networks*, vol. 15, pp. 53–66, 2014. 15

[43] J. Lancrenon, D. Khader, P. Y. Ryan, and F. Hao, "Password-based authenticated key establishment protocols," in *Computer and Information Security Handbook*. Elsevier, 2013, pp. 705–720. 15

[44] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, 2009, pp. 169–178. 15

[45] 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. 15, 17, 18, 55, 69, 73

[46] T. Elgamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *Information Theory, IEEE Transactions on*, vol. 31, no. 4, pp. 469–472, 1985. 15

[47] R. A. Popa, E. Stark, S. Valdez, J. Helfer, N. Zeldovich, and H. Balakrishnan, "Building web applications on top of encrypted data using mylar," in *11th*

USENIX Symposium on Networked Systems Design and Implementation (NSDI 14), 2014, pp. 157–172. 15

[48] J. H. Cheon, M. Kim, and K. Lauter, "Homomorphic computation of edit distance," in *Financial Cryptography and Data Security*, 2015, pp. 194–212. 15, 34, 35, 113

[49] R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *J. ACM*, vol. 21, no. 1, pp. 168–173, Jan. 1974. [Online]. Available: http://doi.acm.org/10.1145/321796.321811 16

[50] K. Lauter, A. López-Alt, and M. Naehrig, "Private computation on encrypted genomic data," in *International Conference on Cryptology and Information Security*, 2014, pp. 3–27. 16, 38

[51] D. Falconer and T. Mackay, *Introduction to Quantitative Genetics*. Longman, 1996. [Online]. Available: https://books.google.ca/books?id=7ASZNAEACAAJ 16

[52] M. Naehrig, K. Lauter, and V. Vaikuntanathan, "Can homomorphic encryption be practical?" in *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*. ACM, 2011, pp. 113–124. 16

[53] W. Wang, Y. Hu, L. Chen, X. Huang, and B. Sunar, "Exploring the feasibility of fully homomorphic encryption," *Computers, IEEE Transactions on*, vol. 64, no. 3, pp. 698–706, 2015. 16

[54] Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption

from (standard) lwe," *SIAM Journal on Computing*, vol. 43, no. 2, pp. 831–871, 2014. 16

[55] L. Ducas and D. Micciancio, "Fhew: Bootstrapping homomorphic encryption in less than a second," in *Advances in Cryptology–EUROCRYPT 2015*. Springer, 2015, pp. 617–640. 16

[56] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*. ACM, 2012, pp. 309–325. 16

[57] M. Van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully homomorphic encryption over the integers," in *Advances in cryptology–EUROCRYPT 2010*. Springer, 2010, pp. 24–43. 16

[58] H. Chen, K. Laine, and R. Player, "Simple encrypted arithmetic library-seal v2. 1," in *International Conference on Financial Cryptography and Data Security*. Springer, 2017, pp. 3–18. 17

[59] "Microsoft SEAL: an easy-to-use and powerful homomorphic encryption library." https://github.com/Microsoft/SEAL, online; accessed 13 May, 2020. 17

[60] S. Halevi and V. Shoup, "Algorithms in helib," in *Annual Cryptology Conference*. Springer, 2014, pp. 554–571. 17

[61] "An Implementation of homomorphic encryption." https://github.com/shaih/HElib, online; accessed 13 May, 2020. 17

[62] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *Journal of the ACM (JACM)*, vol. 43, no. 3, pp. 431–473, 1996. 18

[63] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: an extremely simple oblivious RAM protocol," in *Proceedings of the ACM SIGSAC conference on Computer & Communications Security*, 2013, pp. 299–310. 19, 142

[64] E. Stefanov and E. Shi, "Oblivistore: High performance oblivious cloud storage," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 253–267. 19

[65] M. O. Rabin, "How To Exchange Secrets with Oblivious Transfer." *IACR Cryptology ePrint Archive*, vol. 2005, 2005. 20, 21

[66] 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. 21

[67] M. Naor and B. Pinkas, "Computationally secure oblivious transfer," *Journal of Cryptology*, vol. 18, no. 1, pp. 1–35, 2005. 21, 36

