Privacy-Preserving Techniques for Genomic Data

by

Md Momin Al Aziz

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
May 2022

© Copyright 2022 by Md Momin Al Aziz

Thesis advisor | Author
Noman Mohammed | **Md Momin Al Aziz**

Privacy-Preserving Techniques for Genomic Data

Abstract

Genomic data hold salient information about the characteristics of a living organism. Throughout the last decade, pinnacle developments have given us more accurate and inexpensive methods to retrieve our genome sequences. However, with the advancement of genomic research, there are growing security and privacy concerns regarding collecting, storing, and analyzing such sensitive data. Recent results show that given some background information, it is possible for an adversary to re-identify an individual from a specific genomic dataset. This can reveal the current association or future susceptibility of some diseases for that individual (and sometimes the kinship between individuals), resulting in a privacy violation.

This thesis has two parts and proposes several techniques to mitigate the privacy issues relating to genomic data. In our first part, we target the data privacy issues while using any external computational environment. We propose privacy-preserving frameworks to store genomic data in an untrusted computational environment (*i.e.*, cloud). In particular, we employ prefix and suffix tree structures to represent genomic data while keeping them under encryption throughout its computational life-cycle. Therefore, the underlying methods perform different string search queries and arbitrary computations under encryption without requiring access to the raw sensitive data. We also propose a GPU-parallel Fully Homomorphic Encryption framework that optimizes existing algorithms and can perform string distance metrics such as Hamming, Edit distance and Set Maximal Matching. The GPU-parallel framework is 14.4 and 46.81 times faster for standard and matrix multiplications, respectively compared to the existing techniques.

The second part of the thesis targets another privacy setting where the outputs from different genomic data analyses are deemed sensitive. Here, we propose several

differentially private mechanisms to share partial genome datasets and intermediate statistics providing a strict privacy guarantee. Experimental results demonstrate that the proposed methods are effective for protecting data privacy while computing and analysis of genomic data. Overall, the proposed techniques in this thesis are not specialized for genomic data but can be generalized to protect other types of sensitive data.

Copyright Notices and Disclaimers

Sections of this thesis have been published in different journals and conference proceedings, either previously or forthcoming at the time of publication. Following is a list of publications in which portions of this work appeared, organized by chapter:

Chapter 2

- **Md Momin Al Aziz**, Md Nazmus Sadat, Dima Alhadidi, Shuang Wang, Xiaoqian Jiang, Cheryl L. Brown, and Noman Mohammed. Privacy-preserving techniques of genomic data—a survey. Briefings in bioinformatics, 20(3), 887-895 pages, 2019. (impact factor 11.62). [1]

Chapter 3

- **Md Momin Al Aziz**, Parimala Thulasiraman, and Noman Mohammed. Parallel Generalized Suffix Tree Construction for Genomic Data. In Proceedings of the 7th International Conference on Algorithms for Computational Biology, Missoula, Montana USA, April 2020. [2]
- **Md Momin Al Aziz**, Parimala Thulasiraman, and Noman Mohammed. (2020). Parallel and Private Generalized Suffix Tree Construction and Query on Genomic Data. BMC Genomic Data, to appear.

Chapter 4

- Luyao Chen, **Md Momin Al Aziz**, Noman Mohammed, and Xiaoqian Jiang. Secure large-scale genome data storage and query. Elsevier Computer methods and programs in Biomedicine, 165, 129-137 pages, 2018 (impact factor 5.42). [3] (**co-first author**)

Chapter 5

- Toufique Morshed, **Md Momin Al Aziz**, and Noman Mohammed. CPU and GPU accelerated fully homomorphic encryption. In Proceedings of the IEEE International Symposium on Hardware Oriented Security and Trust (HOST), San Jose, California USA, December 2020. [4].

- **Md Momin Al Aziz**, Toufique Morshed, and Noman Mohammed. Secure Genomic String Search queries using accelerated Parallel Fully Homomorphic Encryption. (under preparation for journal submission)

Chapter 6

- **Md Momin Al Aziz**, Shahin Kamali, Noman Mohammed, and Xiaoqian Jiang. (2021). Online Algorithm for Differentially Private Genome-wide Association Studies. ACM Transactions on Computing for Healthcare, 2(2), 1-27, 2021. [5]

Chapter 7

- **Md Momin Al Aziz**, Md Monowar Anjum, Xiaoqian Jiang, and Noman Mohammed. Generalized Genomic Data Sharing for Differentially Private Federated Learning. Elsevier Journal of Biomedical Informatics, to appear (impact factor 6.317).

Contents

Abstract ii
Copyright Notices and Disclaimers iv
Table of Contents ix
List of Figures x
List of Tables xii
Acknowledgments 1

1 Introduction **2**
1.1 Motivation 2
1.2 Research Objectives 5
1.3 Contributions 7
1.3.1 Part 1: Protecting Data Privacy 7
1.3.2 Part 2: Protecting Output Privacy 9
1.4 Organization 10

2 Background **12**
2.1 General Architecture 12
2.1.1 Entities in Genomic Data Computations 12
2.1.2 Problem Architecture 13
2.1.3 Threat or Adversary Models 14
2.2 Privacy-Preserving Techniques 15
2.2.1 Homomorphic Encryption 16
2.2.2 Garbled Circuit 19
2.2.3 Differential Privacy 22

I Protecting Data Privacy on Untrusted Environment **27**

3 Privacy-Preserving Indexing and String Queries using Generalized Suffix Trees **28**
3.1 Introduction 28
3.2 Preliminaries 31

3.2.1 Haplotype Data 31
3.2.2 Generalized Suffix Tree 32
3.2.3 String Queries 34
3.3 Methods 36
3.3.1 Problem Architecture 36
3.3.2 Parallel GST Construction 39
3.3.3 Privacy Preserving Query Execution 43
3.4 Results 49
3.4.1 Evaluation Datasets and Implementation 50
3.4.2 Performance Analysis 51
3.4.3 Query Execution 53
3.4.4 Limitations and Discussions 56
3.5 Related Works 57
3.5.1 Privacy-preserving String Search 57
3.5.2 Parallel GST Construction 59
3.6 Conclusion 61

4 Large-Scale Genome Data Storage and Query with Privacy-Preserving Techniques 62
4.1 Introduction 62
4.2 Preliminaries 65
4.2.1 Data Representation 65
4.2.2 Graph Database 65
4.2.3 System Architecture Overview 66
4.3 Methods 68
4.3.1 Data Preprossessing 68
4.3.2 Counting Tree Construction 69
4.3.3 Indexing the Counting Tree 70
4.3.4 Encryption of the Tree 73
4.3.5 Search Operation 73
4.4 Results 75
4.4.1 Experimental Setup 75
4.4.2 Storage Requirements 76
4.4.3 Execution Time 77
4.5 Discussion 78
4.6 Related Work 80
4.7 Conclusion 81

5 Genomic String Search using Parallel Fully Homomorphic Encryption 82
5.1 Introduction 82
5.2 Preliminaries 85

5.2.1 Torus FHE (TFHE) . . . 85
5.2.2 Sequential Framework . . . 87
5.2.3 CPU-based Parallel Framework . . . 88
5.2.4 String Search: Problem Definition . . . 90
5.3 Methods . . . 91
5.3.1 GPU-based Parallel Framework . . . 91
5.3.2 Secure String Search Operations . . . 101
5.4 Results . . . 105
5.4.1 GPU-accelerated TFHE . . . 106
5.4.2 Compound Gate Analysis . . . 107
5.4.3 Addition . . . 108
5.4.4 Multiplication . . . 109
5.4.5 Karatsuba Multiplication . . . 111
5.4.6 String Search Operations . . . 111
5.5 Discussion . . . 112
5.6 Related Works . . . 113
5.6.1 Parallel Frameworks for FHE . . . 113
5.6.2 Secure String Distances in Genomic Data . . . 116
5.7 Conclusion . . . 116

II Privacy Preserving Outputs from Genomic Data Analysis 118

6 Online Algorithm for Differentially Private GWAS 119
6.1 Introduction . . . 119
6.1.1 Related Works . . . 121
6.1.2 Contributions . . . 124
6.2 Preliminaries . . . 125
6.2.1 Problem Description . . . 125
6.2.2 Privacy Models . . . 126
6.2.3 Genomic Data and GWAS . . . 127
6.2.4 Online Algorithms and Bin Packing . . . 129
6.3 Methods . . . 131
6.3.1 Differentially Private GWAS . . . 131
6.3.2 Global Model $\mathcal{A}$ . . . 134
6.3.3 Local Model $\mathcal{A}_l$ . . . 139
6.3.4 Privacy Composition with Online Bin Packing . . . 142
6.4 Results . . . 145
6.4.1 Dataset . . . 146
6.4.2 Experimental Setting . . . 146
6.4.3 Global Model . . . 148

6.4.4 Local Model . . . 150
6.5 Discussion . . . 151
6.5.1 Privacy . . . 151
6.5.2 Utility . . . 152
6.5.3 Limitations and Future Work . . . 153
6.6 Conclusion . . . 155

7 Generalized Genomic Data Sharing for Differentially Private Federated Learning 159
7.1 Introduction . . . 159
7.2 Preliminaries . . . 161
7.2.1 Differential Privacy . . . 161
7.2.2 Federated Learning . . . 162
7.3 Related Works . . . 164
7.4 Methods . . . 166
7.4.1 Problem Description . . . 166
7.4.2 Summary of the proposed method . . . 167
7.4.3 Reducing the Data Dimension . . . 168
7.4.4 Privacy Preserving Mechanism . . . 169
7.4.5 Federated Learning Mechanism . . . 174
7.5 Results . . . 175
7.5.1 Dataset and Experimental Setup . . . 176
7.5.2 Area Under Curve (AUC) . . . 176
7.5.3 Execution Time . . . 178
7.5.4 Other Datasets . . . 179
7.6 Discussion . . . 179
7.7 Conclusion . . . 185

8 Conclusion 187
8.1 Summary . . . 187
8.2 Future work . . . 188

Bibliography 219

List of Figures

1.1 Research objectives considered in this thesis 6

2.1 Different entities involved in a genomic data computation 13
2.2 Centralized architecture for sharing and compute on private genomic data 15
2.3 Timeline of the evolution of genomic data studies and seminal development of different privacy preserving techniques 16
2.4 Example of homomorphic operations on encrypted values 18
2.5 Garbled Circuit Execution 20

3.1 Uncompressed Suffix Tree (Trie) construction 32
3.2 GST from Figure 3.1 where gray and white vertices are from `S1` and `S2`, respectively 34
3.3 Computational architecture where data owner has the dataset $\mathcal{D}$ while a researcher submits query q 37
3.4 Vertical partitioning with path graphs $(\%1, \%2)$ merging 38
3.5 Example of Bi-Directional partitioning scheme 40
3.6 Reverse Merkle Hash for Suffix Tree on $S1 = 010101$ where we hash the value of each node in a top-down fashion 43

4.1 Representation of relational and graph database 65
4.2 System Architecture and secure query process 67
4.3 Building the tree from genomic sequence according to Algorithm. The numbers under SNP values in (b) are the counts 68
4.4 Setting position tags on each level and their nodes. The numbers in the parenthesis are sequential labels (position tags) of each layer to be used for generating node ranges for quick indexing 72
4.5 Range of position tags from underlying child in each nodes. The range of each node is the union of ranges of its children 74
4.6 Execution time (seconds) for searching *one* leaf node on different number of SNPs in the Counting Tree 77

4.7 Execution time (seconds) for searching different number of SNPs (randomly selected) in the Counting Tree . . . 79
4.8 Increment of the number of nodes with number of patients . . . 80

5.1 Bitwise addition of two n-bit numbers A and B. a_i, b_i, c_i, r_i are i^{th}-bit of A, B, *carry*, and the *result* . . . 89
5.2 Arbitrary operation between two bits where BS, KS key represents bootstrapping and key switching keys, respectively . . . 92
5.3 Coalescing n-LWE samples (ciphertexts) for n-bits . . . 93
5.4 1-bit Increment and Decrement using Half-adder or subtractor where the $\mathbf{x}_i$ is the input Bit and the Carry bit is propagated into the next bit's operation . . . 95
5.5 Accumulating $n = 8$ LWE samples (L_{ij}) in parallel using a tree-based reduction . . . 96
5.6 Performance analysis of GPU-accelerated TFHE with the sequential and CPU || frameworks . . . 105

6.1 Privacy preserving Genome-wide Association Studies (GWAS) models where data owners share the data (or results) with a central aggregator 126
6.2 Single Nucleotide Polymorphisms (SNPs) in DNA, where `C` and `A` are major and minor allele respectively . . . 128
6.3 Packing 17 items of different size $(0, 1]$ according to next fit algorithm 130
6.4 Accuracy of GWAS (LD, HWE, CATT and FET) averaged over 1000 queries and 10 iterations with three different privacy budgets . . . 148
6.5 Privacy loss and accuracy relation for LD, HWE, CATT and FET where x-axis and y-axis denotes privacy loss ϵ and accuracy respectively 149

7.1 Overview of the privacy problem and proposed solution where multiple data owners are targeting to train a model collaboratively using arbitrary machine learning algorithm given a privacy guarantee over the data . . . 167
7.2 Generating differentially private histogram to release genomic data for federated machine learning algorithm . . . 172
7.3 Accuracy difference with different privacy budgets and methods with a fixed reduced dimension $m' = 20$ . . . 175
7.4 AUC results for Naive Bayes, Random Forest and XGBoost on different ϵ values ($m' = 20$) with custom historgram . . . 177
7.5 Accuracy difference for Naive Bayes and XGBoost on different dimension of data (m') while training privately with $\epsilon = 5$ or without any privacy mechanism . . . 181
7.6 Execution time for Naive Bayes Random Forest, and XGBoost on different dimension size and LAN, WAN settings . . . 182
7.7 Summary of the differentially private data sharing methods . . . 186

List of Tables

1.1 Notable privacy attacks with the help of public genomic data 3

2.1 Mathematical Notations used in the thesis along with their description 13
2.2 Comparison of popular HE implementations 17
2.3 A comparative analysis of existing Homomorphic Encryption schemes for different parameters on 32-bit number where Size is in kilobytes and Additions, Multiplications time in milliseconds 18

3.1 Sample haplotype data representation where $s_i \in \{0, 1\}$ are the different positions on the same sequence 35
3.2 Horizontal and Vertical partition scheme execution time (in minutes) to build GSTs with number of processors $p = \{1, 2, 4, 8, 16\}$ 50
3.3 Execution time (in seconds) of bi-directional partitioning to build GST on different datasets with number of processors $p = \{1, 4, 8, 16\}$ . . . 50
3.4 Maximum Execution time (seconds) of Tree Building (TB), Add Path (AP) and Tree Merge (TM) for D_{1000} 52
3.5 Speedup analysis on D_{1000} for all methods with $p = \{2, 4, 8, 16\}$ . . . 53
3.6 Exact Matching, SMM and TSMM (Query 3.2.3, 3.2.5 and 3.2.6) using GST considering different datasets and query lengths (time in milliseconds) 55
3.7 Secure Exact Matching (EM), SMM and TSMM (Query 3.2.3, 3.2.5 and 3.2.6) using $\mathcal{HI}$ considering different datasets and query lengths (time in milliseconds). QP, GC, $|q|$ denotes query processing time, Garbled Circuit, and Query Length respectively. 56
3.8 Related works in different privacy-preserving genomic string search . 59
3.9 Design-level comparison of previous and our method in parallel GST construction 60

4.1 Example genomic data containing multiple patients 66
4.2 Operations and their required time 76
4.3 Size of different elements of the Counting Tree in the Neo4J database 76
4.4 Relationship of the execution time with query size on different scenarios 78

4.5 Comparison of related works on Secure Count Query chronologically . 81

5.1 A comparison of the execution times (sec) of TFHE [6] and our CPU, GPU framework for 32-bit numbers . . . 84
5.2 Computation time (ms) for Bootstrapping, Key Switching and Misc. for sequential and GPU framework . . . 105
5.3 Execution time (sec) for the n-bit addition . . . 108
5.4 Execution time (sec) for vector addition . . . 108
5.5 Multiplication execution time (sec) comparison . . . 110
5.6 Execution time (min) for vector multiplication . . . 110
5.7 Execution time (in seconds) for variable size query and target sequence m for different distance metrics . . . 111
5.8 A comparative analysis of existing Homomorphic Encryption schemes for different parameters on 32-bit number. . . . 113

6.1 A summary of the accuracy results for seven GWAS with different privacy budgets (details on Section 6.4) . . . 124
6.2 Sample Data ($\mathcal{DB}$) representation for GWAS . . . 126
6.3 Notations used in the proposed method . . . 128
6.4 Contingency table for SNP1 with `C` and `A` as the major and minor allele, respectively . . . 132
6.5 Privacy loss for each GWAS query on different budget classes without any dataset partitions . . . 157
6.6 Comparison between Simmons *et al.* [7]'s ($\epsilon_A = 7, \epsilon_B = 5$ and $\epsilon_C = 2$) and our method on EigenSTRAT and LMM GWAS on top-5 SNPs . 157
6.7 Local model accuracy using both Laplace and Randomized Response methods for LD, HWE, CATT and FET . . . 157
6.8 Benchmarking on TDT with Wang *et al.* [8] ($\epsilon_A = 7, \epsilon_B = 5$ and $\epsilon_C = 2$) on top-10 SNPs . . . 157
6.9 Average and standard deviation of the accuracy and privacy loss values of LD, HWE, CATT and FET over 1,000 random queries and 10 iterations 158
6.10 Local model accuracy using Laplacian methods for HWE, CATT and FET from 3, 6, and 9 data owners . . . 158

7.1 Gene Expression data collected at individual data owner (BC-TCGA) 168
7.2 Different experimental parameters considered in this work . . . 174
7.3 AUC for *GSE2034* and *GSE25066* datasets using Naive Bayes and Random Forest on different parameters such as reduced dimension $m' \in \{20, 50\}$ and privacy budget $\epsilon \in \{5, 10, 15\}$. The maximum baseline AUC (w/o privacy) on GSE2034 and GSE25066 was 0.71 and 0.79 respectively. . . . 179
7.4 Official benchmarking results from the iDASH 2021 . . . 183

Acknowledgments

I would like to express my gratitude to Almighty God for granting me the opportunity to write this thesis.

I would like to thank my supervisor, Dr Noman Mohammed for giving me the opportunity to work under his supervision. I am very grateful for his guidance, suggestions, and feedback throughout the preparation of this thesis. I am thankful to the other members of my committee, Drs. Parimala Thulasiraman and Yang Zhang for the comment and advice they provided.

I am also grateful to all my collaborators, Drs. Xiaoqian Jiang, Shuang Wang, Shahin Kamali, Dima Alhadidi (in no particular order) for their guidance. I am also thankful to my colleagues Md Waliullah, Zahidul Hasan, Nazmus Sadat, Kazi Wasif Ahmed, Tanbir Ahmed, Toufique Morshed, Monowar Anjum and Safiur Rahman Mahdi for their valuable consults and time. I am equally grateful to all the faculties of UofManitoba Computer Science and Bangladesh University of Engineering and Technology who shaped my research and academic standing.

Finally, I would like to thank *my wife and family* for their countless sacrifices and inspirations throughout the years.

Chapter 1

Introduction

1.1 Motivation

The achievements in genomics research have resulted in an abundance of human genomic data over the past decade. Genomic data have largely impacted the health sciences and related scientific researches extending our understanding of different diseases and our overall well-being. Multiple areas of medical genomics are currently realistic due to such accumulation of genomic data [9]. Throughout the world, large and varied genomic datasets now help researchers understand the relation between our genomic codes, our ancestry and future disease susceptibility [10, 11].

However, human genomic data are highly sensitive as it can be used to uniquely identify ourselves from any publicly available datasets. Furthermore, it provides critical information about the susceptibility of several genetic diseases for any individual that is sensitive and considered private information. Also, these unique genomic sequences impose a greater privacy risk as Table 1.1 reveals various privacy attacks using such data. For example, one of the seminal work by Homer *et al.* [15] showed the vulnerability of publicly available data and as a result, public access to any genomic dataset was restricted. Notably, these datasets are also storage exhaustive (varying in range of 100-200 GB [24]) and require a high-performance computation when processed.

During the last decade, external computation services (*i.e.*, cloud computing

Table 1.1: Notable privacy attacks with the help of public genomic data

Author	**Year**	**Summary**
Malin and Sweeney [12]	2000	Identifying of DNA sequence based on available health records and disease background knowledge
Gottlib [13]	2001	Finding employees who are susceptible to genetic diseases depending on genomic data
Lin *et al.* [14]	2004	Identifying a person by only 75 independent SNPs
Homer *et al.* [15]	2008	Telling if a user is present in a DNA mixture
Goodrich [16]	2009	Revealing information about the full identity of an encrypted genomic query sequence
Humbert *et al.* [17]	2013	Inferring close relatives' genomes using statistical inference
Sweeney *et al.* [18]	2013	Identify the individuals in the Personal Genome Project (PGP) by Name
Gymrek *et al.* [19]	2013	Identifying personal genomes from surnames by profiling short tandem repeats on the Y-chromosome
Fredrikson *et al.* [20]	2014	Predicting genetic markers using machine learning models on differentially private data
Shringarpure and Bustamante [21]	2015	Identifying participants from a genomic database (with beacon services) with limited number of queries
Raisaro *et al.* [22]	2016	Modifying attack on beacon services with better adversarial knowledge
Harmanci and Gerstein [23]	2016	Linking phenotypes to genotypes from publicly known genotype-phenotype correlation

solutions) became popular for large-scale storage and computations. These commercial systems enable organizations to tap into larger on-demand infrastructure at a lower price point. Therefore, it is a viable solution for genomic data researchers as it reduces the hardware and software investments for their arbitrary computation requirements based on different research studies. Despite the upfront benefits, health care institutes are reluctant to adopt the commercial cloud infrastructure or external servers as it mandates outsourcing sensitive data to an untrusted service provider that might result in a security breach or privacy violation. Nevertheless, using

personal or organizational infrastructure for health-care data is not risk-free as there has been ever-growing reports of data breach (Data breach reports [25]). Overall, the underlying privacy concerns for outsourcing computations on genomic data is a major hurdle in advancing our knowledge.

These genomic data privacy attack models can be classified into two major groups [26, 27]: *a*) re-identification attack with background knowledge or other online data source and *b*) inference about physical attributes (phenotype) or disease association. One popular approach to mitigate these problems is to enforce strict privacy policies on sharing data. This strategy is effective but challenging as different laws and regulations are followed around institutions worldwide which govern the sharing and disclosure of these sensitive data. Though these policies are protecting the privacy of the participating individuals, they are not the final answer. For example, the time needed by a governing body to review researchers' applications requesting the access of datasets is tedious, and it adversely affects timely research outcomes. This delay often demotivates researchers to pursue specific studies. Furthermore, we cannot foresee the future attacks on genomic data, resulting in a much more generalized policy settings or they can fall short for a new or advanced attack strategy [15].

Privacy attacks can occur at different stages of genomic data: from data collection, storage, computation or even result dissemination [28]. In this thesis, we target two such attack surfaces: a) Privacy while storage and computation and b) Privacy for the outputs (Figure 1.1). In general, we propose cryptographic solutions to propose privacy-preserving mechanisms to supplement the existing policy-based privacy solutions to mitigate the privacy issues in these areas. Since, cryptography is a mature area of research, it can provide a much needed assistance protecting the sensitivity of genomic data. Also, with seminal developments in multiple crypto-primitives in recent years, it should positively impact towards formulating privacy-preserving solutions tailored for genomic data. Therefore, state-of-the-art cryptographic techniques can guarantee both privacy and utility of genomic data which is the underlying research problem.