[68] ——, "Oblivious transfer and polynomial evaluation," in *Proceedings of the thirty-first annual ACM symposium on Theory of computing*, 1999, pp. 245–254. 21

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

computation," in *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE, 2008, pp. 216–230. 22

[70] Y. Huang, D. Evans, J. Katz, and L. Malka, "Faster secure two-party computation using garbled circuits." in *USENIX Security Symposium*, 2011. 22, 69

[71] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, "Oblivm: A programming framework for secure computation," in *IEEE Symposium on Security and Privacy (S&P)*, 2015, pp. 359–376. 22, 69

[72] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway, "Efficient garbling from a fixed-key blockcipher," in *IEEE Symposium on Security and Privacy (S&P)*, 2013, pp. 478–492. 22, 69

[73] 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*, 2015, pp. 850–861. 22, 69, 91, 109, 134

[74] "Semi-honest Two Party Computation Based on Garbled Circuits." https://github.com/emp-toolkit/emp-sh2pc, online; accessed 13 May, 2020. 22

[75] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970. 22

[76] F. Grandi, "On the analysis of bloom filters," *Information Processing Letters*, vol. 129, pp. 35–39, 2018. 24

[77] C. Paar and J. Pelzl, *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009. 24, 104, 109, 127, 143

[78] J. Katz and Y. Lindell, *Introduction to modern cryptography*. CRC press, 2014. 24, 143

[79] H. Li and R. Durbin, "Fast and accurate short read alignment with burrows–wheeler transform," *bioinformatics*, vol. 25, no. 14, pp. 1754–1760, 2009. 26

[80] J. W. Daykin, R. Groult, Y. Guesnet, T. Lecroq, A. Lefebvre, M. Léonard, L. Mouchard, É. Prieur-Gaston, and B. Watson, "Efficient pattern matching in degenerate strings with the Burrows–Wheeler transform," *Information Processing Letters*, vol. 147, pp. 82–87, 2019. 26

[81] 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. 28, 30, 43, 44, 71, 72

[82] 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. 30, 43, 44, 46, 71, 72, 147

[83] R. Agrawal, D. Asonov, M. Kantarcioglu, and Y. Li, "Sovereign joins," in *22nd International Conference on Data Engineering (ICDE)*, 2006, pp. 26–26. 31

[84] M. Naveed, E. Ayday, E. W. Clayton, J. Fellay, C. A. Gunter, J.-P. Hubaux,

B. A. Malin, and X. Wang, "Privacy in the genomic era," *ACM Computing Surveys (CSUR)*, vol. 48, no. 1, 2015. 31

[85] T. Dugan and X. Zou, "A Survey of Secure Multiparty Computation Protocols for Privacy Preserving Genetic Tests," in *IEEE First International Conference on Connected Health: Applications, Systems and Engineering Technologies (CHASE)*, 2016, pp. 173–182. 32

[86] Y. Zhang, M. Blanton, and G. Almashaqbeh, "Secure distributed genome analysis for GWAS and sequence comparison computation," *BMC Medical Informatics and Decision Making*, 2015. 32, 40

[87] H. Perl, Y. Mohammed, M. Brenner, and M. Smith, "Fast confidential search for bio-medical data using bloom filters and homomorphic cryptography," in *IEEE 8th International Conference on E-Science*, 2012, pp. 1–8. 32

[88] 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*, 2010, pp. 519–536. 32

[89] C. Wang, K. Ren, S. Yu, and K. M. R. Urs, "Achieving usable and privacy-assured similarity search over outsourced cloud data," in *Proceedings of the IEEE INFOCOM*, 2012, pp. 451–459. 33

[90] S. Jha, L. Kruger, and V. Shmatikov, "Towards practical privacy for genomic computation," in *IEEE Symposium on Security and Privacy (S&P)*, 2008, pp. 216–230. 33, 35