1.2 Research Objectives

In this thesis, we target the privacy concerns of genomic data storage, computations and outputs. Here, we categorize the privacy threats into two major areas:

Objective 1: *Protecting data privacy on untrusted environment.* The first objective is to protect the privacy of the genomic data when data are stored in an untrusted environment for computation. Here, we intend to investigate cryptographic approaches to perform different statistical operations protecting the privacy of individual records. The proposed frameworks and methods should allow the genomic data owners such as research organizations or hospitals to utilize external services such as low-cost cloud infrastructure for their sensitive records and offshore their computational requirements to a third-party provider. Thus, the objective is to protect the confidentiality of the underlying data while performing computation. However, privacy breaches can still happen through the computation outputs. Research Objective 2 addresses these kinds of indirect inference attacks.

Objective 2: *Privacy preserving outputs from genomic data analysis.* Multiple re-identification attacks have demonstrated that outputs (*e.g.*, aggregate query results) can also cause inference attacks against the underlying participants. Over the last decade, these findings have amplified concerns over data analysis performed on genomic data (see Table 1.1). Even with the state-of-the-art privacy-preserving mechanisms around data storage and computations (Research Objective 1), the outputs need to be protected. These inference attacks against the aggregated results [21, 29] have surfaced recently, and there is a need to propose practical solutions to address these problems. In this thesis, we intend to answer some of these problems using differentially private mechanisms.

In Figure 1.1, we see the first research objective (in red) where we assume that the central server (*i.e.*, cloud) is untrusted. This server is acting as the primary storage for

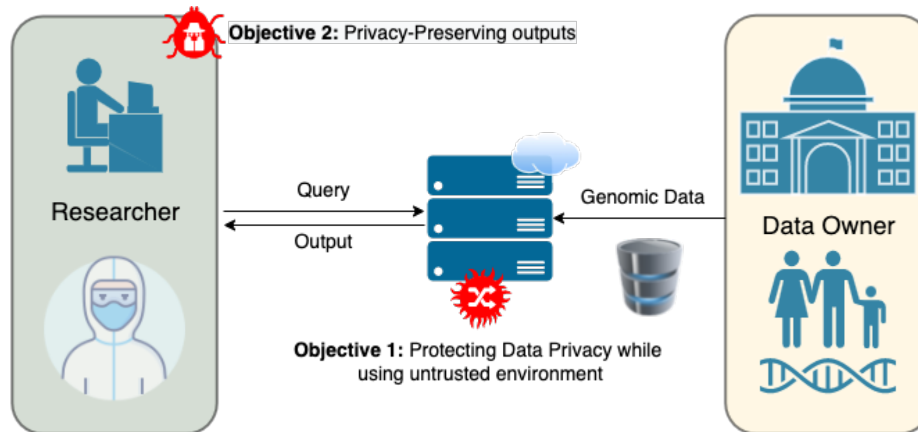


Figure 1.1: The two objectives considered in this thesis where the data owners outsource the underlying genomic data collected from different participants to a centralized environment (i.e., cloud). A researcher or data analyst performs his/her genomic queries or operations on this server in a privacy-preserving fashion.

genomic data where majority of the computations happen. The proposed solutions in this part (Part I) will propose novel privacy preserving methods using different cryptographic techniques that will be computationally efficient and provide a provable security guarantee. Here, the primary objective is to protect the genomic data from any breach which is becoming more frequent against health-care data systems [25]. It is noteworthy that several privacy-preserving techniques for genomic data storage and computations have been proposed as they are discussed and compared in the corresponding chapters.

On the other hand, for privacy, we consider the outputs generated from any analysis of the genomic data to be sensitive. For example, the result from a statistical analysis reveals the presence of an individual on a particularly sensitive dataset. Now, privacy for individuals or certain groups in a genomic dataset can be achieved with different techniques. We target two of such settings in this thesis, taking a look at the problem from two completely separate angle. In the first approach, we provide privacy over the outputs generated from any genomic data analysis (Chapter 6). The second attempt (Chapter 7 takes a more generalized approach to protect the privacy of the individual participants providing a holistic view of the problem while performing federated machine learning.

1.3 Contributions

As we consider two threats, one from the computational environment and the other one posed by the performed analysis or results, we divide the topics accordingly into two parts. We summarize the core contributions of our research below:

- We propose several string index based on prefix and suffix trees for large-scale genomic datasets that allows different search operations and privacy-preserving queries (Chapters 3 and 4).
- We implemented a GPU-parallel fully homomorphic encryption framework along with several optimizations to perform genomic search operations under encryption (Chapter 5).
- We present two different differentially private [30] methods to protect the output privacy from any genomic data analysis: a) Online algorithms to release private statistics (Chapter 6) and b) Private Federated learning mechanisms (Chapter 7).

The individual contributions in these parts are described in the following section.

1.3.1 Part 1: Protecting Data Privacy

The underlying chapters in this part discuss the privacy of genomic data in an outsourced environment and propose several indexing and cryptographic techniques to adhere the issues:

1.3.1.1 Privacy-Preserving Indexing and String Queries using Generalized Suffix Trees

Due to the massive size and scale of real-world genomic data, different indexing techniques and data structures have been proposed to handle these dataset. We investigate one such technique: Generalized Suffix Tree (GST) to execute secure queries on genomic data. In this work, we introduce an efficient parallel generalized

suffix tree construction algorithm that is scalable for arbitrary genomic datasets. Our construction mechanism employs shared and distributed memory architecture collectively while not posing any fixed, prior memory requirement as it uses external memory (disks). We also proposed privacy-preserving techniques to further index and outsource the genomic data into an untrusted cloud server. The experiments on different datasets and parameters on a realistic cloud environment exhibit the scalability of the execution time of these methods on different string queries as they also perform better than the state-of-the-art method [31]. The results of this chapter are published in the 7th International Conference on Algorithms for Computational Biology, 2020 [2] and an extended version is under revision on the journal, BMC Genomic Data.

1.3.1.2 Large-Scale Genome Data Storage and Query with Privacy-Preserving Techniques

In this work, we propose a novel, privacy-preserving mechanism to support secure count queries on an graph database (Neo4j) and evaluated the performance on a real-world dataset of 735,317 Single Nucleotide Polymorphisms (SNPs). In particular, we propose a new tree indexing method that offers constant time complexity (proportion to the tree depth), which was the bottleneck of existing approaches. The proposed method significantly improves the runtime of query execution compared to the existing techniques. It takes less than one minute to execute an arbitrary count query on a dataset of 212 GB, while the best-known algorithm takes around 7 minutes. The results of this chapter are published in the Elsevier Computer Methods and Programs in Biomedicine [3] (impact factor 3.8).

1.3.1.3 Genomic String Search using Parallel Fully Homomorphic Encryption

In this work, our objective is to improve the performance of Fully Homomorphic Encryption (FHE) schemes by designing an efficient parallel framework. We first extended the gate operations from an existing scheme, TFHE [32], to algebraic circuits

such as addition, multiplication, and employ Graphics Processing Units (GPU) to parallel the cryptographic operations. Then, we applied the GPU-parallel FHE framework into a much needed genomic data operation: String Search. Specifically, we employed the most popular string distance metrics (hamming distance, edit distance, set maximal matches) to capture the difference between multiple genomic sequence. The parallel FHE framework is published in the IEEE International Symposium on Hardware Oriented Security and Trust (HOST) 2020 [4] while the extension is submitted to the Elsevier Journal of Information Security and Applications.

1.3.2 Part 2: Protecting Output Privacy

Parallel to Part 1.3.1, another research objective in this thesis is to ensure that no inference attacks are possible using the output from a genomic data analysis. In this part, we will also rely on 'Differential Privacy' [30] as the core technique (detailed in Section 2.2.3) against these inference attacks.

1.3.2.1 Online Algorithm for Differentially Private Genome-Wide Association Studies

Due to the private nature, we investigate an approach that only releases the in-between outputs or statistics from genomic data rather than publishing the whole dataset. The work proposes a generalized Differentially Private mechanism for Genome-wide Association Studies (GWAS) based on private statistics. Our methods provide a quantifiable privacy guarantee that adds noise to the intermediate outputs but ensures satisfactory accuracy of the private results as well. Furthermore, the proposed method offers multiple adjustable parameters that the data owners can set based on the optimal privacy requirements. These variables are presented as equalizers that balance between the privacy and utility of the GWAS. The method also incorporates *Online Bin Packing* technique [33], which further bounds the privacy loss linearly, growing according to the number of open bins and scales with the incoming queries. Finally, we implemented and benchmarked our approach using *seven* different GWAS studies to test the performance of the proposed methods.

The experimental results demonstrate that for 1000 arbitrary online queries, our algorithms are more than 80% accurate with reasonable privacy loss and exceed the state-of-the-art approaches on multiple studies (*i.e.*, EigenStrat, LMM, TDT). This chapter is published in the ACM Transactions on Computing for Healthcare.

1.3.2.2 Differentially Private Federated Learning on Genomic Data

In this work, we propose a generalized gene expression data sharing method for Federated Machine Learning using a differentially private mechanism. Due to the large number of genes available, the data dimension is also reduced to accommodate smaller privacy budgets as we utilize an exponential mechanism to create a private histogram from numeric expression data. The output histogram can be used in any federated machine learning setting having multiple data owners. The proposed solution was submitted to genomic data security and privacy competition, iDash 2020 where it ranked third among 55 teams. This work is recently accepted in the Elsevier Journal of Biomedical Informatics.

1.4 Organization

This thesis is organized as follows:

- Chapter 2 explores some of the necessary background which are utilized by the different methods proposed in this thesis.
- Chapter 3 presents our work on parallel suffix construction and privacy-preserving Techniques query execution.
- Chapter 4 discusses a scalable and secure graph database solution using prefix trees for count queries.
- Chapter 5 demonstrates a GPU accelerated Fully Homomorphic Encryption (FHE) framework and employed in secure string search operations over genomic data.

- Chapter 6 provides online methods for differentially private Genome-Wide Association Studies.
- Chapter 7 presents our methods on private federated learning algorithms for gene expression profiles.
- Finally, Chapter 8 concludes the thesis discussing the potential future works.

Chapter 2

Background

In this chapter, we discuss the different entities and computational settings for genomic data. We also discuss the threat model and security assumptions for general architecture. Finally, 2.2 overviews the privacy-preserving techniques utilized in this thesis.

2.1 General Architecture

In this section, we specify the possible entities and the computational architecture along with the considered threat model. We give a brief overview of the mathematical notations used in the thesis in Table 2.1.

2.1.1 Entities in Genomic Data Computations

In a genomics study, the collaborating parties can be categorized into four general entities: a) end-point users or researcher layer, b) computation layer, c) data owner layer, and d) data storage and operations layer (Figure 2.1). Often times these entities can be merged together or generalized according to their individual objective. For example, data owners can have their own infrastructure to store large quantities of genomic data or their own data storage layer. Also, there are several proposals for introducing a fully trusted entity as well. Regardless of any alteration to the

Table 2.1: Mathematical Notations used in the thesis along with their description

Notation	Description
$\mathbb{Z}$	set of integers
$\mathbb{B}^n$	set of n bits
$\mathbb{N}$	set of non-negative integers
$\mathbb{Z}_q$	set of integers modulo q
$\mathbb{R}$	ring where $\mathcal{R}[X]$ denotes the coefficients
$\mathbf{v}$ or $\overrightarrow{v}$	vector representation
$\mathbf{M}$	upper-case bold letters for matrix representation
$\langle \mathbf{a}, \mathbf{b} \rangle$ or $\mathbf{a} \cdot \mathbf{b}$	inner product of matrix $\mathbf{a}$ and $\mathbf{b}$ $(a_1 b_1 + \ldots + a_n b_n)$
$\mathbf{v} \otimes \mathbf{w}$	tensor (Kronecker) product of two vectors $\mathbf{v}, \mathbf{w}$, $(v_i, w_j)_{i,j}$

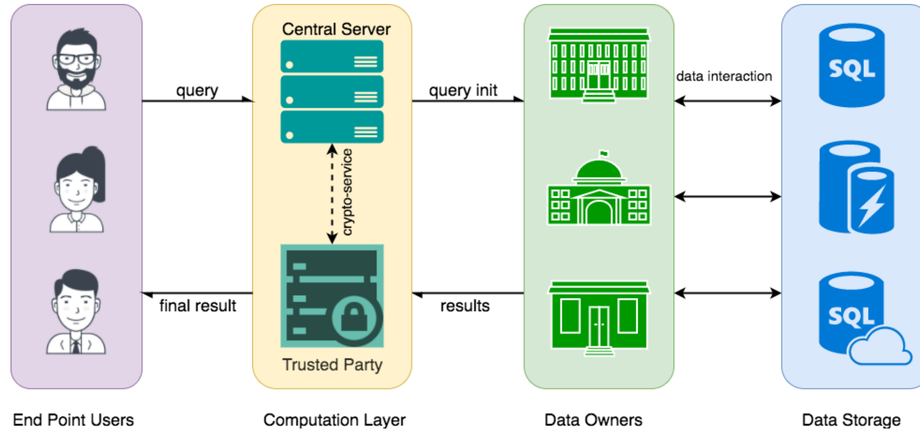


Figure 2.1: Different entities involved in a genomic data computation

structure, these entities are often assigned with different trust models (*i.e.,* malicious, semi-honest or fully trusted).

2.1.2 Problem Architecture

Two different architecture: *Centralized* and *Federated* are present in different literature and real-life use cases as it will mostly denote whether the genomic data is stored in one central location or not. In this thesis, we will pursue both the centralized and federated models for the underlying privacy problems. For example, in Figure 2.2, all the data owners from different geographical locations submit their data to a central server through which researchers can execute their queries. Here,

the genomic data is shared with some degree of privacy protection as transferred to the central server. Therefore, the server cannot leak any additional information about the participating individuals without the secret key. This approach is more popular as it reduces or often mitigates the individual data owners (*i.e.,* hospitals, clinics and research organizations) from auxiliary costs involved in managing such magnitude of data. Also, one single entity can manage the access control compared to a distributed credentialing system. In the federated model, the data never leave individual data owners premises, only in-between results or statistics are shared.

Sometimes another trusted party, Crypto Service Provider (CSP) is added which is solely responsible for generating the cryptographic keys required by the protocols. Here, each data owner receives a key from CSP and uses it to encrypt its local result of the query. CSP also sends the public key to the central server to perform computation on the encrypted results accumulated from data owners. The secret key is sent to the researchers who need it for final decryption.

2.1.3 Threat or Adversary Models

In general, the goal of this thesis is to ensure the external or third parties (*i.e.*, central server, CSP etc.) learn nothing about the genomic data or the individuals beyond what is released by the final query results. Formally, in a secure computation, there are three different threat or adversary models present in literature: a) Malicious, b) Semi-Honest, c) Honest or Trusted. As the name suggests, the level of trust is the least when considering malicious parties whereas honest parties are the most trusted. Semi-Honest parties (also known as honest-but-curious) lie in the middle which follow the underlying protocol but try to gain more information than allowed from the computations or its in-between outputs [34].

Semi-honest adversary models which often realistic external computation or outsourcing settings. Here, different organizations are collaborating to securely store, share or compute on their data for scientific and social benefits. For example, a research organization can use commercial cloud services like Amazon AWS for storage and computation. Therefore,these service providers have no intention to act

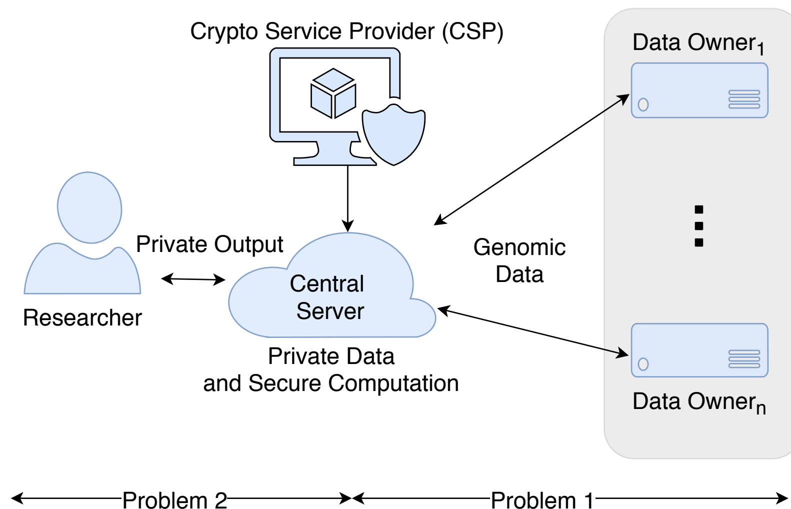


Figure 2.2: Centralized architecture for sharing and compute on private genomic data

maliciously due to the contract and service agreements. As a result, the data owners, the cloud server, nor the CSP has any motivation to maliciously behave and alter the mechanism in the hope of producing incorrect outputs.

However, the underlying computational environment (*i.e.*, central server) can get compromised by internal (employees) or third parties which is not uncommon at present. Data breaches [35] often cost millions in fine [36] and damages hard-earned reputation. We consider such compromised servers to be malicious and will try to infer sensitive data as much as possible.

2.2 Privacy-Preserving Techniques

In this section we discuss the different cryptographic techniques we will use for secure computations or storage of genomic data. Some of the seminal developments in this ecology of privacy preserving techniques and genomics are shown in Figure 2.3 [27]. From the earlier sequencing techniques in 1975 to the very recent developments using Intel SGX in 2017, the genomic data evolution and the cryptographic techniques are presented in a chronological fashion in Figure 2.3. We use the green color and the orange color to describe the contributions in genomic data and privacy preserving techniques, respectively.

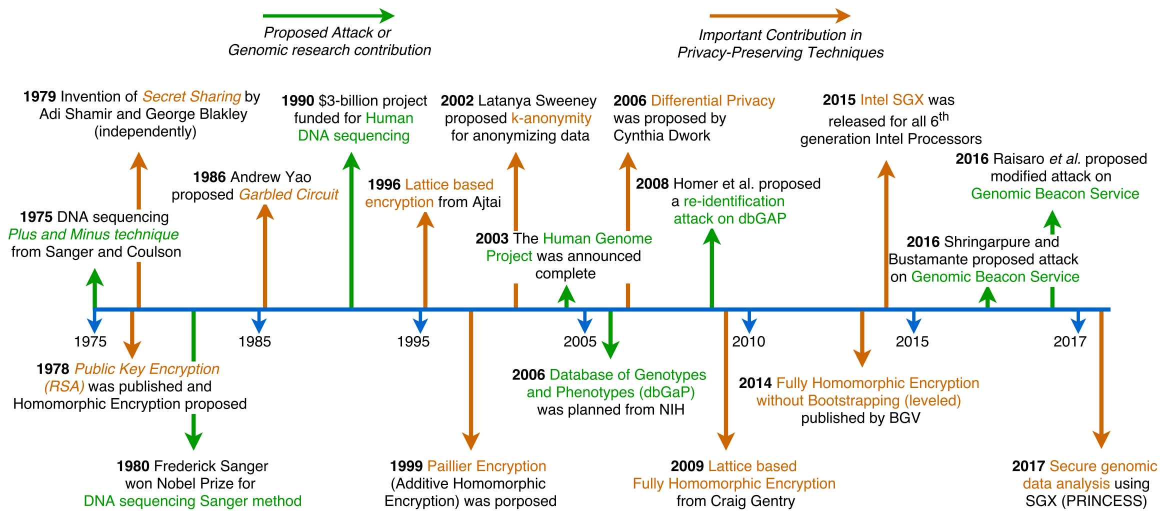


Figure 2.3: Timeline of the evolution of genomic data studies and seminal development of different privacy preserving techniques

2.2.1 Homomorphic Encryption

Homomorphic Encryption (HE) allows one party to compute arbitrary functions over encrypted data without ever decrypting it (Figure 2.4). Though encryption and computation in conjunction might sound contradictory, it is an active area of research in the crypto community. Though, the scheme was defined soon after RSA in 1978 [37], it remained mostly theoretical until the breakthrough from Gentry [38] in 2009.

HE is an important privacy-preserving technique due to its ability to keep the data under encryption at all time. For example, a genomic data owner can encrypt the whole dataset and store it on an untrusted environment. The computations can be performed under HE schemes and the final result is send back to the owner for decryption. Throughout the data life-cycle, it is never kept in plaintexts (except data owner) which protects the privacy of the participants.

2.2.1.1 Categories and Implementations

Homomorphic Encryption techniques are often categorized according to their generalizability of computation. A *Fully Homomorphic Encryption* (FHE) can compute any function any number of times given a set of encrypted inputs. This

Table 2.2: Comparison of popular HE implementations

Feature	**HElib**	**SEAL**	**FV-NFLlib**	**TFHE**
Crypto scheme	BGV [45]	FV [46]	FV [46]	BGV [45]
Fully HE	✓	✗	✗	✓
Language	C++	C++	C++	C++
Library dependency	NTL & GMP	✗	NFLlib	Any FFT
Relinearization	✓	✓	✓	✗
Bootstrap	✓	✗	✗	✓
Fixed-point support	✗	✓	✗	✗
GPU enabled	✓[47]	✗	✗	✓[4]
Wrapper available	Python	C#	✗	✗

is the most applicable form of HE as it offers an arbitrary number of computations. However, the current FHE schemes are computationally expensive and not applicable in any real-life security applications.

Another important category is the *SomeWhat Homomorphic Encryption* (SWHE) which guarantees that a specific circuit will evaluated and will provide the correct results upon decryption. If the depth of the circuit (generally the number of the multiplications) is given as a parameter, SWHE are extended to Leveled Homomorphic Encryption. SWHE schemes are more applicable as as there are multiple computationally efficient SWHE schemes available [39, 40]. Regardless of the definitions, FHE, SWHE (or LHE), all supports additions and multiplications over encrypted data. Partial HE is the primitive and third category which allows only additions *or* multiplications (*i.e.*, Paillier [41], RSA[37]).

Regardless of the recent theoretical breakthroughs for FHE, it is still inefficient for general computation. Recent SWHE schemes are quite popular since they offer faster execution time compared to FHE for a limited computational depth [39]. However, some recent endeavours [42, 43, 44] are making FHE schemes faster and more applicable to a real-world problem scenario. Notably, we will utilize FHE over other schemes since it allows arbitrary number of computations without introducing any errors.

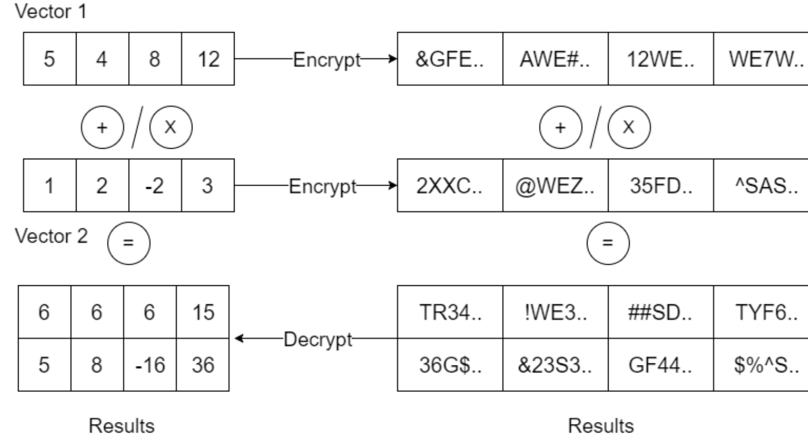


Figure 2.4: Example of homomorphic operations on encrypted values

Table 2.3: A comparative analysis of existing Homomorphic Encryption schemes for different parameters on 32-bit number where Size is in kilobytes and Additions, Multiplications time in milliseconds

	Year	Homomorphism	Bootstrapping	Parallelism	Bit security	Size	Add. (ms)	Mult. (ms)
RSA [48]	1978	Partial	×	×	128	0.9	×	5
Paillier [49]	1999	Partial	×	×	128	0.3	4	×
TFHE [32]	2016	Fully	Exact	AVX [50]	110	31.5	7044	4,89,938
HEEAN [51]	2018	Somewhat	Approximate	CPU	157	7,168	11.37	1,215
SEAL (BFV) [52]	2019	Somewhat	×	×	157	8,806	4,237	23,954
cuFHE [53]	2018	Fully	Exact	GPU	110	31.5	2,032	1,32,231
NuFHE [54]	2018	Fully	Exact	GPU	110	31.5	4,162	1,86,011
Cingulata [55]	2018	Fully	Exact	×	110	31.5	2,160	50,690
Our Method	2020	Fully	Exact	GPU	110	31.5	1,991	33,930

2.2.1.2 Current Techniques in Fully Homomorphic Encryption

The homomorphic encryption schemes can be divided into three major categories: Partially, Somewhat, and FHE schemes. Partially Homomorphic schemes only support one type of operation (e.g., addition or multiplication); such schemes are not useful in performing arbitrary computations on encrypted data.

Somewhat Homomorphic Encryption (SWHE) schemes are more equipped than partially homomorphic encryption schemes. These schemes support both addition and multiplication operations on encrypted data, but for a limited (or pre-defined) number of times. In addition, these schemes are relatively efficient (see Table 5.8 for comparison) and therefore are practical for certain applications. However, even these schemes require complex parameterization and are not powerful enough for

complicated applications such as deep learning.

FHE schemes support both addition and multiplication operations for an arbitrary number of times. This property allows computing any function on the encrypted data. Both SWHE and FHE use the Learning with Error (LWE) paradigm, where an error is introduced with the ciphertext value to guarantee security [56]. This error grows with each operation (especially multiplication) and causes incorrect decryption after a certain number of operations. Therefore, this error needs to be minimized to support arbitrary computation. The process of reducing the error is called Bootstrapping. FHE employs bootstrapping after a certain number of operations resulting in higher computation overhead, while SWHE provides faster execution time by limiting/pre-defining the number of operations on the encrypted data.

The above discussion provides an intuition about the applications of different HE schemes. That is, SWHE is better suited for the applications where the computational depth is shallow and known (/fixed) prior to the computations [57]. However, these schemes are not suitable for applications that require arbitrary depth like deep learning. In order to compute complicated functions like deep learning, the researchers have proposed alternative models that require the existence of a third party [58, 59, 60]. The aim is to minimize the propagated error without executing the costly bootstrapping procedure for SWHE schemes. However, such an assumption (*i.e.*, the existence of a trusted third party) is not always easy to fulfill. In this thesis, we assume that the computational entities (*i.e.*, cloud server) is often standalone, and we show that parallel operations can be used to lower the cost of FHE instead of relaxing the security assumptions for the computation model.

2.2.2 Garbled Circuit

A Garbled Circuit (GC) is a constant round privacy-preserving technique which allows arbitrary function to be computed between multiple parties, hiding both their inputs from each other. This concept was first defined by Yao [61] in 1982. An example is shown in Figure 2.5 of a GC for a simple AND function with two secret input bits I_1

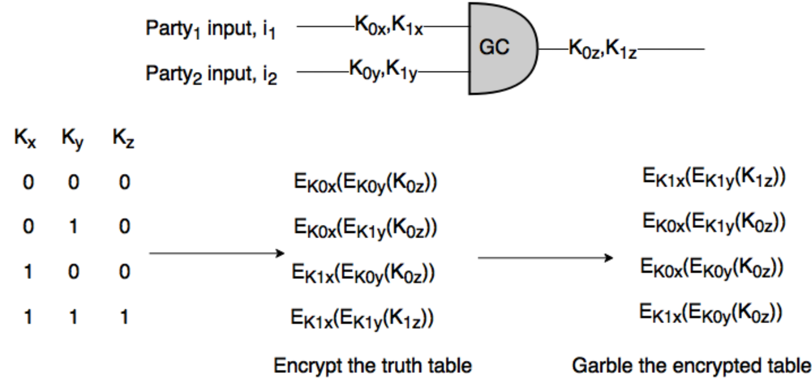


Figure 2.5: Garbled Circuit Execution

and I_2. We give an example of an application of GC with 'The Millionaire Problem':

Example 2.2.1 (Millionaire Problem). Two individuals want to determine who is wealthier than the other but do not want to reveal their exact net worth. They initiate a GC protocol between them that results in a boolean value which denotes whose wealth is greater than the other. One party generates the whole circuit (generator) for the computation and keys whereas the other one evaluates the underlying comparison function (evaluator).

2.2.2.1 Garbled Circuit Mechanism

In this section, we describe the details of a GC execution protocol. Suppose there are two parties p_1 and p_2 with inputs i_1 and i_2, respectively. They want to compute some arbitrary function $f(i_1, i_2)$ and know the result, but do not want to reveal the inputs i_1 and i_2. Ideally, only the output from $f(i_1, i_2)$ will be public while their inputs remain hidden from each other. Formally, the inputs to the function are boolean along with the output as well, $f : \{0,1\}^n \rightarrow \{0,1\}$ and $i_1, i_2 \in \{0,1\}^n$. For simplicity, the `AND` gate only has two bits of input, therefore $n = 1$. Here, functions are often named as *Circuits* (or Boolean Circuits) due to the construction solely using boolean gates.

In a GC protocol, the parties perform special duties to evaluate $f(i_1, i_2)$ to maintain the secrecy of the inputs where one party generates the circuit f while the other party evaluates f. Lets assume p_1 is the generator who constructs the circuit

for $f(i_1, i_2)$ with boolean gates and encrypts it. Here, each value on the truth table is encrypted with different keys $(K_{0x}, K_{1x}, K_{0y}, K_{1x})$ as shown in Figure 2.5. Afterwards, p_1 randomly shuffles the four encrypted values and sends them (encrypted `AND` circuit) along with K_{i_1x} to p_2 for evaluation.

The evaluator p_2 needs the other set of keys K_{i_2y} where $i_2 \in \{0, 1\}$ to execute the circuit. For example, if p_2's input $i_2 = 0$, then it will need the key K_{0y} from p_1, but it does not want to reveal i_2 to p_1. This is done using Oblivious Transfer (OT) protocol (described next) [62]. After receiving the corresponding $K_{i_2}y$'s from p_1 with OT, p_2 decrypts the encrypted `AND` circuit with $K_{i_1}x$ and K_iy. Finally, p_2 will send all four decrypted value to p_1 which matches whether the final output is K_{0z} or K_{1z}, representing 0 and 1, respectively.

2.2.2.2 Oblivious Transfer

In plaintext, Oblivious Transfer (OT) is a protocol that allows a party p_2 to pick an element from a list of elements residing in another party p_1 without p_1 having no knowledge about it. Formally,

Definition 2.2.1 (Oblivious Transfer)**.** If p_1 has a list of n elements $(x_0, \ldots, x_{n-1})$ where p_2 wants to select the i-th element $(i \in \{0, \ldots, n-1\}$, an OT protocol will only output x_i to p_2 while the value of i is hidden from p_1.

This is known as the `1-out-of-n` OT protocol as it can be reduced to `1-out-of-2` where $n = 2$. Though there are many different methods to achieve the aforementioned OT definition [63, 62, 64, 65], we give a simple example based on Diffie-Hellman-Merkle key exchange.

Suppose p_1 and p_2 agrees on a prime p and a base $g \in \mathcal{G}$ (primitive root modulo p) where $\mathcal{G}$ is a cyclic group. p_1 generates a random number $a \leftarrow \mathcal{G}$ and sends $A = g^a$ to p_2. Based on the input $i \in \{0, 1\}$, p_2 generates $B = \{g^b, Ag^b\}$ where $b \leftarrow \mathcal{G}$ and sends it to p_1.

Upon receiving B, p_1 performs a hash operation $(H : \mathcal{G} \rightarrow \{0, 1\}^n)$ and creates two keys $k_0 = H(B^a)$ and $k_1 = H((B/A)^a)$. These keys are utilized to encrypt the two elements x_0, x_1 and sent to p_2. Hence, p_2 receives two encrypted messages $E_{k_0}(x_0)$

and $E_{k_1}(x_1)$. Before decrypting the two ciphertexts, p_2 uses the same hash function to generate its own key $k_i = H(A^b)$. With k_i, p_2 will try to decrypt both $E_{k_0}(x_0)$ and $E_{k_1}(x_1)$. However, only one decryption will be successful here as if $i = 0$ then p_2 will know x_0.

Undergoing much optimizations of the original protocol throughout the years [66], currently there are several implementations available: ObliVM [67], FastGC [68], TinyGC [69] and EMP-Toolkit [70].

Use Case. Conventionally GC is employed on a 2-party setting where each party have a strict requirement of not revealing their data or input to the computation. Also, the final output should not be revealed publicly, but only to the two parties. GC has seen real-life applications, specially on the Secure Multi-Party Computation (SMPC) settings [71, 72, 73]. However, the major drawback of this technique is the network communication which is proportional to the computational circuit depth. In other words, the communication overhead increases along with the complexity of the functions. However, this is quite realistic as communication costs are much cheaper than storage these days.

2.2.3 Differential Privacy

Differential privacy (DP) was proposed around 2006 [30, 74] and since then it has become a gold standard of privacy [75]. It can theoretically offer quantifiable bound of privacy on the disclosure of data or any query result. Formally,

Definition 2.2.2 (ϵ-Differential Privacy)**.** A randomized algorithm $\mathcal{A}$ is a differentially private over a set of neighbouring databases $\mathcal{DB}$ and $\mathcal{DB}'$ where $\mathcal{DB}$ is different from $\mathcal{DB}'$ in at most in one record (*i.e.,* $\mathcal{DB} \Delta \mathcal{DB}' \leq 1$) and for all possible databases $\hat{\mathcal{DB}}$,

$$\mathcal{P}[\mathcal{A}(\mathcal{DB} = \hat{\mathcal{DB}})] \leq e^{\epsilon} \mathcal{P}[\mathcal{A}(\mathcal{DB}' = \hat{\mathcal{DB}})]$$

Here, ϵ is a *privacy budget* which regulates the amount of noise allowed in the output (informally). This parameter $\epsilon > 0$ is (usually) public and predefined by the data

owner or publisher. Lower values (i.e., $(0, 0.5]$) of ϵ denotes the ratio of the probability space of $\mathcal{A}(\mathcal{DB}_1)$ and $\mathcal{A}(\mathcal{DB}_2)$ being almost the same ($\epsilon(0, 0.5) \sim 1$) which refers stringent privacy. Thus, larger ϵ values (i.e., $(ln(2), ln(4)]$) will loosen the privacy guarantee (less noise) but improve the accuracy of the analysis [76].

There is an alternative definition available with an additional factor δ (Definition 2.2.3). Traditionally, the value for δ is set less than the inverse of any polynomial in the size of the dataset $|\mathcal{DB}|$.

Definition 2.2.3 ((ϵ, Δ)-Differential Privacy)**.** A randomized algorithm $\mathcal{A}$ is a differentially private over a set of neighbouring databases $\mathcal{DB}$ and $\mathcal{DB}'$ where $\mathcal{DB}$ is different from $\mathcal{DB}'$ in at most in one record (*i.e.,* $\mathcal{DB}\Delta\mathcal{DB}' \leq 1$) and for all possible databases $\hat{\mathcal{DB}}$,

$$\mathcal{P}[\mathcal{A}(\mathcal{DB} = \hat{\mathcal{DB}})] \leq e^{\epsilon}\mathcal{P}[\mathcal{A}(\mathcal{DB}' = \hat{\mathcal{DB}})] + \delta$$

Definition 2.2.3 is also known as *Approximate DP* as we get the same privacy guarantee as *Pure DP* (Definition 2.2.2) with probability $1 - \delta$. However, with probability δ, the randomized mechanism will not provide any privacy which is why the value of δ is kept small. For example, δ in order of $1/|\mathcal{DB}|$ will be critical as it will reveal the records coming from a small number of participants. In short, δ provides an additional factor in favor of the randomized mechanism $\mathcal{A}$ picking $\mathcal{DB}'$ over $\mathcal{DB}$.

Here, we will consider real and linear functions $f : \mathcal{DB} \rightarrow \mathbb{R}^d$ which narrates a function f being executed on $\mathcal{DB}$ and mapping them to a set of d real numbers $\mathbb{R}$. These can be referred to as the queries we perform in the dataset which is a fundamental operation. Thus, functions and queries are used interchangeably and can be visualized as database operations (*i.e.*, count, max, etc.).

2.2.3.1 Sensitivity

Sensitivity of the underlying function f is an important property in differential privacy. It captures the effect of f on the different rows of $\mathcal{DB} = \{r_1, r_2, \ldots, r_n\}$. For example, every record r_i is iterated for a count query to check whether it satisfies

the query conditions, $f_{count}(\mathcal{DB}) = \sum_1^n cond(r_i)$ where $cond(r_i) \in \{0,1\}$. Hence, l_1 sensitivity can be defined as,

Definition 2.2.4 (l_1-sensitivity)**.** Sensitivity for any real valued function $f : \mathcal{DB} \rightarrow \mathbb{R}^d$ is defined as,

$$\Delta f = \max_{\mathcal{DB}_1, \mathcal{DB}_2} ||f(\mathcal{DB}_1) - f(\mathcal{DB}_2)||_1$$

The aforementioned definition shows the sensitivity of one record in $\mathcal{DB}_1, \mathcal{DB}_2$ since $||\mathcal{DB}_1 - \mathcal{DB}_2|| \leq 1$ according to definition 2.2.2. For example, one record in a count query can only effect f_{count} by 1 (absent/present); hence $\Delta f_{count} = 1$. However, for maximum (or minimum, average, median etc.) queries, the sensitivity will not be 1, since one record can change the output measurably. In a nutshell, Δf provides the data publisher an estimation of the upper bound on the noise to be added to protect privacy of the data under the queries f. We can also define an $l2$-sensitivity with $\max_{\mathcal{DB}_1, \mathcal{DB}_2} ||f(\mathcal{DB}_1) - f(\mathcal{DB}_2)||_2$ which is based on the $l2$ distance which is less than or equal to $l1$-sensitivity.

2.2.3.2 Privacy Compositions

One of the most important property of a DP algorithm is the privacy compositions. Specifically, there are two separate composition theorem available that allows us to bound the privacy loss or budget ϵ:

Definition 2.2.5 (Sequential Composition [77])**.** For any arbitrary dataset $\mathcal{DB}$ and $\epsilon > 0$, the privacy loss from any differentially private algorithm for k queries on will be $\sum_{i=1}^k \epsilon$.

Definition 2.2.6 (Parallel Composition [78])**.** For k different queries on disjoint datasets, $\{\mathcal{DB}_1, \dots \mathcal{DB}_k\} \in \mathcal{DB}$, k differentially private algorithms each with $\{\epsilon_1, \dots, \epsilon_k\} \in \epsilon$ will incur a total privacy loss of $\max(\epsilon))$.

The fundamental difference between sequential and parallel composition is the accumulated privacy cost. For example, if we employ a ϵ-DP algorithm for k queries subsequently, we will accrue $k\epsilon$ privacy budget. In other words, the randomized mechanism will be $k\epsilon$ differentially private for the total k number of operations.

However, if we can partition the dataset into k individual splits for each query and ensure that there are not overlapping records then we can utilize parallel composition. It is a more efficient way to manage the privacy budget ϵ compared to Definition 2.2.5. For example, we can spend larger *epsilon* (or budgets) for queries that will positively impact the analysis and spend less otherwise. However, partitioning the dataset can prove to be tedious as no single individual can appear into two partitions, in which case the privacy loss will rely on the sequential composition.

2.2.3.3 Laplace Mechanism

Laplace Mechanism ($\mathcal{A}_{\mathcal{L}}$) depends on the Laplace distribution which has a probability density function $Lap(x|\lambda) = \frac{1}{2\lambda} \exp\left(\frac{-|x|}{\lambda}\right)$ with variance $2\lambda^2$. Plainly, Laplace Mechanism $\mathcal{A}_{\mathcal{L}}$ executes a function $f(\mathcal{DB})$ and adds a noise from the independent and identically distributed (i.i.d.) Laplace distributions:

$$f(\mathcal{DB})' = f(\mathcal{DB}) + Lap(\lambda), \tag{2.1}$$

where, $Lap(\lambda)$ simply denotes the random noise. Here, λ is computed from the aforementioned sensitivity of the functions Δf (Δ hereafter) and privacy budget ϵ as, $\lambda = \Delta f/\epsilon$.

Theorem 2.2.1 (($\epsilon, 0$) Differential Privacy). For any numeric function $f : \mathcal{DB} \rightarrow \mathbb{R}^d$, DP algorithm $\mathcal{A}_{\mathcal{L}}$ that adds i.i.d. random noise from $Lap(\lambda)$ where $\lambda = \Delta f/\epsilon$ to each $f(\mathcal{DB})$ satisfies $(\epsilon, 0)$ differential privacy.

2.2.3.4 Exponential Mechanism

Apart from adding noise to numerical answers (Laplace Mechanism), there is another algorithm, Exponential Mechanism [79] in differential privacy literature that has been effectively used with lower ϵs to answer a query. For example, mere addition of noise will be detrimental to applications where we operate on non-numeric outputs (*i.e.*, online bidding, recommendation system etc.). Here, for a possible input, we calculate the probability for all possible outputs $o \in \mathcal{O}$ with respect to a utility function:

Theorem 2.2.2 (Exponential Mechanism)*.* For any numeric utility function u : $(\mathcal{DB} \times \mathcal{O}) \rightarrow \mathbb{R}$, if a randomized mechanism $\mathcal{A}$ chooses an output o with probability proportional to $exp(\epsilon u(\mathcal{DB}, o)/2\Delta u)$ satisfies $(\epsilon, 0)$-differential privacy.

Here, the utility function $u : (\mathcal{DB} \times \mathcal{O}) \rightarrow \mathbb{R}$ assigns a real valued score to all outputs o as higher scores represent better utility. Therefore, exponential mechanism generates a probability distribution over $\mathcal{O}$ and outputs a class o with probability proportional to $exp(\epsilon u(\mathcal{DB}, o)/2\Delta u)$. Importantly, we get an output from a finite set whereas in Laplace (or Gaussian) mechanism acted differently.

Here, the sensitivity Δu is calculated with $max_{\mathcal{DB},\mathcal{DB}',o}|u(\mathcal{DB}, o) - u(\mathcal{DB}', o)|$. This assures that the final output from an exponential mechanism will represent a higher utility score compared to rest of the classes which can be useful for more accurate analysis.

Use Case. The major motivation behind using differentially private mechanisms are the theoretical privacy guarantee of the input and the output. Though it introduces some inaccuracies in the final result, applications where privacy of the data is more important might benefit from these concepts. However, as the accuracy rely on the privacy parameters of the DP algorithm, it can be properly tuned for a specific dataset and the underlying queries. For example, privacy issues like the Genomic Beacon Service [22, 21], Data Dissemination [80] might benefit from a differentially private solution. Also it can be combined with the aforementioned three cryptographic techniques to achieve privacy over the input and output.

Part I

Protecting Data Privacy on Untrusted Environment

Chapter 3

Privacy-Preserving Indexing and String Queries using Generalized Suffix Trees

In this part, we discuss the privacy of genomic data when outsourcing the computations into any untrusted environment. Notably, untrusted environments are not necessarily confined to cloud or external computing services but also can be any organizational infrastructure or personal computer with limited security. There have been several reports of data theft and leaks with internal and external forces at play when it comes to sensitive genomic data [36]. In this Chapter, we propose a suffix tree-based index on genomic data and show its efficacy while executing privacy-preserving string queries on untrusted environment.

3.1 Introduction

The scientific achievements in human genomics have advanced the understanding of different diseases and our well-being. It has also given us concepts like genomic (or personalized) medicine and genetic engineering which are slowly becoming a reality that seemed impossible a decade before. We are now capable of storing thousands of genome sequences from patients along with their medical records. Today, medical

professionals utilize these large-scale data to study associations or susceptibility to certain diseases [81].

The recruitment for different genomic research is increasing as the cost for genome sequencing is ever-reducing through technological breakthroughs in the last few years. This growth in genomic data has resulted in consumer products where companies offer healthcare solutions and ancestry search based on human genomic data (*e.g.*, Ancestry.com, 23AndMe.com). Interestingly, all these applications share one major operation: *String Search*. Informally, the string search denotes the presence (and locations) of an arbitrary query nucleotide sequence in a large dataset. The search results comprise the individuals who carry the same nucleotides in the corresponding positions. Thus, we can perceive the relation between an unknown sequence to pre-existing sequences with such search queries.

On the other hand, the suffix tree is proven useful for searching different patterns or arbitrary queries on genomic data [82]. However, their construction suffers from the *locality of reference*, as reported in the initial work [83]. The locality of reference denotes the memory accesses in the same locations within a short period while building the suffix tree. Moreover, it gets severe as suffix trees perform best when the tree (vertices and edges) completely fits in the main memory. Unfortunately, this is quite impossible with off-the-shelf implementations and large scale genomic data.

Privacy is also one of the important aspects of genomic data as multiple attacks have been proposed over the years [27]. Since our genomic data are unique and can potentially reveal our presence in sensitive datasets, they are usually kept under a curtain of regulations and stringent access control mechanisms that hinder scientific discoveries. Therefore, privacy-preserving query execution on a genomic dataset is an important research area that has attracted the cryptographic community in general. Specifically, the large-scale volume and the computational complexity have made these problems challenging as researchers want to adhere to the participant's privacy and also get a timely response from secure computations.

In this work, we initially construct a Generalized Suffix Tree (GST) in parallel, then propose privacy-preserving string query techniques on such indexing. It is important to note that building a suffix tree efficiently and in parallel is an well-

studied area and not our primary contribution. Instead, we target GSTs which can represent multiple genomic sequences [84]. Here, we employed two different memory architectures for our parallel GST construction: a) distributed and b) shared memory. In a distributed architecture, we utilized multiple machines with completely *separate main memory* systems interconnected within a network. On the contrary, these processors have several cores, which share the *same main memory*. These cores are employed in our shared memory model. Furthermore, we employ a data specific parallelism based on the fixed nucleotide set in this construction for the shared memory architecture. Finally, our GSTs are built on the external file system to remove the dependency for a sizeable memory requirement.

Our primary contribution is the privacy-preserving method for performing different string queries. The proposed method relies on a hash-based scheme combined with cryptographic primitives. With two different privacy-preserving schemes, we demonstrate that the proposed methods provide a realistic execution time for a large genomic dataset. We can summarize our contributions below:

- Initially, we lay the groundwork by proposing a parallel framework using the distributed and shared memory model to construct GST for a genomic dataset. We also utilized the external memory (or disks) since GSTs for large-scale genomic data require notable memory size, usually not available in a single machine.