[91] R. Wang, X. Wang, Z. Li, H. Tang, M. K. Reiter, and Z. Dong, "Privacy-preserving genomic computation through program specialization," in *Proceedings of the 16th ACM conference on Computer and communications security*, 2009, pp. 338–347. 33, 34, 35

[92] G. Asharov, S. Halevi, Y. Lindell, and T. Rabin, "Privacy-preserving search of similar patients in genomic data," *Proceedings on Privacy Enhancing Technologies*, vol. 2018, no. 4, pp. 104–124, 2018. 33, 35, 96, 97

[93] W. K. Wong, D. W.-l. Cheung, B. Kao, and N. Mamoulis, "Secure kNN computation on encrypted databases," in *Proceedings of the ACM SIGMOD International Conference on Management of data*, 2009, pp. 139–152. 34

[94] L. Zhou, Y. Zhu, and A. Castiglione, "Efficient k-NN query over encrypted data in cloud with limited key-disclosure and offline data owner," *Computers & Security*, vol. 69, pp. 84–96, 2017. 34

[95] Q. Wang, S. Hu, K. Ren, M. He, M. Du, and Z. Wang, "CloudBI: Practical privacy-preserving outsourcing of biometric identification in the cloud," in *European Symposium on Research in Computer Security*, 2015, pp. 186–205. 34

[96] "iDASH Privacy and security workshop 2018 - secure genome analysis competition," http://www.humangenomeprivacy.org/2018/competition-tasks.html, online; accessed 24 January, 2019. 35, 119, 133, 148

[97] M. J. Atallah, F. Kerschbaum, and W. Du, "Secure and private sequence com-

parisons," in *Proceedings of the ACM workshop on Privacy in the electronic society*, 2003, pp. 39–44. 36

[98] J. R. Troncoso-Pastoriza, S. Katzenbeisser, and M. Celik, "Privacy preserving error resilient DNA searching through oblivious automata," in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007, pp. 519–528. 36

[99] J. Katz and L. Malka, "Secure text processing with applications to private DNA matching," in *Proceedings of the 17th ACM conference on Computer and communications security*, 2010, pp. 485–492. 36

[100] H. Sudo, M. Jimbo, K. Nuida, and K. Shimizu, "Secure Wavelet Matrix: Alphabet-Friendly Privacy-Preserving String Search," *BioRxiv*, 2016. 36

[101] Y. Ishimaki, H. Imabayashi, K. Shimizu, and H. Yamana, "Privacy-Preserving String Search for Genome Sequences with FHE bootstrapping optimization," in *IEEE International Conference on Big Data (Big Data)*, 2016, pp. 3989–3991. 36, 37, 116, 117, 139, 140, 143, 146

[102] M. Blanton and M. Aliasgari, "Secure outsourcing of DNA searching via finite automata," in *IFIP Annual Conference on Data and Applications Security and Privacy*, 2010, pp. 49–64. 36

[103] 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, 2013, pp. 95–106. 38

[104] 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. 38

[105] 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. 39, 40

[106] 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. 39

[107] C. C. Khor, S. Davila, W. B. Breunis *et al.*, "Genome-wide association study identifies FCGR2A as a susceptibility locus for Kawasaki disease," *Nature Genetics*, vol. 43, no. 12, pp. 1241–1246, 2011. 39

[108] H. Zhicong, A. Erman, F. Jacques, H. Jean-Pierre, and J. Ari, "GenoGuard: Protecting Genomic Data against Brute-Force Attacks," in *IEEE Symposium on Security and Privacy (S&P)*, May 2015, pp. 447–462. 39

[109] 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. 40

[110] 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)*, 2016. 40

[111] W. Xie, M. Kantarcioglu, W. S. Bush, D. Crawford, J. C. Denny, R. Heatherly, and B. A. Malin, "SecureMA: protecting participant privacy in genetic association meta-analysis," *Bioinformatics*, vol. 30, no. 23, pp. 3334–3341, 2014. 40

[112] S. Wang, Y. Zhang, W. Dai, K. Lauter, M. Kim, Y. Tang, H. Xiong, and X. Jiang, "HEALER: Homomorphic computation of ExAct Logistic rEgRession for secure rare disease variants analysis in GWAS," *Bioinformatics*, vol. 32, no. 2, pp. 211–218, 2015. 40