- The primary contribution of this chapter is the privacy-preserving query execution techniques that incorporate a tree-based structure (Reverse Merkle Hash) that serves as a secure index to execute different string queries. We further extend this method with a more formal cryptographic primitive named Garbled Circuit [61].

- We test the efficiency of our GST and the privacy-preserving queries with multiple string searches. Specially, we analyze the parallel speedup in terms of dataset size, the number of processors, components of the hybrid memory architecture and different indexing.

- Experimental results show that we can achieve around *4.7 times* speedup compared to the sequential algorithm with 16 processors to construct the GST for a dataset with 1000 sequences, where each sequence has 1000 nucleotides.

- Our privacy-preserving query mechanism also demonstrates promising results as it only takes around 36.7 seconds to execute a set-maximal match in the aforementioned dataset. Additionally, we compared with a the state of the art solution utilizing Burrows Wheeler Transform [31] (under the same setting) which takes around 160.85 seconds giving us a 4$\times$ speedup.

3.2 Preliminaries

In this section we discuss the necessary background for the work including the underlying data, problem definitions and briefly overview the generalized suffix trees.

3.2.1 Haplotype Data

Our genomic data inside the chromosomes consist of four nucleotide values represented by A, T, G and C. However, each location of individual chromosome mostly have two possible values which are named as alleles and these locations are named bi-allelic. These bi-allelic genome data can be represented as a binary sequence containing 0's and 1's where 0 and 1 denotes the reference and observed allele, respectively. Multi-allelic such as tri-allelic sites are rare (around 2% in human genome [85]) as it can have three different alleles present at a single location.

Therefore, we consider the bi-allelic genomic data which is also called haplotype data, where each allele (or position) on the chromosome is inherited from a single parent. In other words, in one specific location, we can only perceive two variations for such a dataset; therefore, we utilize a binary representation. However, our proposed method is not limited to such binary representation and generalizable over any dataset with a fixed character domain (Table 3.1).

Formally, a S be a haplotype sequence with m alleles such as $S = s_1 s_2 \dots s_m$ over a fixed size alphabet $\sum \in \{0, 1\}$. A substring of S is another string $S : i^j =$

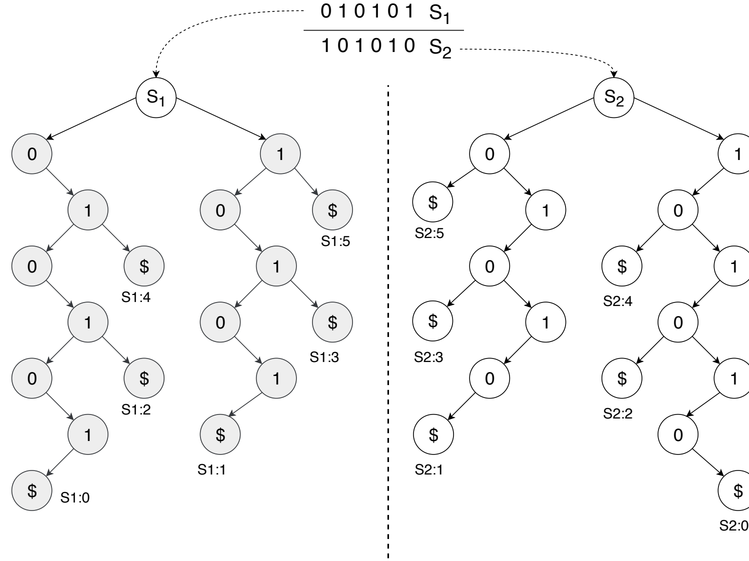


Figure 3.1: Uncompressed Suffix Tree (Trie) construction

$s_i s_{i+1} \ldots s_j$ where $1 \leq i \leq j \leq m$. A suffix is a specialized substring where $S : i = s_i s_{i+1} \ldots s_m$ with length $m - i$.

3.2.2 Generalized Suffix Tree

Suffix Trie and Tree:

Trie (from re**trie**val) is a data structure where each data point is placed in the vertex of a tree. Here, the edges represent the relation of one data to the other. In our problem scenario, each nucleotide of a sequence can be seen as the other data points or vertices of a Trie. We can differentiate two notable categories of Trie in literature: a) Prefix or b) Suffix. Specifically, we are interested in suffix tries in this work, though the proposed method is generalizable towards prefix trie/tree construction as well (§ 3.3.2).

Definition 3.2.1 (Suffix Trie [86])**.** a trie is a rooted tree where each node represents a symbol from the alphabet $\sum$ (except root) and no two sibling (children of the same node) share a common symbol.

Similarly, the Suffix Tries from sequence S is a rooted tree which contains all

possible suffixes of S at its leaf nodes. Notably, the memory requirements of the such suffix trie is significant as it needs $\mathcal{O}(m^2)$ for m characters. However, Suffix Trees are a compressed version of their Trie counterpart as it reduces the quadratic size to a linear one. For example, if a single vertex has only one child on a suffix trie, they are joined and denoted as a single vertex on the Suffix Tree. In Figure 3.1, we show two suffix tries `S1` and `S2` from sequences 010101 and 101010 respectively. For any sequence `S1`$=$ 010101, we consider all possible suffixes such as $[1, 01, 101, 01010, 10101, 010101]$ and construct the tree.

Definition 3.2.2 (Suffix Tree [87])**.** A Suffix Tree for an m character string S is a Trie (Definition 3.2.1) consisting of exactly m leaf nodes. Each node (except root and leaves) have at least two children where the edges represent a non-empty substring of S and no two sibling can have the same starting character. Finally, the m strings obtained from m leaf nodes should represent every suffix of $S : i$ where $i = 1, 2 \ldots, m$.

The suffix tree will also represent the end of the sequence with a special end character (`$`). For example, the suffix 01 (left Figure 3.1-`S1`) has an end character with label `S1:4` which denotes the sequence number and the start position of the suffix. Here, a node is labeled as `Sx:y` *s.t.* $x \in \{1, n\}$ and $y \in \{0, m-1\}$ for n sequences of m length.

Generalized Suffix Tree (GST): Generalized Suffix Tree is a collection of suffix trees (following Definition 3.2.2) constructed for multiple sequences. Here, we merge two suffix trees `S1` and `S2` from Figure 3.1 and construct `S12` in Figure 3.2. Fundamentally, there are no difference in constructing GST as we need to build individual suffix tree per sequence and merge them afterwards. Thus, the runtime for one GST construction depends on these suffix tree construction and size. For example, the traditional Ukkonen algorithm to build the suffix tree has a linear runtime $O(m)$ for m length sequences [88]. Therefore, n sequences with m characters will require $\mathcal{O}(nm)$ in the worst case.

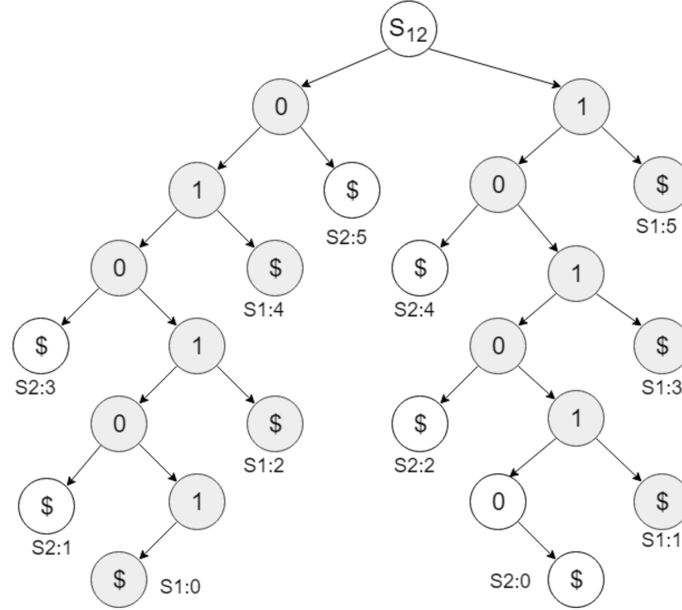


Figure 3.2: GST from Figure 3.1 where gray and white vertices are from `S1` and `S2`, respectively

3.2.3 String Queries q

We considered different string queries to test privacy-preserving methods proposed based on the GST and other cryptographic scheme (Section 3.3.3). The four queries discussed here are connected as they are challenging, incrementally. Nevertheless, the input to these queries will be a genomic dataset $\mathcal{D}$ consisting n individuals with m nucleotides. Since we are considering $n \times m$ haplotypes, $\mathcal{D}$ will have $\{s_1, \dots s_n\}$ records where $s_i \in [0,1]^m$. The query can be of arbitrary length ($1 \leq |q| \leq m$).

Definition 3.2.3 (Exact Match-EM)**.** For a genomic dataset $\mathcal{D}$ and an arbitrary query q, an exact match will only return the record (s) x_i which observe $q[0,m] = x_i[0,m]$ where m is the number of nucleotides available on each genomic sequence in $\mathcal{D}$.

Definition 3.2.4 (Exact Substring Match-ESM)**.** An exact substring match will only return the records x_i which observe $q[0,|q|-1] = x_i[j_1,j_2]$, where $q[0,|q|-1]$ denotes the full query and $x_i[j_1,j_2]$ is a substring of the record x_i given $j_2 \geq j_1$ and $j_2 - j_1 = |q|-1$.

Table 3.1: Sample haplotype data representation where $s_i \in \{0,1\}$ are the different positions on the same sequence

#	SNP_1	SNP_2	SNP_3	SNP_4	SNP_5	...	SNP_m
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
			$\vdots$				
n	0	1	0	1	0	...	1

Definition 3.2.5 (Set Maximal Match-SMM)**.** For the same inputs, a set maximal match will return the record x_i, which have the following conditions:

1. there exists some $j_2 > j_1$ such that $q[j_1, j_2] = x_i[j_1, j_2]$;
2. $q[j_1 - 1, j_2] \neq x_i[j_1 - 1, j_2]$ and $q[j_1, j_2 + 1] \neq x_i[j_1, j_2 + 1]$, and
3. for all $i' \neq i$ and $i' \in n$, if there exist $j_2' > j_1'$ $s.\ t.$ $q[j_1', j_2'] = x_i[j_1', j_2']$ then it must be $j_2' - j_1' < j_2 - j_1$.

Definition 3.2.6 (Threshold Set Maximal Match-TSMM)**.** For a predefined threshold t, the TSMM will report all records following the constraints from SMM (definition 3.2.5) and $j_2 - j_1 \geq t$.

A haplotype dataset X is presented in Table 3.1 which is of size $n \times m$. Exact queries (definition 3.2.3) with $q = \{1, 0, 0, 0, 1, \ldots, 0\}$ perfectly matches the first row x_i; hence the output set for this input q will be the first sequence in X. Now, for another $q = \{1, 1, 1\}$, the output for Query 3.2.5 (Exact Substring Match) should contain row 2 as q is present there as a substring. Similarly for $q = \{1, 1, 0, 1\}$ and Query 3.2.5, we will have the records $\{3, 4, n\}$ as outputs since they have $110, 101, 101$ substrings respectively.

3.3 Methods

As we fist build the GST in parallel prior to the private execution of different queries, the proposed methods are divided into two major components. However, the problem architecture are common and summarises the methods of this work:

3.3.1 Problem Architecture

The architecture consists of three entities: a) Data Owner, b) Cloud Server and c) Researchers as outlined in Figure 3.3. Here, the genomic dataset $D_{|n \times m|}$ is collected by the data owner where the researchers want to perform different queries q. The queries are dealt by an intermediary cloud server as the data owner generates the GST and stores it privately on the cloud. We assume that the researcher has limited computational power compared to the data owner since s/he is only interested in a minuscule portion of $\mathcal{D}$. Also, researcher have no interaction with the data owner as all query operations are handled by the cloud server. In summary, the proposed method presented in this article has two steps: a) constructing the GST in parallel, and b) executing q with a privacy guarantee over the data.

3.3.1.1 Parallel GST Construction

To construct the GST in parallel, firstly, the genomic data is evenly partitioned among different computing nodes by the data owner. Here, we consider two types of memory environment: distributed and shared. In the distributed memory, the machines are connected via network as they each have *mutli-core* processors and fixed-size memory (RAM). The multi-core processors on these machines collectively use the physical memory, which is called as shared memory. Hence, we have $|p|$ computing nodes that construct our desired GST jointly.

Our memory dispersion tackles one of the significant disadvantages of the GST construction: the sizeable memory requirement for longer sequences. For example, a thousand length sequence can create atleast thousand vertices, and n sequences can lead to an order of nm. Thus, for an arbitrary genomic dataset, it often outruns the

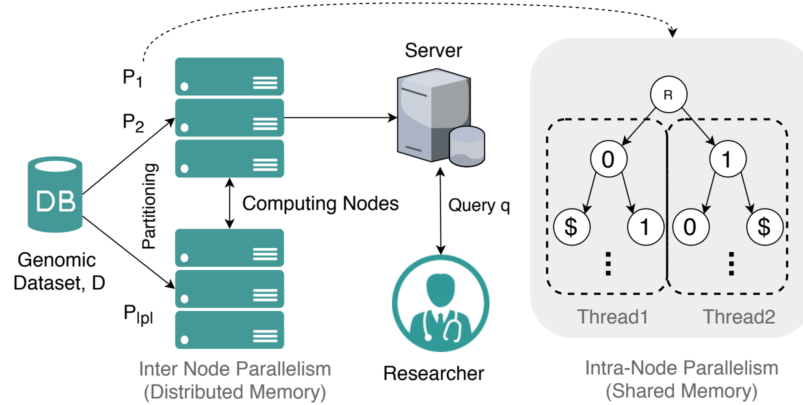


Figure 3.3: Computational architecture where data owner has the dataset $\mathcal{D}$ while a researcher submits query q

memory. Hence, this motivates us to construct our targeted GST in a *distributed memory setting*.

This leads to our proposed design, where we distribute the data (partition) and build the suffix tree separately in different computing nodes. These nodes can construct each sub-tree, which is later shared with the other nodes. These shared sub-trees are then merged, and the final tree includes all suffix sub-trees combining the outputs from all computing nodes (§ 3.3.2.4). The multiple processors in each node will also use a shared memory model while constructing and merging their individual GST in parallel (§ 3.3.2.2 and 3.3.2.3). Therefore our three design goals can be summarized as follows:

1. Partition the dataset for different nodes in a *distributed memory architecture* where individual computing nodes receive a part of the data and only constructs a sub-tree of the final GST (Inter-node Parallelism)

2. As these nodes are equipped with multiple cores, and they will build the individual GSTs in parallel using *shared memory architecture* (Intra-node Parallelism)

3. Use external memory to store and share the resulting GSTs to reduce the sizeable main memory requirement

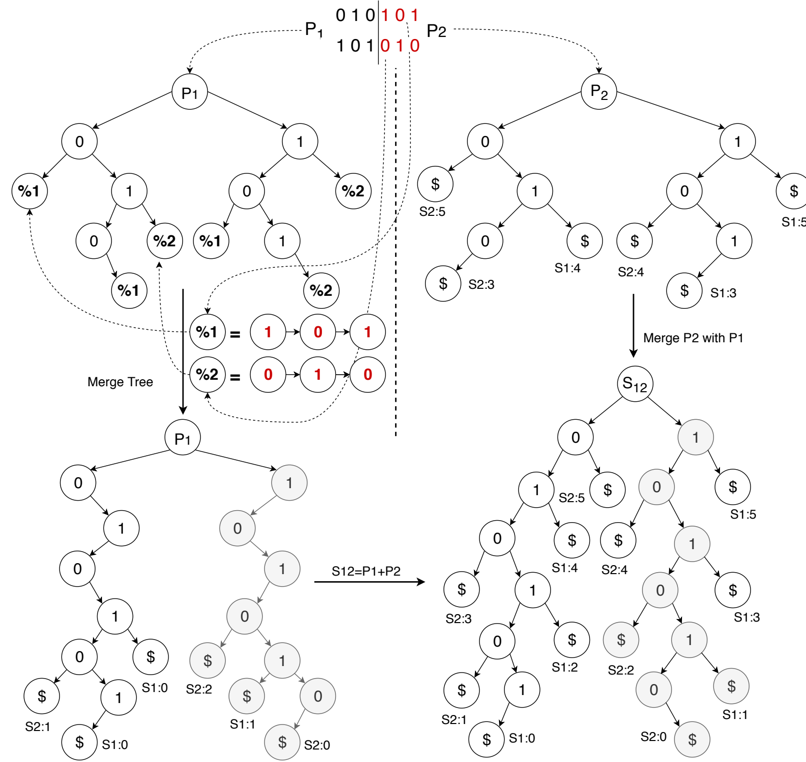


Figure 3.4: Vertical partitioning with path graphs (%1, %2) merging

3.3.1.2 Private Storage and Queries

After constructing the GST in parallel in a private cluster, the resulting GST is stored in a third party cloud server. The utility of a commercial cloud service is motivated by its low cost and higher storage requirement from GST. Furthermore, cloud service provides a scalable and cost-effective alternative to the procurement and management of required infrastructure costs, which will primarily handle queries on genomic data. As shown in Figure 3.3, the researchers only interact with the cloud server, which contains the parallel constructed GST.

However, using a third-party vendor for storing and computing sensitive data is often not permissible as there have been reports of privacy attacks and several data leaks [27]. Therefore, we intend to store the genomic data on these cloud servers with some privacy guarantee and execute corresponding string queries alongside. Specifically, our privacy-preserving mechanisms will conceal the data from the cloud

server; in case of a data breach, the outsourced genomic data cannot be traced back to the original participants. Further details on the threat model are available in § 3.3.3.4.

3.3.2 Parallel GST Construction

We propose multiple techniques to construct the GST in parallel. These approaches fundamentally differ in partitioning and agglomeration according to the PCAM (Partitioning, Communication, Agglomeration and Mapping) model [89]:

3.3.2.1 Data Partitioning:

We utilize different data partitioning scheme based on the memory locality, availability and the number of computing nodes:

Horizontal partitioning groups a number of sequences for the existing computing nodes. Each node will receive one such group and construct the corresponding GST afterwards. For example, if we have $n = 100$ sequences and $p = 4$ nodes, then we will split the data into 4 groups where each group will contain $|n_i|= 25$ records or genomic sequences. Each node, p_i will build their GST on $|n_i|$ sequences of m length in parallel without any communication. Figure 3.1 depicts a simple case of this partition scheme for $n = p = 2$.

Vertical partitioning divides the data across the columns and distributes it following the aforementioned mechanism. However, this scheme will have some additional implications while merging the resulting sub-trees (§ 3.3.2.2). For example, if we have genomic data of length $m = 100$ and $p = 4$, we will have $n \times m_i$ partitions where each dataset will have $|m_i|= 25$ columns.

Bi-directional data partitioning combines both the horizontal and vertical approach as it divides the data into both directions. Notably, it can only operate for $p \geq 4$ cases. Given $n = 100$, $m = 100$ and $p = 4$, each node will receive a $n_i \times m_i = 50 \times 50$ sized data for their computations.

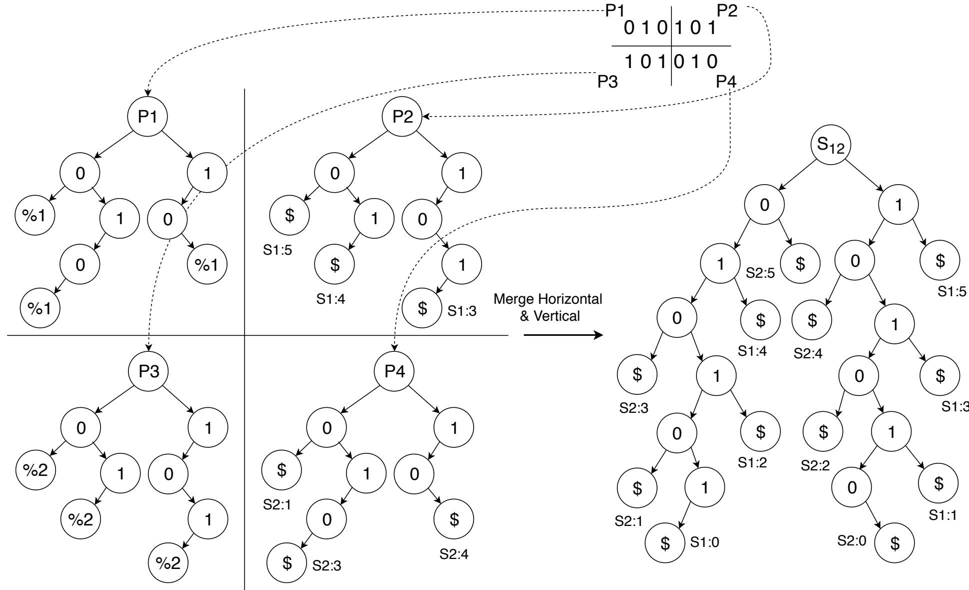


Figure 3.5: Example of Bi-Directional partitioning scheme

3.3.2.2 Distributed Memory:

We use several machines (or nodes) to build the final GST in parallel (*Inter*-node Parallelism), each with their individual global memory and connected via network. After receiving the partitioned data, these computing nodes are required to build their own GSTs. For example, if there are $p_0, \ldots, p_{|p|}$ nodes then we will have $GST_0, \ldots, GST_{|p|}$ trees. Regardless of the partitioning mechanism, we use the same linear time method to construct the suffix trees using Ukkonen's algorithm [88]. After these individual nodes build their GSTs, they need to share them for the merging operation described next.

Definition 3.3.1 (Merge Generalized Suffix Trees)**.** Given two suffix trees T_1 and T_2 from two sequences $S1$ and $S2$ with m length, the leaf nodes of the merged tree T_{12} will contain all possible suffixes of $S1 : i$ and $S2 : i$ $s.\ t.$ $i \in [1, m]$.

In Figure 3.1, we see a horizontally partitioned GST construction. Here, two suffix trees are merged where the grey and white colored nodes belonged to different trees. Notably, the merge operation did not duplicate any node at a particular depth. For example, if there was already a node with the value 0 is present, then it will not

create another node and simply merge onto its branches. This condition is applied to all merge operations to avoid duplicate branches.

However, for vertical and bi-directional partitioning, the merging requires an additional step for datasets where $m_i < m$. We illustrate this in Figure 3.4 where $n = 2, p = 2, m = 6$ and we are creating GST for the sequences `S1,S2`= $\{010101, 101010\}$. Here, $p1$ operates on $\{010, 101\}$ partitions whereas $p2$ generates the tree for $\{101, 010\}$. Here, the GST from $p1$ needs to have different end characters compared to $p2$ as each end points in a suffix tree needs to represent that the suffix has ended there. However, since we are splitting the data on columns, it needs to address the missing suffices.

Therefore, we perform a simple merge for all cases with $m_i < m$ as we add the *Path Graphs* with $m - m_i$ characters on the resulting GST. For example, in Figure 3.4, we add the path graph of 101 (represented as %1) in all end characters of `S1` in $p1$ (after 010). Similarly, we need to add 010 for `S2` represented as %2. During this merge, we also do not create any duplicate nodes. The addition of these paths will require the merge operation described next.

3.3.2.3 Shared Memory:

In the distributed memory environment, the individual machines get a partition of the genomic data to build the corresponding GSTs. However, these machines or nodes also have multiple cores available in their processor which share the fixed global memory. Therefore, we also employ these cores to build and merge the GSTs in parallel.

We utilize the fixed alphabet size property of genomic data in our shared memory model (*Intra*-node Parallelism). Since there can be only fixed number of children from the root, we can use the separate cores to process the GSTs. For example, the first core (or process) can handle the 0 leading suffixes whereas another core operates on 1's. In Figure 3.1, two processes $p1$ and $p2$ will generate the suffix tree of $\{01, 0101, 010101\}$ and $\{1, 101, 10101\}$ respectively. The output will be two suffix trees, one from each process which is joined to the root for the final tree.

It is noteworthy that the GSTs on the partitions can also be build with this shared environment. Here, we will partition the data into the cores and they will

build, merge the GST in parallel. However, the number of cores and memory is limited which will restrict the construction for large datasets.

3.3.2.4 Merging GSTs:

As mentioned earlier, the merge operation takes two different GSTs and adds all their vertices. Hence, all $|p|$ GSTs are merged into the final GST where $GST = GST_0 + \ldots + GST_{|p|}$. Here, we employed the shared memory parallelism as the children of the root (0/1) are totally separate and do not have any common edges. In other words, we can treat the root's 0 branch separately from child 1. This allows us to perform the merge operation in parallel and utilize the Intra-node parallelism in each computing nodes.

Notably, merging one branch of a tree is a serial operation as multiple threads cannot add or update branches simultaneously. This creates a bottleneck as we need to perform merge operation in all GST_i's and add the path graphs mentioned in § 3.3.2.2. However, we can use multiple cores for different branches as mentioned in § 3.3.2.3. For example, we can create two processes for handling the 0 and 1 branch from the root. This can be extended for the suffixes starting with $00, 01, 10, 11$ as well. Notably, this parallel operation can be followed for any dataset with fixed character set.

The full merge operation is depicted in Figure 3.5 where we perform the bi-directional partition and merge accordingly. Inherently, the bidirectional strategy employs both vertical and horizontal merging strategies as the end columns do not include the $m - m_i$ characters.

3.3.2.5 Communication and Mapping:

We use a sequential distribution of work where incremental computing nodes receive contiguous segments of the data. For example, with horizontal and vertical partitioning, each node p_i will receive $\lceil n/p \rfloor \times m$ and $n \times \lceil m/p \rfloor$ records, respectively.

As p_i constructs its GST_i, it stores it in the file system for further processing. Upon completion, all GSTs are sent via network to the nearest processor based on

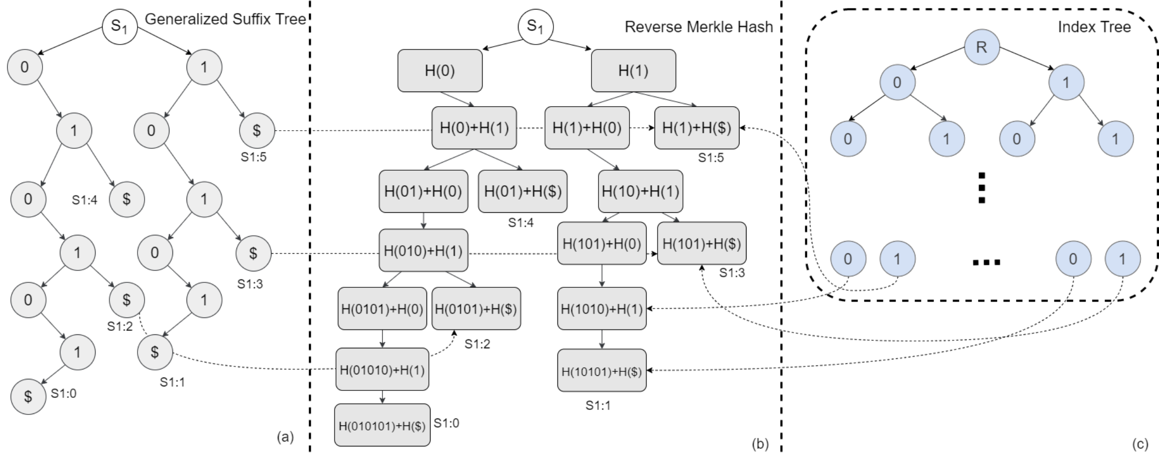


Figure 3.6: Reverse Merkle Hash for Suffix Tree on $S1 = 010101$ where we hash the value of each node in a top-down fashion

latency. For example, in Figure 3.5, P3 and P4 will send their GST to P1 and P2 respectively and P1, P2 will merge these trees in parallel. We utilize external memory as the suffix tree are arbitrarily large for a genomic dataset and can overflow the main memory of a single computing node.

3.3.3 Privacy Preserving Query Execution

In this section, we discuss the mechanisms that allow privacy preserving queries on suffix trees.

3.3.3.1 Merkle Tree

Merkle tree is a hash-based data structure which is often used as a data compression technique [90]. Here, the data are represented as leaf nodes of a binary tree and they are hashed together in a bottom-up fashion. The individual node values are determined from its children as they are concatenated and hashed with any cryptographic hash function (*i.e.*, MD5, SHA-2 etc.). For example, the parent A of leaf nodes with value 0 and 1 will denote $A = Hash(Hash(0)||Hash(1))$. Similarly, if its sibling is denoted by B and then their parent will have $C = Hash(Hash(A)||Hash(B))$ ($||$ is a concatenation).

3.3.3.2 Reverse Merkle Tree

In this work, we utilize a reverse of the Merkle Tree hash where the data is hashed in a top-down fashion. For example, a child node will have the hash value $A = Hash(P||Hash(0))$ where 0 and P is the hash value of the node and its parent, respectively. The sibling will have $B = Hash(P||Hash(1))$, analogously as shown in Figure 3.6(b). We initialize the root's hash value with a random value (SALT) for additional security which is mentioned in § 3.3.3.

Here, as the GST is constructed in parallel, we hash the content of the individual nodes alongside the SNP values. The hash values are passed down to the children nodes and added with their hashed SNP value. In Figure 3.6, we show the example of a reverse hash tree for the sequence $S1 = 010101$. Here, in each node, we take the hash of the parent node and add it to the hash of that node's value. Notably, in Figure 3.6, we write $H(AB)$ to replace $H(H(A)||H(B))$ in short. The leaf nodes will also have the position of the suffix appended together with the nucleotide value (represented as $ in Figure 3.6(b)).

The rational behind using the reverse Merkle tree is to represent the suffixes using the hash values for faster matching. Here, the hash values on the leaf nodes represent the corresponding suffixes of that edge in the GST. For example, the longest path in Figure 3.6 will represent $S1 : 0$ and contains the hash for suffix 010101. We also keep the position of the suffix alongside the hash values. These leaf hash values are kept separately for incoming queries which accelerate the search process as we describe it in § 3.3.3.

3.3.3.3 Cryptographic Hash Function

The cryptographic function employed to hash the values in each node is quite important. As there are multiple hash functions available (*i.e.*, MD5, SHA-1, etc.), they ultimately serve a similar purpose. These functions provide a deterministic, one-way method to retrieve a fixed bit size representation of the data. Therefore, it can also be considered as a compression technique that reduces the genomic data size in this problem.

We utilized MD5 as an *example* function throughout the work as it was executed on every node as described in § 3.3.3.2. Here, it is important to consider the size of the hashed values as MD5 provides a fixed 128-bits output. Using another hash function with better collision avoidance or more security (*i.e.*, SHA-1) may result in longer (256 bits) hash values, which will increase the execution time linearly in order of the bit size. Nevertheless, MD5 is given as an example that can be replaced with any cryptographic hash function.

3.3.3.4 Privacy Model

The first goal here is to ensure that the privacy of the data (or GST) in an untrusted cloud environment. Therefore, we expect the cloud to learn nothing about the genomic sequences beyond the results or patterns that are revealed from the GST traversal. Note that the proposed method do not guarantee the privacy derived from the query results as it might be possible for the researchers to infer private information of an individual using the query results. The proposed secure techniques do not defend the genomic data against such privacy attacks, where researchers may act maliciously. Nevertheless, we discuss some preventive measures in § 3.4.4.

However, the privacy model is different for the cloud service provider (CS) as we adopt the semi-honest adversary model [66]. We assume that CS will follow the implicit protocols but may attempt to retrieve additional information about the data from the underlying computations (*i.e.*, logs). This is a common security definition, and it is realistic in a commercial cloud setting since any cloud service providers comply with the user agreement and cannot use/publish the stored data without lawful intervention.

In addition, the system has the following properties: a) the CS does not collude with any third party or researchers to learn further information, b) in case of an unwanted data breach in the cloud service, the stored GST (or genomic data) does not reveal the original genomic sequences, and c) Researchers are assumed honest as they do not collude with other parties to breach the data.

3.3.3.5 Privacy-Preserving Outsourcing

As the GST is constructed in parallel in a private cluster, the resulting suffix tree is stored (or outsourced) in a commercial cloud server (CS). The researchers will present their queries to this CS, and CS will search on the GST for the corresponding queries. For example, if we consider the four queries from § 3.2.3, each will warrant a different level of search throughout the GST.

Since we intend to ensure the privacy of the genomic data in an untrusted environment, we remove the plaintext nucleotide values from the GST. While storing the GST in CS (from the private cluster), we only consider the hashed values after following the Reverse Merkle Tree method, as mentioned in § 3.3.3.2. For example, the hash tree in Figure 3.6 (b) is only stored on the CS which points to the leaf nodes (sequence number and suffix position) of the GST and do not contain any data.

Since a genomic dataset will only have limited input characters (A, T, G, C), hashing them individually will always produce the same output. As a result, CS (or any third party) can infer the hashed genomic sequences. Therefore, to protect the privacy of the data, we utilize two methods: a) A random byte array is added to the root of the GST, kept hidden from the CS, and b) the final hash values are encrypted with Advanced Encryption Standard (AES) in the block cipher mode (AES-CBC) prior to storage.

As the one-way public hash function reveals the genomic sequence with a limited alphabet size, we need to randomize the hash values so that CS cannot infer additional information. This inference is avoided with a standard random byte array, namely `SALT`. Here, the root of the GST (Figure 3.6 (a)) contains this SALT byte array which is no revealed to CS. As this SALT array of the root node is appended to its children nodes, it will cascadingly alter all the hash values downstream.

For example, while generating Figure 3.6 (b) from (a), the left and right child of root $S1$ will contain the value $H(SALT||H(0))$ and $H(SALT||H(1))$, respectively. The random SALT byte has the same length as of the hash function output (128 random bits for MD5). Since CS does not know these SALT bits, it will be challenging to guess an arbitrarily long genomic dataset since the hashing is also done repeatedly.

Algorithm 1: Encrypted Reverse Merkle Tree (RMT)

```
  Input: rootNode, SALT bytes, secret key
  Output: encrypted nodes using AES-CBC and reverse merkle hashing
2 Procedure ReverseMerkleTree(node, previousValue)
3   | node.val ← Hash(randomBytes||node.val)
4   | foreach child of node do
5   |   | ReverseMerkleTree (child, node.val)
6   | end
7   | encryptedNode← AES − CBC(node, key)
8   | return encryptedNode
9 ReverseMerkleTree (root, SALT)
```

The SALT bytes are also sent to the researcher as it is required to construct the query as well.

These individual hash values are also encrypted with AES-CBC with 128 bit keys. This AES mode requires an random Initialization Vector (IV) which is also shared with the researcher but kept hidden from CS. This encryption is done to provide an additional layer of security if the server ever gets compromised as it should prevent any data leakage. The procedure to get the Encrypted Reverse Merkle tree is described in Algorithm 1.

Therefore, according to our privacy model in § 3.3.3.4, the Reverse Merkle Tree containing the encrypted hash values of the original dataset is safe to transfer over to a semi-honest party. As we also assume the CS to be honest-but-curious, it will follow the assigned protocols and will not attempt any brute force attacks on the hashed values. However, under any data breach, the proposed encrypted tree will suffer the same limitations of symmetric encryption.Notably, some of them can be avoided by using asymmetric encryption or separate secret keys for different heights or depth of the GST which will strengthen the security; we discuss this in § 3.4.4.

3.3.3.6 Privacy-Preserving Query Execution

The four queries mentioned in § 3.2.3 will be executed using the AES-CBC encryption and Reverse Merkle Tree-based hash values (as outlined in § 3.3.3.2). These hash values compress the nucleotides available on each edge to a fixed number of bits (size

Algorithm 2: Encrypted query using RMT ($\mathcal{E}_H$)

Input: Query String q, SALT bytes, secret key
Output: Encrypted Query String, $\mathcal{E}_H$
2 $hashVal \leftarrow SALT$ **foreach** *character of query* q **do**
3 | $hashVal \leftarrow Hash(hashVal||Hash(character))$
4 **end**
5 **return** *AES-CBC(result,key)*

of the hash) and offer an advantage when searching over the whole GST.

Hash Index ($\mathcal{HI}$): We created another intermediate index on these encrypted hash values using a traditional binary tree (B-tree). Since our hash function will always provide a fixed sized output (in bits) for each node, we can construct a B-Tree based on the symmetrically encrypted bits. For example, MD5 will always output the same 128-bit hash for the same SALT and series of nucleotides along the Reverse Merkle tree. Encrypting these fixed size values with AES-CBC with the same key will produce ciphertexts which can later be utilized for searching as the researchers can come up with the same ciphertexts given a matching query.

The output from the AES-CBC bits are kept in a binary tree having a fixed depth of 128 (from root to leaf) as we use 128 bit encryption. Here, the leaf nodes will point towards the hash value or the nodes appearing on the Reverse Merkle Tree. We name this B-tree as $\mathcal{HI}$ as it replaces an exhaustive search operation on GST. Notably, we incorporate the positions of the suffixes from GST into the $\mathcal{HI}$ using the end character S symbol which was appended with the genomic sequences. This positional value (*i.e.*, $S0$) contained the starting index of the suffix which was necessary for all queries with a targeted position.

We can demonstrate the efficacy of $\mathcal{HI}$ for the Exact Match (EM) query as defined in Definition 3.2.3. Here, the researcher initiates the query as s/he can have one or multiple genomic sequences to search for in the dataset $\mathcal{D}$. The researcher constructs the hash representation of the query using the secret key and random byte array (SALT) that was employed to make the GST stored in the CS. For example, if the query is `010101`, then the query hash will be: $\mathcal{Q}_H = H\left(H(\ldots + H((H(0) + H(SALT)) + H(1))\right.$. Later, it will be encrypted with the key

$\mathcal{E}_H = \mathcal{E}(\mathcal{Q}_H, key, IV)$ and sent to CS for matching. The procedure to retrieve $\mathcal{E}_H$ is briefed in Algorithm 2.

CS will search for this $\mathcal{E}_H$ in the fixed size ($\mathcal{HI}$) first. If the hash exists on the B-tree, CS returns the leaf node that $\mathcal{HI}$ is referencing. Here, only the leaf nodes of $\mathcal{HI}$ keep a reference of the Reverse Merkle Tree nodes which is sent as the final result to the researcher (in case of a match). For a mismatch, we will not have a node on $\mathcal{HI}$ for the query hash, resultingly, do not need to check GST anymore.

Lemma 3.3.1 (Runtime for Exact Match). For a hash function H with fixed output size $|H|$, Exact Match (definition 3.2.3) will require a runtime in order of $\mathcal{O}(\log|H|)$ for any arbitrary query.

In Lemma 3.3.1 we consider the output size of the hash function for simplicity as AES will produce the same number of bits as inputs. Nevertheless, we can extend the method for EM (Lemma 3.3.1) to execute the rest of the queries. For example, a substring Match can be an extension of EM where we will consider a query length, $|q|$ smaller than the sequence length ($\leq m$) and should match in the exact positions of the dataset sequences. This is also possible employing $\mathcal{HI}$ which represents the strings residing in a GST.

Similarly, for the Set Maximal Matching (SMM-definition 3.2.5), the researcher and CS perform an iterative protocol. The researcher initially searches for the whole query following Lemma 3.3.1 on the $\mathcal{HI}$ leading to the GST residing in CS with the specific position. For a mismatch, it reduces the query length by one and iterates the search until there is a match. The worst-case running time for such operation will be in order of $\mathcal{O}(|q|\log|H|)$. PVSMM (definition 3.2.6) is an extension of the same protocol where we have a threshold constraint which further reduces the computations ($\mathcal{O}(t \log|H|)$ given $t > |q|$).

3.4 Results

Before discussing the findings, we will describe the implementation and underlying dataset details:

Table 3.2: Horizontal and Vertical partition scheme execution time (in minutes) to build GSTs with number of processors $p = \{1, 2, 4, 8, 16\}$

Horizontal Partitioning												
Data	Serial	Distributed				Shared				Hybrid		
	1	2	4	8	16	2	4	8	16	2	4	8	16
200	0.08	0.23	0.09	0.09	0.10	0.14	0.05	0.04	0.03	0.14	0.07	0.05	0.05
300	0.27	1.04	0.23	0.2	0.23	0.38	0.15	0.11	0.08	0.37	0.16	0.12	0.12
400	0.59	2.03	0.55	0.38	0.38	1.18	0.35	0.21	0.2	1.12	0.31	0.23	0.25
500	1.53	3.14	1.32	1.06	1.01	2.27	0.57	0.36	0.28	2.09	0.52	0.38	0.41
1000	**14.55**	16.23	8.34	6.31	**6.09**	17.38	5.56	3.27	**2.28**	17.14	4.18	3.12	**3.08**
Vertical Partitioning													
	1	2	4	8	16	2	4	8	16	2	4	8	16
200	0.08	0.19	0.08	0.05	0.03	0.16	0.07	0.04	0.02	0.14	0.05	0.03	0.02
300	0.27	0.56	0.28	0.17	0.09	0.48	0.22	0.16	0.08	0.39	0.13	0.10	0.06
400	0.59	1.41	1.05	0.36	0.16	1.44	1.01	0.34	0.19	1.21	0.32	0.21	0.13
500	1.53	3.07	1.49	1.08	0.37	3.18	1.49	1.08	0.36	2.35	0.58	0.40	0.24
1000	14.55	**25.24**	12.25	9.06	5.20	22.56	13.11	7.2	4.37	18.22	6.31	4.49	3.10

Table 3.3: Execution time (in seconds) of bi-directional partitioning to build GST on different datasets with number of processors $p = \{1, 4, 8, 16\}$

Data	Serial	Distributed			Shared			Hybrid			
	1	4	8	16	4	8	16	4	8	16	32
200	4.8	94.2	90	87	43.8	42.6	38.4	70.8	73.2	75.6	1.51
300	16.2	121.8	107.4	106.2	72	48.6	43.2	88.8	75	75.6	1.37
400	35.4	168.6	148.2	124.8	102.6	54	57.6	114	87	96	1.36
500	91.8	231.6	151.8	154.8	145.2	76.2	62.4	146.4	103.8	105	1.36
1000	873	1135.2	428.4	291.6	856.8	202.2	154.8	635.4	312	214.2	1.36

3.4.1 Evaluation Datasets and Implementation

We evaluate our framework on uniformly distributed synthetic datasets as it allows us to perturb the dimensions and check the performance of the underlying methods. Hence, we generate different datasets with $n, m \in$ {200, 300, 400, 500, 1000} and name them accordingly. We agree that genomic data of n, m in millions will portray the true benefit of our proposed parallel constructions, but due to our computational restrictions, we limited our experiments as we were only able to access a small computing cluster [91]. However, we argue that larger datasets will denote the same trend in terms of execution time as we increase the parallel computational power.

3.4.2 Performance Analysis

We analyze our proposed approach in terms of n, m, p and all three (distributed, shared and hybrid) memory models. Here, the distributed memory model will not incorporate any intra-node parallelism instructions as discussed in § 3.3.2.3 whereas the hybrid method will utilize both.

The shared memory architecture distributes the work into different co-located processors (cores) on one single node. Notably, in this model, we do not require any communications between two processes whereas the distributed model will incur communicating the GSTs. However, the number of processors and memory available in shared model is fixed and limited as we can add new machines in the distributed model. Nevertheless, this comparison will denote the difference in the two memory architecture.

In Tables 3.2 and 3.3, we show the execution time of horizontal, vertical and bi-directional partitioning, respectively. Each method is executed on $p = \{2, 4, 8, 16\}$ processors, whereas $p = 1$ denotes the serial or sequential execution. The sequential method is plaintext Ukkonen's algorithm [88]. Furthermore, the proposed hybrid approach uses both distributed and shared memory model with two cores on each processor of distributed machines for the 0 and 1 branches of GSTs. All experiments are conducted on the mercury cluster at UofManitoba Computer Science (`www.cs.umanitoba.ca/computing`).

In Table 3.2, The GST building time for smaller datasets ($n, m \leq 200$) are almost same for all settings. However, as the dataset size increases, the difference in execution time starts to diverge. For example, the sequential execution of D_{200} takes 0.08 minutes whereas D_{1000} requires 14.55 mins. The same operation takes 3.08 mins on the hybrid approach with $p = 16$. Similarly, the distributed model takes 6.09 mins, which shows the impact of intra-node parallelism.

However, one interesting outcome is the shared model's performance. It takes the minimum time of 2.28 mins with $p = 16$ which is the lowest in all three experimental settings. However, it is noteworthy that it ran out of memory for datasets $n, m >$ 1000. This depicts the necessity of the distributed or hybrid model as a shared

Table 3.4: Maximum Execution time (seconds) of Tree Building (TB), Add Path (AP) and Tree Merge (TM) for D_{1000}

	Horizontal			Vertical			Bi-directional		
p	TB	AP	TM	TB	AP	TM	TB	AP	TM
4	113.35	-	70.02	292.97	2.7	66.8	4.01	0.37	3.85
8	47.38	-	85.4	138.87	2.9	61.1	0.62	0.16	1.8
16	15.6	-	98	64.4	3.2	57.6	0.12	0.07	1.2

memory model is more suitable for datasets which only fits the main memory.

Table 3.2 also shows the impact of vertical partitioning where we need to add the path graphs. This addition is the only difference from the horizontal approach as all the nodes working on data $m_i < m$, needs to merge $m - m_i$ characters to the underlying GSTs. For example, with vertical method it takes 25.24 mins to process D_{1000} whereas it took only 16.23 on horizontal approach. The rest of the execution time also follows the same trend as more processor leads to faster executions overall. The performance gain with shared model compared to hybrid is also lost due to the thread synchronization as the threads operate on $m_i < m$ requires more time for sequential path graph addition.

In Table 3.3, we show our best results where the data is partitioned into both directions. Here, the tree building cost is reduced compared to the prior two approaches as it resulted in smaller sub-trees. For example, with $n = m = 100$ and $p = 4$, each processor p_i will work on 25×25 sized matrix whereas it will lead to 25×100 and 100×25 partitions for horizontal and vertical, respectively.

Table 3.4 demonstrates the granular execution time for tree building, path graph addition and the merge operation. We took the maximum time from each run as these functions were executed in parallel. Notably, these values are the building blocks for Table 3.2 and 3.3. For example, the tree building time decreases with the increment of processors p. Furthermore, the bi-directional tree build cost decrements with the increment in processors as it divides the data by half.

In Table 3.5 we summarize the speedup ($= T_{par}/T_{seq}$) results for D_{1000}. Here, the shared model performs well compared to distributed model due to its zero communication cost. Notably, the distributed one is competitive for all $p > 2$ cases.