[113] T. Rahman, M. Czerwinski, R. Gilad-Bachrach, and P. Johns, "Predicting about-to-eat moments for just-in-time eating intervention," in *Proceedings of the 6th International Conference on Digital Health Conference*. ACM, 2016, pp. 141–150. 43

[114] S. Abdullah, E. L. Murnane, M. Matthews, M. Kay, J. A. Kientz, G. Gay, and T. Choudhury, "Cognitive rhythms: unobtrusive and continuous sensing of alertness using a mobile phone," in *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. ACM, 2016, pp. 178–189. 43

[115] H. Ahmadi, N. Pham, R. Ganti, T. Abdelzaher, S. Nath, and J. Han, "Privacy-

aware regression modeling of participatory sensing data," in *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*. ACM, 2010, pp. 99–112. 43

[116] S. Purcell, B. Neale, K. Todd-Brown, L. Thomas, M. A. Ferreira, D. Bender, J. Maller, P. Sklar, P. I. De Bakker, M. J. Daly *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. 46

[117] D. N. Paltoo, L. L. Rodriguez, M. Feolo *et al.*, "Data use under the NIH GWAS Data Sharing Policy and future directions," *Nature Genetics*, vol. 46, no. 9, pp. 934–938, 2014. 47, 49, 98

[118] C. Hazay and Y. Lindell, *Efficient secure two-party protocols: Techniques and constructions*. Springer Science & Business Media, 2010. 49, 98

[119] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft, "Privacy-preserving ridge regression on hundreds of millions of records," in *IEEE Symposium on Security and Privacy (S&P)*, 2013, pp. 334–348. 57

[120] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S. G. Choi, W. George, A. Keromytis, and S. Bellovin, "Blind Seer: A Scalable Private DBMS," in *IEEE Symposium on Security and Privacy (S&P)*, May 2014, pp. 359–374. 60

[121] G. M. Church, "The personal genome project," *Molecular systems biology*, vol. 1, no. 1, 2005. 69

[122] 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. 76

[123] D. Vatsalan and P. Christen, "Privacy-preserving Matching of Similar Patients," *Journal of Biomedical Informatics*, vol. 59, pp. 285–298, 2016. 77

[124] "iDASH Privacy and security workshop 2015 - secure genome analysis competition," http://www.humangenomeprivacy.org/2015/competition-tasks.html, 2015. 91

[125] A. Aziz, M. Momin, M. Z. Hasan, N. Mohammed, and D. Alhadidi, "Secure and Efficient Multiparty Computation on Genomic Data," in *Proceedings of the 20th International Database Engineering & Applications Symposium (IDEAS)*, 2016, pp. 278–283. 96

[126] "iDASH Privacy and security workshop 2016 - secure genome analysis competition," http://www.humangenomeprivacy.org/2016/competition-tasks.html, online; accessed 23 July 2017. 110

[127] E. Ukkonen, "On-line construction of suffix trees," *Algorithmica*, vol. 14, no. 3, pp. 249–260, 1995. 116, 125

[128] G. Oded, *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, 2009. 122

[129] D. Gusfield, *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge University Press, 1997. 125

[130] A. Salem, P. Berrang, M. Humbert, and M. Backes, "Privacy-preserving similar patient queries for combined biomedical data," *in Proceedings of the Privacy Enhancing Technologies*, vol. 2019, no. 1, pp. 47–67, 2019. 141

[131] C. Gentry, K. A. Goldman, S. Halevi, C. Julta, M. Raykova, and D. Wichs, "Optimizing ORAM and using it efficiently for secure computation," in *International Symposium on Privacy Enhancing Technologies Symposium*, 2013, pp. 1–18. 142

[132] C. Gentry, *A fully homomorphic encryption scheme*. Stanford University, 2009. 147

[133] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, "Using innovative instructions to create trustworthy software solutions," in *Proceedings of the International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, vol. 11, 2013. 147