Table 3.5: Speedup analysis on D_{1000} for all methods with $p = \{2, 4, 8, 16\}$

Method	Distributed			Shared			Hybrid		
	4	8	16	4	8	16	4	8	16
Horizontal	1.19	1.61	2.80	1.11	2.02	3.33	2.31	3.24	4.69
Vertical	1.74	2.31	2.39	2.62	4.45	6.38	3.48	4.66	4.72
Bi-directional	0.77	2.04	2.99	1.02	4.32	5.64	1.37	2.80	**4.08**

Nevertheless, the shared model could not finish the $D_{10,000}$ as it ran out of the shared memory. On the contrary, both distributed and our hybrid model constructed the targeted GST as it did not depend on the limited, fixed main memory of one machine.

One of the limitations of the proposed framework is the size of the resulting suffix tree. Since the node contents are hashed and encrypted, it also increases the memory requirements as we utilized file-based memory to handle queries. For example, for a dataset of $\{500 \times 500, 1000 \times 1000, 2000 \times 2000, 5000 \times 5000\}$ takes around $\{109, 433, 1754, 11356\}$ megabytes of storage space. Notably, this storage accounts for the hashed tree and the AES-CBC encrypted node values on the Merkle tree. Furthermore, we opted to experiment with a relational database (MySQL) to save the encrypted tree, which is detailed in our code repository [92].

3.4.3 Query Execution

3.4.3.1 Experimental Setup

In this section, we analyze and discuss the execution time of the four queries in private and non-private settings as defined in § 3.2.3. We utilized an Amazon EC2 cloud server (specification `g4dn.xlarge`) as the CS while the researcher was in Winnipeg, Canada. The average network latency between the CS and the researcher was around 49ms. The key components of the result analysis are as following:

1. Execution time for all queries with worst-case inputs,

2. Effect of dataset size and query length

3. The impact of GST and $\mathcal{HI}$, and

4. The runtime comparison between hashing and GC

We targeted the worst-case input queries as it will highlight the maximum execution time for each type of query. For example, for exact matches (EM), we randomly picked a query sequence from the dataset. As any mismatch on the $\mathcal{HI}$ will forcefully reduce the computations, we chose to pick available sequences for Query 3.2.3. For SMM and TSMM Queries (3.2.5 and 3.2.6), we preferred a random query sequence which was not present in the dataset. As for a mismatch, SMM (and TSMM) will redo the search altering the query sequence. This will show the maximum execution time required. Alternatively, if we picked a sequence from the dataset (similar to EM), it was not necessary to traverse the $\mathcal{HI}$ and it will output the same execution time as EM.

Therefore, our targeted four queries can be reduced to EM. For example, we do not discuss the exact *substring* matches in this section as it took the same time as the EM. We also limit the execution time for two datasets D_{1000} and D_{500} as the data size will increase the size of the GST but not $\mathcal{HI}$. Therefore, we examine the scalability issues with different query lengths $|q| \in \{300, 400, 500\}$ and $(n, m) \in \{(1000, 1000), (500, 500)\}$.

3.4.3.2 Execution Time for GST (w/o privacy)

Initially, we analyze the execution time of the targeted queries on plaintexts without any privacy guarantee in Table 3.6. Here, we only execute the queries on the generalized suffix tree (GST) as they are outsourced on CS and simulate the researcher on the same server to avoid the random network latency. The execution time from the Table 3.6 clearly shows that longer query sequences (*i.e.*, $|q| = 500$) require more time than smaller queries. As we are searching on the suffix tree, our GST indexing presents a runtime linear to the order of the query length of $|q|$. Notably, GST allowed us to remove the runtime dependency with the number of sequences or nucleotide (n or m) which is often higher for genomic datasets.

One interesting observation here is the scalability property of GST on different sized datasets. As we considered two different datasets D_{1000} and D_{500} with

Table 3.6: Exact Matching, SMM and TSMM (Query 3.2.3, 3.2.5 and 3.2.6) using GST considering different datasets and query lengths (time in milliseconds)

Query Length $\lvert q\rvert$	D_{1000}			D_{500}		
	EM	SMM	TSMM	EM	SMM	TSMM
300	0.5	140	94	0.3	80	70
400	0.5	140	150	.4	130	120
500	0.6	210	220	0.5	190	180
1000	1.1	680	720	-	-	-

$n, m = \{1000, 500\}$, it seems that the runtime does not increase significantly. Ideally, traversing the GSTs from D_{1000} or D_{500} for a query should not be different but the increased number of nodes on memory adversely affects the query execution.

3.4.3.3 Execution Time for $\mathcal{HI}$ (with privacy)

Since the query length $|q|$ can also be arbitrarily large, we reduce its impact on execution time by employing $\mathcal{HI}$. This index $\mathcal{HI}$, built on the GST allows us only to search up to the hash output length $|H|$ rather than $|q|$. We see its effect in Table 3.7, as for different $|q|$'s, the execution time for EM did not increase which was the opposite for plaintext GST as shown in Table 3.6.

Since we considered the worst-case inputs (non-matching query sequences) for SMM and PVMM, both types of queries required more matching requests on the cloud server. These iterative query executions increased the runtime incrementally. The effect of the dataset size is also analogous with our earlier argument as the time vary slightly for different sized datasets. We do not show the results for SMM over the garbled circuit as they required over an hour each on the worst-case inputs.

We also benchmark with recent work from Shimizu *et al.* [31] which utilized positional Burrows-Wheeler Transformation with Oblivious Transfer (OT-secure protocol) for input privacy. From the results in Table 3.6, it is evident that our Merkle hash along with $\mathcal{HI}$ provides a $4\times$ speedup compared to the earlier work as it takes 160.85 seconds to execute a set maximal match on D_{1000} (our method required 36.76s). However, since this benchmarking method only used OT rather than more expensive GC operations, it was faster than the GC protocol.

Table 3.7: Secure Exact Matching (EM), SMM and TSMM (Query 3.2.3, 3.2.5 and 3.2.6) using $\mathcal{HI}$ considering different datasets and query lengths (time in milliseconds). QP, GC, $|q|$ denotes query processing time, Garbled Circuit, and Query Length respectively.

$\lvert q\rvert$		Reverse Merkle Hash with $\mathcal{HI}$						GC		Shimizu *et al.* [31]	
		D_{1000}			D_{500}			D_{1000}	D_{500}	D_{1000}	D_{500}
	QP	EM	SMM	PVSMM	EM	SMM	PVSMM	EM	EM	SMM	SMM
300	0.79	41.4	11599	2862	37.1	11100	2761	63246	63583	50358	43163
400	0.84	43.9	15337	3901	36.9	15385	3760	63194	62639	64867	55609
500	0.9	42.7	18563	4836	37.2	18477	4875	63439	62048	70754	67965
1000	1.58	45.2	36761	9368	-	-	-	63391	-	160854	

3.4.4 Limitations and Discussions

Parallel construction of GST: GST provides an efficient index to the genomic data which can be utilized in different search queries, fundamental to the utility of the data. However, the best sequential algorithm is linear to the sequence length which can prove to be significant for a large dataset with longer genomic sequences n, m. Therefore, constructing such an expensive tree-based data structure is handled by the proposed parallel mechanism, which is required to be executed only once while pre-processing any dataset.

Storage complexity of GST: On the other hand, we use a file-based GST due to the limited main memory in comparison to the cheaper disk-memory. This also warrants the usability of cloud servers, which offers less-expensive storage solutions. Here, GST levies an expensive storage cost as the number of suffixes increases linearly in order of the length of the sequence (m). For example, a genomic sequence of length m has $m + 1$ suffixes which increases for larger n, m. Resultingly, we incorporate another fixed-size index $\mathcal{HI}$ on GST, which acts as the principal component while searching and can fit into the main memory.

Privacy guarantee from encrypted hash value: The privacy of the data relies on the symmetric AES cryptosystem along with the random SALT bytes kept on the root node of Reverse Merkle Hashing. We did not use any asymmetric public-key encryption scheme due to the resulting ciphertext size expansion. Nevertheless, the recently improved homomorphic encryption schemes might be beneficial in this task

and provide additional security guarantee [38, 93]. This is an interesting future work. *Output privacy:* To protect the data against researchers with malicious intent, we can perturb the outputs from CS with some privacy guarantee. This can be done by adding noise to the query results, and these techniques have been studied in the literature (*i.e.*, anonymization [94], differential privacy [27]). However, we did not opt for these strategies as they will thwart the exact results from any query and validity is quintessential in any scientific research. The popular model in genomic research also assumes the researchers to be honest as they adhere and understands the privacy requirements of genomic data.

3.5 Related Works

Since we target two different research area concerning genomic data, we discuss the related works separately below:

3.5.1 Privacy-preserving String Search

There are different types of string search functions targeted on genomic data using different cryptographic protocols. Table 3.8 summarizes some of the the existing approaches for privacy-preserving searches where we show the high-level difference among them.

Apart from these cryptographic computation approaches, there has been several works on publishing genomic data with theoretical privacy guarantee. Chen *et al.* [95] proposed a differentially private [30] mechanism to publish n-grams with variable-lengths. These n-grams can be used for count queries or mining string patterns. However, due to the Laplace noise utilized, the outputs will have inaccuracies which we do not have in this work.

Secure exact string search has been a popular since the realization of the privacy aspects of human genomic data. In 2003, Atallah *et al.* [96] proposed a dynamic programming algorithm to compare two sequences using homomorphic encryption. Other contemporary approaches utilized finite automata with Recursive Oblivious

Transfer (ROT) to solve the DNA searching problem [97]. Troncoso *et al.* [98] proposed a similar approach for approximate string matching requiring (amortized) linear time. Sudo *et al.* [99] presented a unique algorithm employing secure wavelet matrix and additive homomorphic encryption to search for substrings in logarithmic time. Using Garbled Circuits (GC) and tree-based index, Mahdi *et al.* [100, 101] proposed count queries on genomic data which relied on exact string matching. In this work, we avoid the expensive GC protocols with Reverse Merkle Hashing.
Secure exact substring search targeting a specific position is a related but a different problem where the SNP position plays an important part. In this chapter, we proposed a secure model to find a match between query and dataset sequences, where a starting position can be specified. In 2016, Ishimaki *et al.* [102] addressed this problem with a look-up table constructed from positional Burrows-Wheeler transform (PBWT) and utilized Fully Homomorphic Encryption (computation under encryption).
Set-maximal match is another important problem that is employed in genealogy queries popular in ancestry searching. Shimizu *et al.* [31] introduced a secure variable length prefix or suffix matching on SNP genomic sequences. The authors utilized Positional Burrows Wheeler Transform (PBWT) as proposed by Durbin *et al.* [103] for preprocessing the raw genomic data (alternatively, we employed the GSTs). To ensure the privacy, we utilized the reverse Merkle hash with random SALT, whereas the authors used the Recursive Oblivious Transfer (ROT) with additive homomorphic encryption. Yamada *et al.* [104] utilized a fully (and somewhat) homomorphic encryption schemes to formulate a string search framework. However, the query lengths were only selected from $[5, 25]$ taking a maximum of 80 seconds. In comparison, our method only takes around 40 seconds for a query size of 1000.

In iDASH 2019 competition [105], there was a separate track for secure set-maximal matching due to its popularity on ancestry applications. The winning solution was proposed by Sotiraki *et al.* [106] using Goldreich-Micali-Wigderson (GMW) protocol [107]. Here, the problem required two parties (data owner and researcher) and needed a private evaluation of XOR, AND and NOT operation between them. In comparison, here, we propose a secure mechanism to outsource

Table 3.8: Related works in different privacy-preserving genomic string search

Work	Exact	Substring	Set-Maximal	Query Pos
Atallah *et al.* [96]	✓	✓	✗	✗
Sudo *et al.* [99]	✓	✓	✗	✗
Ishimaki *et al.* [102]	✓	✓	✗	✓
Shimizu *et al.* [31]	✗	✗	✓	✓
Sotiraki *et al.* [106]	✗	✗	✓	✓
Our Work	✓	✓	✓	✓

genomic data to public cloud and execute similar queries. Our secure query method is also faster as it takes around 12 seconds for a query size 300 and database size of 1000×1000 where Sotiraki *et al.* takes around 60 seconds.

In 2021, Mahdi *et al.* proposed another privacy preserving query framework using Suffix trees [108]. Their experimental results show that a dataset of 2184 records, each containing 10000 SNPs, requires approximately 2 seconds for a set maximal match. They employed the garbled circuit technique but did not hash the tree contents. Therefore, the matching heavily relied on GC protocol whereas our proposed framework is secure due to the encrypted data.

3.5.2 Parallel GST Construction

Suffix tree construction is a fairly mature and well-studied problem as there have been multiple works which are shown in Table 3.9. Since Generalized Suffix Tree of a large genomic dataset does not fit a sizeable memory, there have been attempts to construct the tree in a file system [84]. These disk-based suffix trees usually store the individual subtrees on file similar to our approach [109]. For example, Tian *et al.* [110] showed a different suffix tree merging method *ST-Merge* using the Top-Down Disk (TDD) Algorithm.

Wavefront [112] and its successor ER_A (Elastic Range) [113] both targeted disk-based and parallel approach to construct suffix trees. However, these works only considered a suffix tree and distributed memory model, whereas, in this work, we propose a hybrid method and GST. Comin and Farreras [114] proposed Parallel

Table 3.9: Design-level comparison of previous and our method in parallel GST construction

Work	Parallelism model		Disk-based	GST
	Distributed	Shared		
TDD [110]	✗	✗	✓	✗
TRELLIS [111]	✗	✗	✓	✗
Wavefront [112]	✓	✗	✓	✗
ER_A [113]	✓	✗	✓	✗
PCF [114]	✓	✗	✗	✗
DGST [115]	✓	✗	✓	✓
Our Work	✓	✓	✓	✓

Continuous Flow (PCF) which efficiently distributes the lexical sorting process into multiple processors. Analogous to this work, Shun and Belloch [116] also proposed a parallel construction scheme utilizing *cilk* (shared memory) in 2014. Flick and Alura [117] proposed another distributed in-memory suffix tree construction solving the All Nearest Smaller Values (ANSV) problem. However, both works targeted suffix trees whereas GST can contain a large number of sequences which is more complicated and at the same time more useful.

There have been some Hadoop MapReduce based solutions to create suffix trees [118]. Li *et al.* [119] proposed a spark implementation of ER_A to construct GST where it was parallel to the number of sequences. In a recent work in 2019, DGST [115] offered a 3× speed up with such data-parallel platform which is better than the state-of-the-art method ER_A [113]. Nevertheless, it did not employ the shared or hybrid model as we performed better with 4× speedup. Notably, Mišić *et al.* report speedup up to 6× utilizing parallelism from Graphics Processing Units (GPUs) [120]. However, we do not use any specialized hardware and could not benchmark as their implementations are unavailable.

In this work, we target an out-of-core GST construction using distributed processors along with their multiple cores in parallel. We utilize a simple data partitioning scheme to demonstrate that our methods are comparable to these proposed methods in terms of speed-ups. Nevertheless, this parallel construction is not the primary contribution as it served as a gateway for the privacy-preserving

queries on genomic data.

3.6 Conclusion

In this work, we constructed GSTs for genomic data in parallel using external memory. We also analyzed its performance using different datasets and sample string queries. Furthermore, we proposed a novel hash-based method to outsource and execute privacy-preserving queries on the GST structure. The proposed parallel constructions and privacy-preserving queries can also be generalized for other data structures (*e.g.*, prefix trees [3], PBWT [31]) and thus can be useful for different genomic data computations.

Availability of materials

The evaluation source code can be found at
`https://github.com/mominbuet/ParallelGST`

Chapter 4

Large-Scale Genome Data Storage and Query with Privacy-Preserving Techniques

4.1 Introduction

Seminal breakthroughs in genome sequencing have provided us with an abundance of genomic data. The next generation sequencing techniques made this growth somewhat exponential as we are starting to observing datasets in volume of Petabytes [121]. This increasing availability of genome data of different individuals gives us an opportunity to zoom into the micro level and analyze the complex correlation or causation. However, this is increasingly challenging due to the size of the data, novel methods, computational complexity, and inherent privacy issues.

As mentioned earlier, the immense size of genome data comes at a price of higher storage space. An economical solution will be leveraging the cost-efficient commercial cloud computing services (*i.e.*, Amazon EC2, Microsoft Azure, or Google Cloud Platform, etc.) to host data and conduct required analysis on demand. For example, Amazon S3 and Azure Storage Services charge only $0.0208 to store 50 terabytes on a monthly base [122, 123]. More importantly, these cloud services also reduce the operational costs of running large scale experiments on such large-scale

data.

Surely the commercial cloud services can provide a cost-effective and efficient solution to the ongoing genome data storage and computation issues. However, the privacy of these records is another notable aspect as public (or unrestricted) access of genome data might lead to re-identification attacks [124], surname recovery [125], facial and voice traits reconstruction [126, 127]. Thus, genome data are highly sensitive because they are irrevocable and have stigmatizing consequences to both the individuals and their family, particularly first-degree relatives [128]. There are some surveys that demonstrate and discuss these privacy and security issues [26, 27].

Due to these concerns and reported vulnerability of the public cloud [129], data custodians are not comfortable in depositing sensitive genome data in a third-party untrusted environment without enforcing necessary protection [130]. An ideal approach is to develop a secure genome database, *i.e.*, encrypting the data and providing a security layer on top of the operations interface for safeguarding the data analysis process. Assuming the cloud service provider is semi-honest (honest but curious [131]), and we only want to protect the data from external malicious users, data custodians can run queries on the encrypted data without establishing a complete, trusted relationship.

However, this computation on encrypted data induces a cost on performance as these security primitives are not efficient as their plaintext counterparts. Scalability is another challenge as large memory consumption imposed by these security protocols might hinder the practicability of a realistic system. Therefore, in this chapter, we look into the balance between privacy and efficiency of the computation of genome data. We consider the count query operation which is the building block for various statistical analysis on genome data. A count query procedure to obtain the number of individuals satisfying a SQL-like query can be represented as:

```
SELECT count(*) FROM Sequences
    WHERE SNP1='A' AND SNP2='T' AND ...
    AND Disease = Yes
```

Single Nucleotide Polymorphism (SNP) refers to a variation of a single position on a DNA sequence (of a certain individual) such that more than 1% of the population

does not carry the same value. Although not all SNPs correspond to disorders, some of them are known to be associated with some diseases. A count query between SNPs and a specific condition is the first step to explore the correlations and serves as the building block for Genome-Wide Association Studies (GWAS).

Contributions

In this work, we propose a framework that provides better scalability and handles security issues of large-scale computations on genome data outsourced (transferred/stored) to a third party, public cloud server. Furthermore, we utilized a homomorphic cryptographic combined with garbled circuit scheme to ensure the security and tree structure to represent the arbitrary genome data for computational efficiency. The major contributions of this work can be summarized below:

- We propose a method utilizing graph-based database to store and allow computations on real-world genome data *securely*.
- A novel indexing scheme is proposed on such database to make the secure query operations more efficient.
- We test the proposed approach along with the corresponding indexing scheme on a large-scale genome dataset containing $736,317$ human SNPs ($\sim 200GB$ data).
- Experimental results show that it takes less than a minute for a query compared to best-known attempts where it required around 7 minutes [132, 133].

The rest of the chapter are organized as follows. Necessary backgrounds are discussed in Section 4.2. We discuss the proposed methods in Section 4.3 and show the results in 4.4. In Section 4.6 we discuss some of the related work. Finally, we conclude and discuss some future works in Section 4.7.

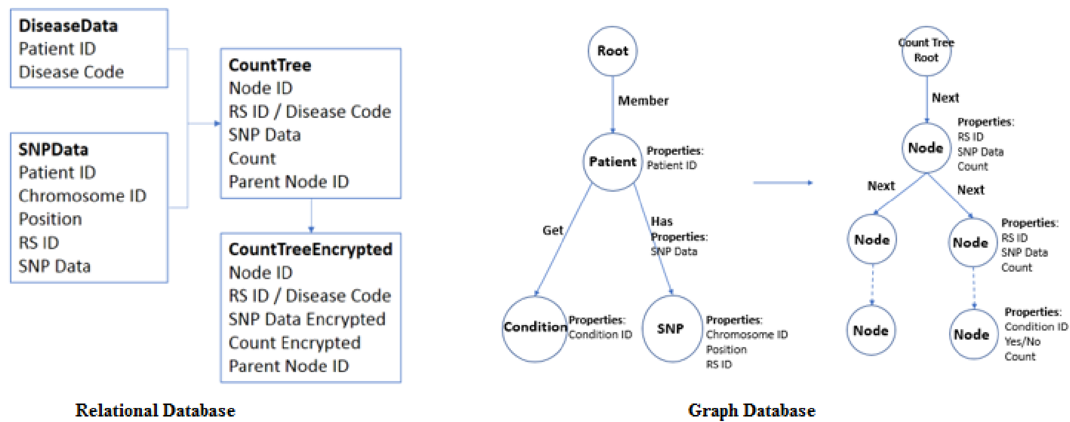


Figure 4.1: Representation of relational and graph database

4.2 Preliminaries

In this section, we introduce some of the concepts (related to cryptography and genome data) required in understanding the proposed method.

4.2.1 Data Representation

In this work, we consider *Single Nucleotide Polymorphism* (SNP) of human DNA and its association with specific disease. For example, a mutation in BRCA1/2 genes has been reported to be associated with Breast cancer. A variant in BRCA1, *rs1799950* is one of the *25 SNPs* to express an increased risk for breast cancer [134]. We considered a similar SNP dataset with a specific disease association. The data is represented in Table 4.1.

4.2.2 Graph Database

Graph database uses different interconnected graph compositions to represent the data. In contrast to relational (traditional) database, graph database considers data points as the nodes and the relation between them as edges. This approach has proved much useful [135, 136] in different literature and use cases as most of the

Table 4.1: Example genomic data containing multiple patients and corresponding SNPs

	Genomic Sequence					Phenotype
Patient	SNP_1	SNP_2	SNP_3	...	SNP_5	Disease
1	A	T	G		C	Yes
2	T	C	C		G	No
3	A	T	C		C	No
4	A	C	C		C	Yes

relational data can be represented as a hierarchical data where one record is closely related to another. Graph database consists of nodes and edges where the nodes are interconnected with edges. Furthermore, there might be directional edges defining the connectivity of the nodes, though for simplicity we will only consider the non-directional edges throughout the rest of the chapter. Regardless of the directions, the edges usually represent the relation between the nodes. In Figure 4.1 we depict the difference between a relational and graph database.

Formally, in a graph database (compared to relational tables), there are *relationships* which connect the *entities*. These entities can have specific properties. The relationships commonly described by verbs, for example, a patient 'get' certain conditions or a patient 'has' many SNPs. A relationship also has properties, for instance, the property 'has' describes the detail data of SNP.

4.2.3 System Architecture Overview

We consider four different parties involved in our proposed architecture:

1. *Data Owners:* Data owners are the parties who sequence human genomic data. They have the proprietary rights over the data. Due to their technical limitation or the data aggregation requirements, they do not share the data directly. Instead, they hand over (or outsource) the data to the certified institutions.

2. *Certified Institution:* The certified institution is the trusted entity who generates and manages the cryptographic keys and responsible for the security of the

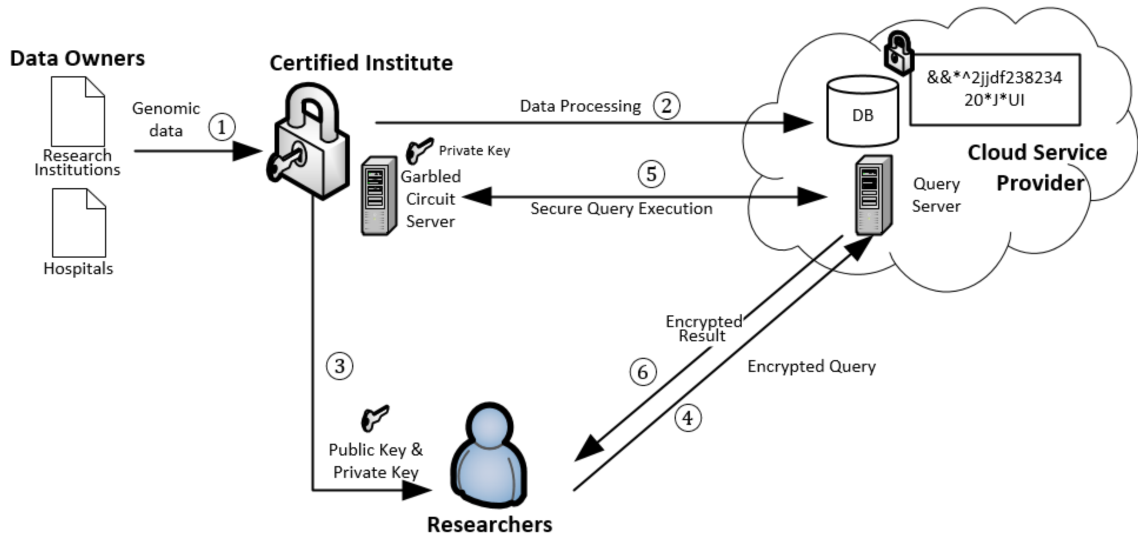


Figure 4.2: System Architecture and secure query process

proposed solution. We assume that a government organization such as NIH can play the role of a certified institution.

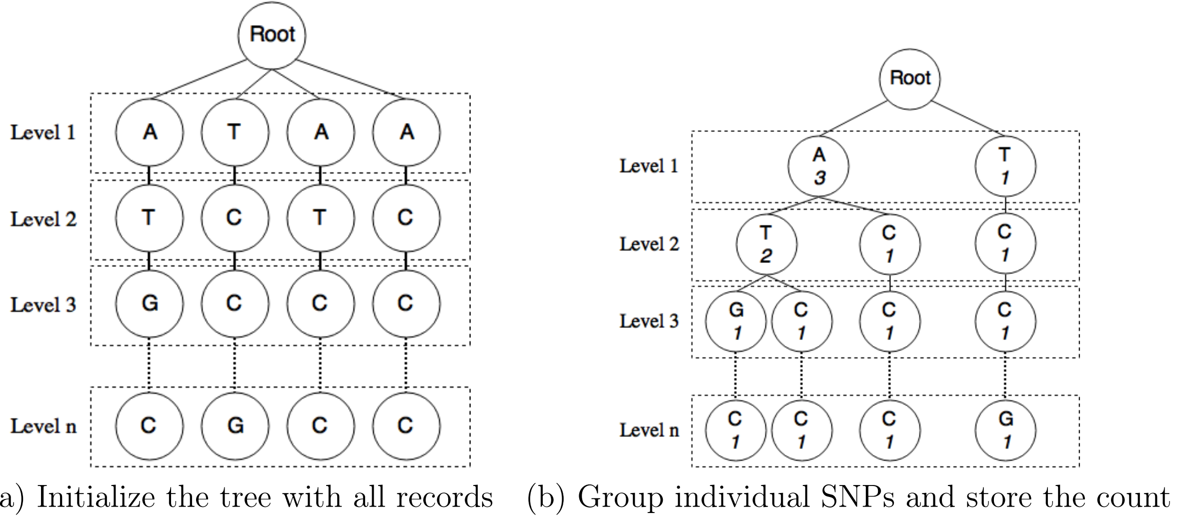
3. *Cloud Service Provider:* Cloud is responsible for storing and executing different queries over the encrypted data. We assume that the cloud service provider is a semi-honest entity and it only receives the public key. Hence, the cloud is unable to decrypt the encrypted data or the query.

4. *Researchers:* Researchers gain access to the query system from the Certified Institution. They acquire the public key to encrypt their query data and private key to decrypt the corresponding results.

Our proposed architecture is shown in Figure 4.2. We briefly overview the major steps of the architecture below:

- *Key distribution:* The certified Institution sends the public key to the Cloud and distributes the public and private keys to authorized researchers.

- *Data processing:* This task is done by the certified institution (CI) prior to sending the encrypted data to the cloud. Initially, the CI collects the data from different data owners. Then, CI builds a count tree on the aggregated data and



(a) Initialize the tree with all records (b) Group individual SNPs and store the count

Figure 4.3: Building the tree from genomic sequence according to Algorithm. The numbers under SNP values in (b) are the counts

uses the count tree to build the index. Finally, it encrypts the tree and data before sending it to the cloud.

- *Query execution:* The cloud receives an encrypted query from researchers and executes them on the encrypted index. This operation is performed by executing a garbled circuit protocol (Section 2.2.2) between the Cloud and researcher. Finally, the Cloud sends the encrypted query result to the researcher who decrypt the result using the private key.

4.3 Methods

In this section, we describe the techniques applied to represent the genome data in a graph database. Furthermore, we explain the secure execution of count query using two cryptographic primitives (HE and GC) described in the earlier section.

4.3.1 Data Preprossessing

Initially, we consider the data to be in a raw tabular format similar to Table 4.1 and stored in a text file. In some earlier attempts [133, 137], such row-wise data were

preprocessed to a relational table and stored in a SQL database. In this work, we incorporate a graph database to store such data and essentially convert the relational tables into a tree structure. This approach is more realistic as we can model the data into three entities: a) patients, b) conditions (diseases), and c) SNPs. Then these entities are connected with relationships such as a specific patient has a particular condition and several SNPs.

4.3.2 Counting Tree Construction

We generate the tree containing all the SNPs and the patients from a comma-separated values (CSV) file (similar to Table 4.1). We outline the algorithm to import the data to in Algorithm 3. Initially, we also sort the SNPs according to their Entropy so that the SNPs with more diversity are considered later when constructing the tree. Then, we generate an empty tree node and mark it as the root. Then for all SNPs from each patient (row-wise in Table 4.1) are linked sequentially in the tree. The first SNPs will be linked back to the root node (Figure 4.3a).

Subsequently, for each level of the tree, we group the nodes by their values (nucleotide values 'A/T/G/C') and keep the unique ones. Thus, the resulting tree will only contain unique nodes on a particular level, along with the number of occurrences. For example, if the nucleotide 'A' appears 3 times in level 1 of the tree, the aggregated node will have 3 as the count value. Figure 4.3a shows the initialization of the tree where we create all the nodes according to the CSV file (Table 4.1). Then,the nodes are aggregated only storing the unique SNP values in Figure 4.3b. The algorithm for creating this Counting Tree is provided in Algorithm 3.

This process is also much simpler than the earlier work [133] where the authors opted for processing the data row-wise. On the contrary, we considered the data column-wise and grouped on each level according to their nucleotide values. Here, we reordered the tree after generating the tree as it reduces the size of the final counting tree. The ordering according to the SNPs entropy also ensures that the early levels of the tree has less branching, which positively impacts the size of the tree. This order is also stored for transforming the query sequences as well.

Algorithm 3: Generate a Counting Tree structure from human genome data

```
   Input: CSV file of SNP datase
   Output: Counting Tree in graph database
 2 Sort the SNPs according to their entropy
 3 Create empty root node
 4 foreach patient do
 5 |   Load SNP sequence with default quantity 1
 6 |   Link SNP sequence using relationship 'Next'
 7 |   Link the first SNP in the sequence to root node
 8 end
   // a tree generated but has all nodes with quantity 1
 9 for i ← 1 to h // h is the height of the tree do
10 |   N ← Group SNP data of level i by the same SNP value
11 |   foreach N where sum(N) > 1 do
12 |   |   Create a new node n with SNP value and count Link node n to the
   |   |     parent level delete the old nodes N Link those child nodes whose
   |   |     parent in N to new node n
13 |   end
14 end
```

4.3.3 Indexing the Counting Tree

We propose an indexing scheme on top of the tree structure compared to the earlier work from Hasan *et al.* [133]. In our search algorithm, one of the important feature is to confirm the linkage between a parent node and his children. The fundamental tree search functions result in a logarithmic runtime without any indexing [138], while the runtime in our graph database is linear to the depth or height of the tree. Thus, experimental results (Figure 4.6) demonstrates that on a million depth tree constructed from genome data (according to Algorithm 3) takes around 10 minutes from root to leaf nodes (empirically).

Our indexing scheme assigns *position tags* and stores the corresponding *range information* into the nodes along with the other information such as nucleotide value, count of patients. Initially, we take all the nodes residing at the same level (siblings) and number them sequentially. We show this next to the nucleotide value inside parenthesis in Figure 4.4. Then, the range of the child nodes are added to the data

Algorithm 4: Number of records with specific SNP values

```
   Data: Search string, S (formatted as SNP1=A and SNP2=T ...
   Result: The encrypted count of patients that meet the query string
 2 Function SearchTree (S)
 3   Q ← Parse the search string, S  // Q will be an array representing
       the query
 4   Sort Q by the depth of the SNP according to the Counting Tree
 5   result ← 0, range ← [0, ∞]
 6   foreach SNP_i in Q do
 7       N ← Nodes in SNP_i // traversal utilizes position tags and
           ranges Q_SNP ← the encrypted value of the SNP nucleotide from the
           query
 8       tmp ← 0
 9       foreach node n in N do
10           currentRange ← range stored in n
11           if currentRange is between range then
12               t_n ←encrypted nucleotide encoding+r_n // random number r_n
13               if r_n == t_n − Q_SNP then
14                   tmp ← 1
15                   result ← the encrypted count value stored in n
16                   range ← currentRange
17               end
18           end
19       end
20       if tmp = 0 then
21           return E(0)// no matches were found
22       end
23   end
24   return result
```

of each node. It is noteworthy that this range information inherits all the position tags of its underlying children.

Figure 4.4 and 4.5 details the corresponding steps which we describe in details here. The process works in two steps: First, we traverse the tree level-wise and assign an incremental number to each node. For example, in level 1 we assign 0 and 1 to the two nodes (A, T) available. Sequentially, at level 2, we assign 0, 1 and 2 to the nodes T, C and C respectively. Thus, we label each of the nodes residing at the same level of the newly constructed tree.

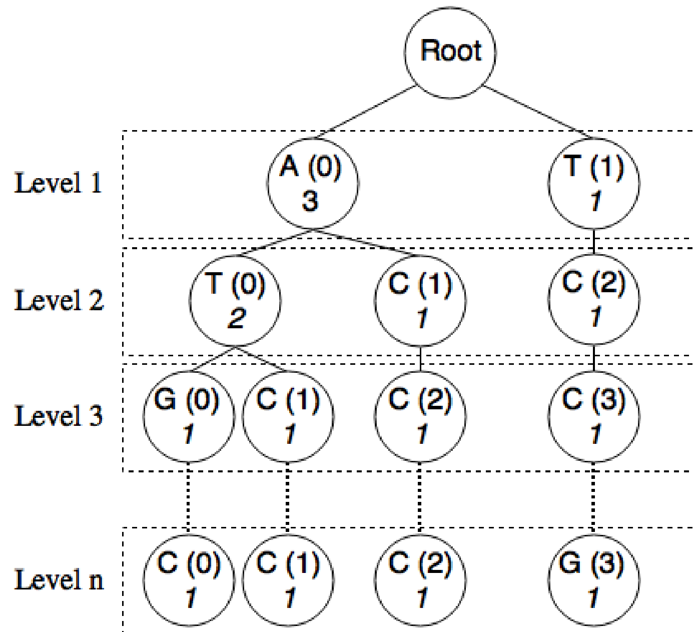


Figure 4.4: Setting position tags on each level and their nodes. The numbers in the parenthesis are sequential labels (position tags) of each layer to be used for generating node ranges for quick indexing

In the second step, we start from the last level which denotes the n-th SNP of each patient. These are the leaf nodes of the tree where there are no children nodes. We assign the range of nodes according to their position tags and its connection with their children. For the leaf nodes, the range denotes their sequence number. For example, in Figure 4.5 (from Table 4.1), the first leaf node has a range of $(0, 0)$ which denotes the first sequence. Its sibling $C(1)$ also represents the second sequence. As both these leaf node has the same grand-parent $T(0)$, it will include the whole range of sequences $(0, 1)$. Therefore, all parents include the range of their children positions. The root node will contain the range for all sequences appearing on the tree which is $(0, 3)$. These range information are kept alongside the SNP nucleotide value, counts and the position as mentioned above.

During the validation, we only need to compare the ranges of parent and child nodes. If the range of the parent node covers the child's, then they are connected. For example, node A in level 1 has a range of $(0, 2)$, so it has connectivity with the child nodes with position 0 to 2. Thus even the leaf nodes that belong to this range $(0, 2)$

are connected to node A in level 1. However, any node having a position not included in the range $(0, 2)$ are not connected. For example, the leaf node G at position 3 is not covered by A's range in level 1 ($(0, 2)$). Thus, this denotes that (leaf) node G is not connected with A (level 1).

4.3.4 Encryption of the Tree

The SNP nucleotide value and count will then be encrypted to protect the privacy of the data. We use the additive homomorphic encryption scheme, *Paillier* to encrypt the nucleotide and count values of each SNPs. However, before encrypting the value of the nucleotides we utilize a numerical encoding for each value A, C, G, T to $0, 1, 2, 3$ respectively. Therefore, an encryption, $E(A)$ will be stored as $E(0)$ in the counting tree. This encoding scheme will also be public as the researcher will know the encoding of $\{A, C, G, T\} = \{0, 1, 2, 3\}$.

As the applied cryptographic scheme (Paillier [49]) produces a randomized ciphertext, as they are indistinguishable under chosen plaintext attack. Therefore, the same numeric values for the SNPs will have different ciphertexts after encryption. For example, encryption of 'A' (or 0) will be different each time and seemingly random. Thus, an adversary cannot distinguish between two encryption of the same value (known as semantic security [49]). Paillier is also partially homomorphic which allows additions under encryption.

4.3.5 Search Operation

In our framework, the search operation is based on queries like 'How many patients are there with SNP1=A and SNP2=C ...' from a particular dataset containing a specific disease. In our scheme, the cloud server has the public key only where the researcher has the private key as well. We utilize the tree structure (and index) mentioned above and encrypt the data of the query (with encoding) accordingly. Our proposed method uses reference SNP IDs *(rs ids)* [139], which is equivalent to chromosome and position in the following example.

Initially, the researcher encodes his/her query parameters such as $SNP1 = 0$,

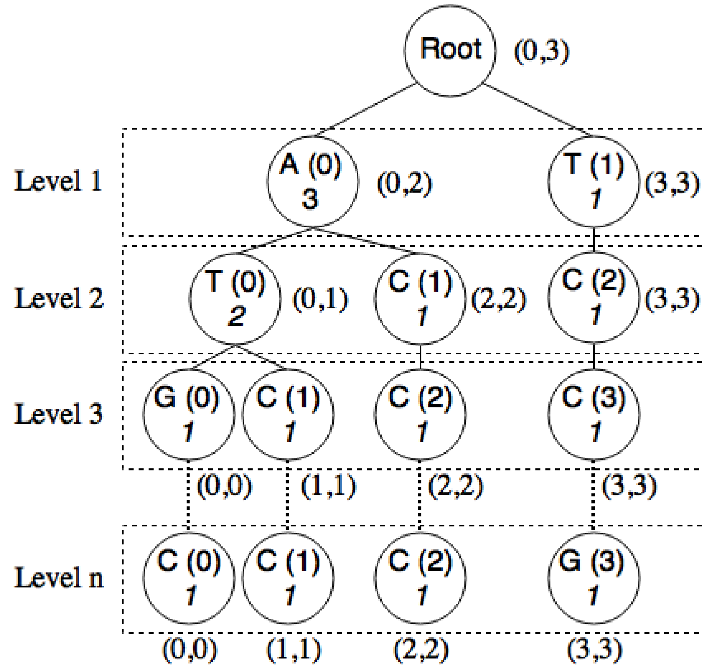


Figure 4.5: Range of position tags from underlying child in each nodes. The range of each node is the union of ranges of its children

$SNP2 = 1$. Then it encrypts them as $SNP1_A = E(0), SNP2_C = E(1)$ and sends to the cloud server as:

```
SELECT count(*) FROM Sequences
    WHERE SNP1=E(0) AND SNP2=E(1) AND ...
    AND Disease = E(1)
```

Here, the presence of the disease is also encrypted as a boolean value of 0 or 1. The cloud server separates the incoming query parameters (i.e., $SNP1_A$ = *E(0)* and $SNP2_C$ = *E(1)* ...) and sort them according to the tree order based on the entropy as discussed in Section 4.3.2.

For example, based on the tree in Figure 4.5, SNP2 is the child of SNP1, the array would be queried in the order of $[SNP1_A = E(0), SNP2_C = E(1)]$. For each of these query SNP positions, we search along the tree with the assistance of the index created by position tags and ranges. For SNP1, let us assume it is positioned on the first level of Figure 4.5, which has two nodes A and T. The cloud service provider generates two random values $r1$ and $r2$, which are added to the SNP values of A and

T, $SNP1'_A = E(0 + r1)$ and $SNP1'_T = E(3 + r2)$. These values are returned to the researcher, who subtracts its encrypted $SNP1_A$ and retrieves two random numbers $r01 = decrypt(SNP1'_A - SNP1_A)$ and $r02 = decrypt(SNP1'_T - SNP1_A)$. This subtraction is essentially an inverse addition ($SNP1'_A = E(-0 - r1)$) and possible with the additive property of partially homomorphic Paillier Encryption scheme.

A Garbled Circuit protocol is then executed to check whether $r01 = r1$ or $r02 = r2$. As only $r1 = r01$ is true in this case, the cloud server only proceeds further on the left side and checks the branches connected to $SNP1 = A$. However, which equality is true is only known to the cloud server and not revealed to the researcher. Suppose, $SNP2$ is positioned on Level n (due to the sorting of SNPs), we only need to check the three C nodes under the branch of node A in Level 1 (as their position ranges are falling between 0 and 2, which follows the range of children for node A). There is no need to check on the other node with SNP value G at Level n, which has a position range of (3,3), outside the child range of node A at Level 1.

The same verification procedure for SNP1 is repeated for SNP2 and the counts on the satisfying nodes (with SNP values equal C) are summed up at the cloud service provider to be E(3) because there are three C nodes under the branch of A (at Level 1) falling between 0 and 2. The final layer is about the disease/phenotype diabetes, which has a binary value (yes/no). If 2 out of the 3 C nodes from Level n have 1's (1 means positive), the final count will be E(2). This encrypted value is returned to the researcher, who gets a final count of 2.

4.4 Results

For the experiments, we used a realistic and large-scale genome dataset from PGP [140]. The dataset had 173 patients, each with 736,317 (∼0.75 million) SNPs.

4.4.1 Experimental Setup

We utilized the cloud services from Amazon *AWS m4.xlarge* instances (4 CPUs, 16GB memory, 500GB disk space) to store and perform the required computations.

Table 4.2: Operations and their required time

Operation	*Time*
Load raw data and preprocessing	3 hours
Building the tree structure	8 hours
Adding position tag and range values	8 hours
Encryption of the tree nodes	5.5 days

Table 4.3: Size of different elements of the Counting Tree in the Neo4J database

Store Sizes	Size (GB)
Node Store	1.80
Property Store	27.97
Relationship Store	12.13
String Store	170.37
Total Size	*212.27*

Furthermore, for comparison with the earlier works [132, 133], we executed these algorithm with 7 patients and 736K SNPs in the same environment. Unfortunately, the programs ended with 'Stack Overflow' error during the early phase of encryption process. It indicated that the encrypted data is unduly excessive to be handled in the main memory. This also motivates the utility of using external memory and a database system like GraphDB. In Table 4.2, we show the running times of the different phase.

4.4.2 Storage Requirements

The generated tree from the aforementioned dataset generated 120 million nodes and required *223.41 GB of disk space*. We used Neo4j as the graph database on a Linux system. Table 4.2 indicates that the most time-consuming task was the encryption of the contents of the nodes. Hence, we utilized a multi-threaded architecture where multiple nodes were encrypted at the same time due to the non-atomic nature of the process. Furthermore, this process can be made faster using clustered programs on several cloud servers.

Table 4.3 scrutinizes the space requirements of different component of the

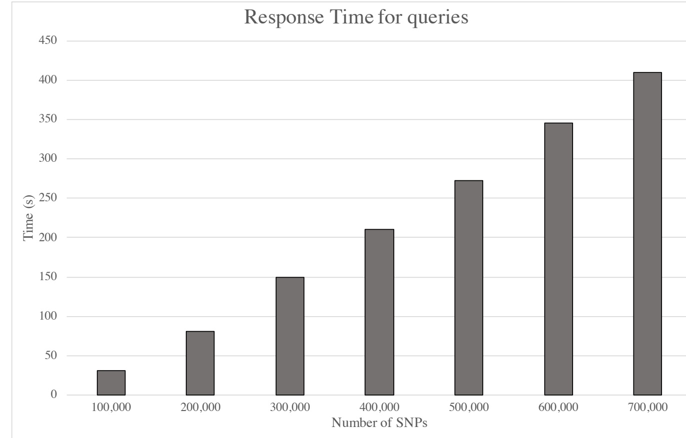


Figure 4.6: Execution time (seconds) for searching *one* leaf node on different number of SNPs in the Counting Tree

Counting Tree on the database. As the content of the nodes (SNP values) are encrypted, it takes the most space (*String Store*) of 170 GB. This can be reduced with different encryption scheme which is not under consideration in this work.

4.4.3 Execution Time

In Figure 4.6, we demonstrate the execution time for searching one node in different number of SNPs available in the database. We selected leaf nodes as the query SNPs to search as this would result the time required to traverse the whole tree. Evidently, the execution time of the count query increases with more SNPs, though it takes 410 seconds (6 minutes) to search in around 700k SNPs. For a query on 100k SNPs, it takes only 31 seconds and it generated around a million nodes on the tree.

In Figure 4.7, we depict the effect of the query size on any given query. Here, we experimented with different number of SNPs on the query sequence (*50, 100, 200, 500*) and analyze the execution time of retrieving the results. Furthermore, the effect of caching on the graph database was considered as well. We evaluated three scenarios:

1. *ColdDB:* Execution of a query with no caching

2. *HotDB:* Execution of multiple queries with *same* SNPs (full caching)

Table 4.4: Relationship of the execution time with query size on different scenarios

	SNPs in the Query			
Scenarios	*50*	*100*	*200*	*500*
Parse Query	51	95	277	519
ColdDB	86	631	998	2090
HotDB	24	33	74	93
WarmDB	35	47	87	270

3. *WarmDB:* Execution of multiple queries with *random* SNPs (Figure 4.7)

Table 4.4 shows the effect of the aforementioned three scenarios where parsing a query depends on the number of SNPs and takes a significant amount of time. Caching effects are also available in this result where a fully cached query is returned much faster than the other two.

One critical implication of constructing a tree from genomic data is the increment in the number of nodes related to the patients and their SNPs. In Figure 4.8 we depict the number of nodes required for storing total 736,317 SNPs and different number of patients. It is apparent from the figure that the number of the nodes (proportional to storage overhead) are not quite linear in order to the number of patients. For example, the system required 800,074 nodes for 5 patients where for 10 patients it needed 1,512,961 nodes. Though, in worst case, the expansion can be $2^{736,317}$ (if we see every variant of bi-allelic SNP), in our case we only observed 8,283,083 nodes after constructing the full tree for 173 patients. This is much smaller than the worst-case scenario as the ratio of the bi-allelic SNPs follow beta distribution [141].

4.5 Discussion

Limitations: Regardless of the encrypted sensitive data or the node values, we are not immune to these security leakages:

1. *Search pattern of the researcher:* Since the tree traversal depends on the GC outputs and researcher's query input, the corresponding path will be revealed to the Cloud. It is important to analyze this leakage as even with this search

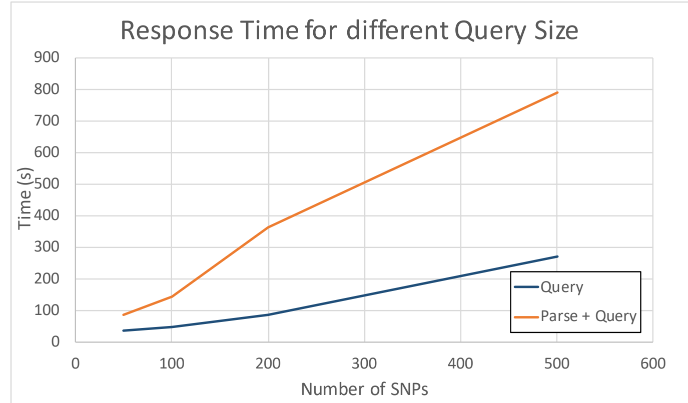


Figure 4.7: Execution time (seconds) for searching different number of SNPs (randomly selected) in the Counting Tree

pattern the cloud will not acquire the sensitive information unless it colludes with the researcher. Furthermore, as all the nucleotide values are encrypted to random ciphertexts, the cloud server cannot infer any information about them. ObliviousRAM [142, 143] concepts can be used to mitigate this issue which will add additional computational complexity.

2. *Dishonest researcher:* In this work, we do not consider malicious researchers as they have the private key which decrypts the results. We can overcome this at an additional cost of involving the data owner on every decryption. This will incur further communication and computational cost which will be deemed inefficient.

Future Work: The principal direction for extension will be utilizing the proposed framework to answer complex queries. Though count queries are the building blocks of different statistical analysis (i.e., GWAS), different aggregation functions might also be useful in many cases [144]. The other key area of interest might be performing different machine learning algorithms.

Regarding the crypto primitives, instead of Paillier [49], we can analyze recent homomorphic schemes[145]. This will reduce the ciphertext size and speed-up the encryption time which seems to be a performance overhead.

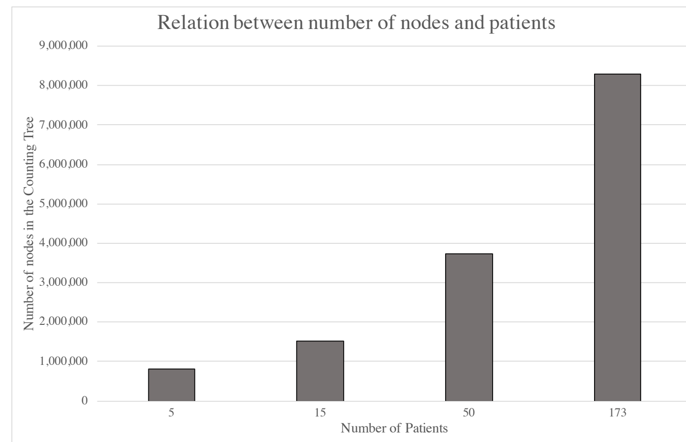


Figure 4.8: Increment of the number of nodes along with number of patients $\{5, 15, 50, 173\}$

4.6 Related Work

Our proposed methods offers significant modifications to Hasan *et al.* [133] where the authors proposed a solution for secure count query on encrypted genomic data using an Index tree. Their Index tree is a subset of our Counting tree but it lacks the indexing scheme proposed in Section 4.3.3. Their encryption and security models are similar as both of these works are provably secure under the semi-honest trust model [146]. However, our storage scheme has a significant difference as Hasan *et al.* solely relied on volatile memory. Previously, Hasan *et al.* 's method was only tested in a small database containing 300 SNPs. So, we also tested on a large number of SNPs to see the scalability of methods. As a result, Hasan *et al.* 's method was not practical for real size genome data (e.g., $736,317$ SNPs in our setting) and their program ended with memory error during the early phase of encryption process, while ours took less than *50* seconds for a query consisting of 100 SNPs (on the same 16G RAM environment).

Similar problem of secure outsourcing and count query execution has been proposed by Ghasemi *et al.* [147], Canim *et al.* [148] and Kantarcioglu *et al.* [149] in 2016, 2012 and 2008 respectively. In most of these works, data were kept encrypted though un-indexed and tested in smaller datasets. However, in reality, providing

Table 4.5: Comparison of related works on Secure Count Query chronologically. It is noteworthy that we experimented with a real world dataset and our scheme is invariant to the number of records

Authors	Year	Method	SNPs	Time (s)
Kantarcioglu *et al.* [149]	HE	2008	40	6900
Canim *et al.* [148]	Hardware	2012	50	600
Hasan *et al.* [133]	HE, GC	2016	300	6
Ghasemi *et al.* [147]	HE	2016	50	90
Our Work	HE, GC	2017	$736,317$	40

security and efficiency in a realistic size of genomic data is much harder and reflected in our work. The prior works are summarized in Table 4.5.

4.7 Conclusion

In this chapter, we demonstrate a realistic use case of secure genome data storage and retrieval application using graph database. Our new mechanisms are more scalable compared to the previous work due to the proposed indexing schemes. A demonstration responding arbitrary queries on different number of SNPs from $\sim 700k$ SNPs (per person) within one minute shows the feasibility of our methods. However, the encryption mechanism (offline) is a major bottleneck of the scheme considering frequent database updates. This can be replaced by the recent state of the art HE mechanisms [46, 145] to improve efficiency.

Chapter 5

Genomic String Search using Parallel Fully Homomorphic Encryption

5.1 Introduction

Fully Homomorphic Encryption (FHE) [38] has been an active area of research in modern cryptography on its own right. FHE cryptosystems provide a strong security guarantee and can compute arbitrary number of operations *over encrypted data*. Due to the emergence of various data-oriented applications [150, 151, 152] on sensitive data, the idea of *computing under encryption* has recently gained momentum. Therefore, FHE is the ideal cryptographic tool that addresses this privacy concern by enabling computation on encrypted data.

Regardless of the promises from the cryptosystem, FHE is still prohibitively slow when applied to generic computations. As a result we do not see a widespread adoption of FHE and very limited real-world applications. For example, addition of two 32-bit encrypted numbers requires around 7 seconds whereas multiplications are even slower; taking around 8 minutes (Table 5.1). Therefore, to utilize FHE in any real-world application, it needs to get a speed-up; whether from a theoretical perspective (reduced computational complexity) or with parallel computations.

In this work, we target a parallel framework for FHE computations using the Graphics Processing Units (GPUs). In recent years, GPU architecture has positively

impacted different machine learning algorithms by speeding up the model training mechanisms over large datasets. Taking similar route, we utilize the multi-core architecture of GPUs and propose a parallel FHE framework employing on the Torus FHE (TFHE) cryptosystem [32, 93].

In this work, we intend to connect FHE operations on genomic data as several security issues has surfaced while keeping such sensitive data in plaintext [26, 27]. For example, keeping the genomic data under encryption, while storing or computing should provide a better security in face of a data breach or compromised system. Therefore, alongside the speed-up, we also extend the proposed parallel FHE framework for three string search operations: Hamming Distance, Edit Distance and Set-Maximal Matches. These search algorithms play a key role in several applications such as Ancestry Search [153] and Similar Patients Query [154, 155] operating on sensitive private data. The effectiveness of these search operations are paramount as they recently uncovered a high-profile crime case known as the 'Golden State Killer' [156].

Undoubtedly, our genomic data have several privacy issues as they can uniquely identify us and reveal our ancestry and certain disease susceptibility which can be deemed private. Therefore, we cannot share or store genomic data without any security guarantee [1]. Hence, an efficient and secure FHE framework can play a vital role in securing the data which is another objective of this Chapter. Here, we divide the underlying contributions into two steps: a) propose a parallel FHE computation framework, and b) string search operations using the proposed framework. We highlight the major contributions of this work below:

Contributions

- At first, we extend the boolean gates (*i.e.*, XOR, AND etc.) from an existing FHE framework [32] to secure algebraic circuits such as addition and multiplication.
- We utilize the recent advancements of GPU architecture and propose parallel FHE operations. Furthermore, we propose several optimizations such as bit

Table 5.1: A comparison of the execution times (sec) of TFHE [6] and our CPU, GPU framework for 32-bit numbers

	Gate Op.	Addition		Multiplication	
		Regular	Vector	Regular	Matrix (mins)
TFHE [32]	1.40	7.04	224.31	489.93	8,717.89
CPU-Parallel	0.50	7.04	77.18	174.54	2,514.34
GPU-Parallel	0.07	1.99	11.22	33.93	186.23

coalescing, compound gates, and tree-based additions to implement the secure algebraic circuits.

- We have conducted several experiments to compare the execution time of the sequential TFHE [32] with our proposed GPU parallel framework. From Table 5.1, our proposed GPU || method is 14.4× and 46.81× faster than the existing technique for regular and matrix multiplications, respectively. We have also benchmarked our performance with existing TFHE frameworks on GPU, namely, cuFHE [53], NuFHE [54].

- Finally, we target different string search operations in genomic dataset (Hamming Distance, Edit Distance and Set-Maximal Matches) and perform them under encryption. Experimental results show that it takes around 12 minutes to perform Hamming distance and Set-Maximal matching on two genomic sequences with 128 genomes. Furthermore, for 8 genomes, the framework takes 11 minutes for an edit distance operation whereas it took 5 hours on an earlier attempt from Cheon *et al.* [157].

Notably, existing GPU enabled TFHE libraries, cuFHE [53] and NuFHE [54], have implemented the TFHE boolean gates using GPUs, whereas our goal was to construct an optimized arithmetic circuit framework. Our design choices and algorithms reflect this improvement and resultantly, our multiplications are around 3.9 and 4.5 times faster than cuFHE and NuFHE, respectively. The code is readily available at `https://github.com/UofM-DSP/CPU-GPU-TFHE`.

The rest of the work is organized as follows: We discuss the required background of the work in Section 5.2. Section 5.3 discusses the underlying methods including

the GPU-parallel framework and the string search operations using such parallel operations. In Section 5.4, we show the experimental analysis whereas Section 5.5 discusses them in details. Section 5.6 denotes the related works and finally concluded in Section 5.7.

5.2 Preliminaries

In this section, we describe the employed cryptographic scheme, TFHE [32] and later define the string search problem.

5.2.1 Torus FHE (TFHE)

We employ Torus FHE (TFHE) [32] in this work where the plain and ciphertexts are defined over a real torus $\mathbb{T} = \mathbb{R}/\mathbb{Z}$, a set of real numbers modulo 1. The ciphertexts are constructed over Learning with Errors (LWE) [56] and represented as Torus LWE (TLWE) where an error term (sampled over a Gaussian distribution χ) is added to each ciphertext. For a given dimension $m \geq 1$ (key size), secret key $\mathbf{s} \in \mathbb{B}^m$ (m-bit binary vector), and error $e \in \chi$, an LWE sample is defined as $(\mathbf{a}, b)$ where $\mathbf{a} \in \mathbb{T}^m$ *s. t.* $\mathbf{a}$ is a vector of torus coefficients of length m (key size) and each element $\mathbf{a}_i$ is drawn from the uniform distribution over $\mathbb{T}$, and $b = \mathbf{a} \cdot \mathbf{s} + e$. The error term ($e$) in LWE sample grows and propagates with the number of computations (*e.g.*, addition, multiplication). Therefore, bootstrapping is introduced to decrypt and re-encrypt the ciphertexts under encryption to remove the noise.

TFHE considers the binary bits as plaintext and generates LWE samples as ciphertexts. Hence, LWE sample computations in ciphertext are analogous to binary bit computations in plaintext. As a binary vector represents an integer number, an LWE sample vector ($\mathbb{L}^n$) can represent an encrypted integer. For example, an n-bit integer becomes n-LWE sample after encryption. Thus, the boolean gate operations of an addition circuit between two n-bit numbers correspond to the similar operations on LWE samples of encrypted numbers. Throughout this chapter, we use *bit* and *LWE sample* interchangeably. Here, we choose TFHE for the following reasons:

- **Fast and Exact Bootstrappping.** TFHE provides the fastest with exact bootstrapping requiring around 0.1s. Some recent encryption schemes [51, 158] also proposed faster bootstrapping and homomorphic computations in general. However, they do not perform exact bootstrapping and erroneous after successive computations on the same ciphertexts.

- **Ciphertext Size.** Compared to the other HE schemes, TFHE offers smaller ciphertext size as it operates on binary plaintexts as shown in Table 5.8. Nevertheless, this minimal storage advantage allowed us to utilize the limited and fixed memory of GPU when we optimize the gate structures.

- **Boolean Operations.** TFHE also supports boolean operations that can be extended to construct arbitrary functions. These binary bits can then be operated in parallel if their computations are independent of each other.

Existing Implementation: The current TFHE implementation comes with the basic cryptographic functions (*i.e.*, encryption, decryption, etc.) and all binary gate operations. Although the gates are computed somewhat sequentially in the original implementation [32], the underlying architecture uses Advanced Vector Extensions (AVX) [50]. AVX is an extension to x86 instruction set from Intel which facilitates parallel vector operations. The bootstrapping procedure requires expensive Fast Fourier Transform (FFT) operations ($O(n \log n)$). The existing implementation uses the Fastest Fourier Transform in the West (FFTW) [159] which inherently uses AVX.
Why TFHE? There have been several attempts in improving the asymptotic performance and numerical operations of FHE [160, 161, 46], which are pivotal to this work (Section 5.6 for details). Torus FHE (TFHE) [32] is one of the most renowned FHE schemes that meets the expectation of arbitrary depth of circuits with faster bootstrapping technique. TFHE also incurs lower storage requirement compared to the other encryption schemes (Table 5.8). The plaintext message space is binary in TFHE. Hence, the computations are based solely on boolean gates, and each gate operation entails a bootstrapping procedure in gate bootstrapping mode.
Why GPU? Most of the FHE schemes are based on the Learning With Errors

(LWE), where plaintexts are encrypted using polynomials and can be represented with vectors. Therefore, most computations are operated on vectors that are highly parallelizable. On the contrary, Graphics Processing Units (GPUs) offer a large number of computing cores (compared to CPUs). These cores can be utilized to compute parallel vectors operations. Therefore, we can utilize these cores to parallel the FHE computations. However, we also have to consider the fixed and limited memory of GPUs (8-16GB) and their reduced computing power compared to any CPU core.

5.2.2 Sequential Framework

In this section, we present a brief overview of the sequential arithmetic circuit constructions using boolean gates as background which we extend later.

Addition

A carry-ahead 1-bit full adder circuit takes two input bits along with a carry to compute the sum and a new carry that propagates to the next bit's addition. Therefore, in a full adder, we have three inputs as a_i, b_i and c_{i-1}, where i denotes the bit position. Here, the addition of bit a_1 and b_1 in $A, B \in \mathbb{B}^n$ requires the carry bit from a_0 and b_0. This dependency enforces the addition operation to be sequential for n-bit numbers [162]. In this work, we have also used half-adders for numeric increment and decrements.

Multiplication

Naive Approach: For two n-bit numbers $A, B \in \mathbb{Z}$, we multiply (`AND`) the number A with each bit $b_i \in B$, resulting in n numbers. Then, these numbers are `left shift`ed by i bits individually resulting in $[n, 2n]$-bit numbers. Finally, we `accumulate` (reduce by addition) the n shifted numbers using the addition.

Karatsuba Algorithm: We consider the divide-and-conquer Karatsuba's algorithm for its improved time complexity $O(n^{\log^3})$ [163]. It relies on dividing

Algorithm 5: Karatsuba Multiplication [163]

Input: X, Y $\in \mathbb{B}^n$
Output: Z $\in \mathbb{B}^{2n}$

```
2  if n < n_0 then
3  |  return BaseMultiplication(X, Y)
4  end
5  X_0 ← X mod 2^{n/2}
6  Y_0 ← Y mod 2^{n/2}
7  X_1 ← X/2^{n/2}
8  Y_1 ← Y/2^{n/2}
9  Z_0 ← KaratsubaMultiply(X_0, Y_0)
10 Z_1 ← KaratsubaMultiply(X_1, Y_1)
11 Z_2 ← KaratsubaMultiply(X_0 + Y_0, X_1 + Y_1)
12 return Z_0 + (Z_2 − Z_1 − Z_0)2^n + (Z_1)2^{2n}
```

the large input numbers and performing smaller multiplications. For n-bit inputs, Karatsuba's algorithm splits them into smaller numbers of $n/2$-bit size and replaces the multiplication by additions and subsequent multiplications (Line 12 of Algorithm 5). Later, we introduce parallel vector operations for further optimizations.

5.2.3 CPU-based Parallel Framework

We propose a CPU || framework utilizing the multiple cores available in computers. Since the existing TFHE implementation uses AVX2, we employ that in our CPU || framework.

5.2.3.1 Addition

Figure 5.1 illustrates the bitwise addition operation considered in our CPU framework. Here, any resultant bit r_i depends on its previous c_{i-1} bit. The dependency restricts incorporating any data-level parallelism in the addition circuit construction.

Here, it is possible to exploit task-level parallelism where two threads execute the `XOR` and `AND` operations (Figure 5.1), simultaneously. We observed that the time required to perform such `fork-and-join` between two threads is higher than executing them serially. This is partially due to the costly thread operations and

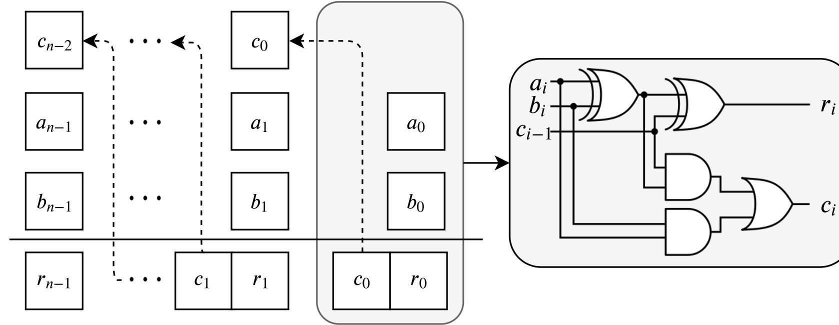


Figure 5.1: Bitwise addition of two n-bit numbers A and B. a_i, b_i, c_i, r_i are i^{th}-bit of A, B, *carry*, and the *result*

eventual serial dependency of the results. Hence, we did not employ this technique for CPUs.

5.2.3.2 Multiplication

Out of the three major operations (`AND`, `left shift`, and `accumulation` (addition) in multiplications, `AND` and left shifts can be executed in parallel. For example, for any two 16-bit numbers A, B ($\in \mathbb{B}^{16}$), and four available threads, we divide the `AND` and `left shift` operation among four threads.

On the other hand, the accumulation operation is demanding as it requires n (for n-bit multiplication) additions. The accumulation operation adds and stores values to the same variable, which makes it atomic. Therefore, all threads performing the previous `AND` and `left shift` have to wait for such accumulation which is termed as global thread synchronization [164]. Given that it is computationally expensive, we do not employ this technique in any parallel framework.

We utilized a custom reduction operation in OpenMP [164], which uses the global shared memory (CPU) to store the in-between results. This customized reduction foresees additions of any results upon completion and facilitates a performance gain by avoiding the global thread synchronization. The custom reduction resulted in much better performance compared to the straightforward approach of waiting on all threads to complete their tasks (global thread synchronization).

5.2.4 String Search: Problem Definition

We are proposing privacy-preserving methods to measure the string distances using Hamming, Edit Distance and Set-maximal matching. We define the query string as q whereas the target genomic sequence is denoted as y. For simplicity, we assume that all sequences have an equal number of m genes where each gene is bi-allelic. Bi-allelic genes are represented as $\{0, 1\}$, resulting query to be a bit vector where $q = [q_1, q_2, \ldots, q_m]$ as $q_i \in \{0, 1\}$. On the other hand, any target sequence is defined as $y = [y_1, y_2, \ldots, y_m]$ as $y_i \in \{0, 1\}$.

In this problem, the query, q and data y are encrypted with Fully Homomorphic Encryption (FHE) scheme [32]. Upon encryption, we denote the query as a vector of encrypted bits and represented with $\mathbf{q}$. The encrypted data $\mathbf{y}$ is hosted in a cloud environment where a researcher is sending his/her encrypted query. Notably, the target can be a set of genomic sequences, denoted by $\mathbf{Y}$. The target is to calculate exact or approximate a string distance score for $\mathbf{q}$ against $\mathbf{y}$ under FHE with a certain algorithm such as Hamming or Edit Distance. Since it is a asymmetric encryption scheme, we assume that the cloud server only has access to the public key. On the other hand, the researcher and data owner has the private key to decrypt the result and encrypt the genomic data, respectively. The targeted string distance metrics are formally defined below:

Definition 5.2.1 (Hamming Distance)**.** The hamming distance $hd(\mathbf{q}, \mathbf{y})$ measures the difference or number of genes that are different in two sequences $\mathbf{q}$ and $\mathbf{y_i}$: $hd(\mathbf{q}, \mathbf{y}) = \sum_{k\in[1,m]}(\mathbf{q}[k] \neq \mathbf{y}[k])$

Definition 5.2.2 (Edit Distance)**.** The edit distance $ed(\mathbf{q}, \mathbf{y})$ between two sequences $(\mathbf{q}, \mathbf{y})$ is defined as the minimum cost taken over all edit sequences that transform query $\mathbf{q}$ into $\mathbf{y}$. That is $ed(\mathbf{x}, \mathbf{y}) = min\{C(s)|s$ is a sequence of edit operations (insert, update or delete) transforming $\mathbf{q}$ into $\mathbf{y}\}$.

Definition 5.2.3 (Set Maximal Distance)**.** A set maximal score or distance $sd(\mathbf{q}, \mathbf{y})$ denotes the maximum number of consecutive matching genes between $\mathbf{q}$ and $\mathbf{y}$, which have the following conditions:

1. there exists some index $k_2 > k_1$ such that $\mathbf{q}[k_1, k_2] = \mathbf{y}[k_1, k_2]$ (same substring);
2. $\mathbf{q}[k_1 - 1, k_2] \neq \mathbf{y_i}[k_1 - 1, j_2]$ and $\mathbf{q}[k_1, k_2 + 1] \neq \mathbf{y}[j_1, j_2 + 1]$, and
3. for all other genes, $k' \neq k$ and $k' \in [1, m]$, if there exist $k'_2 > k'_1$ *s. t.* $\mathbf{q}[k'_1, k'_2] = \mathbf{y}[k'_1, k'_2]$ then it must be $k'_2 - k'_1 < k_2 - k_1$.

The set maximal distance is defined as, $sd(\mathbf{q}, \mathbf{y}) = k_2 - k_1$.

5.3 Methods

In this section, we outline our proposed solutions to compute the string distance metrics for the targeted algorithms. Firstly, we propose the GPU parallel FHE framework on top of which we build the string search operations described later.

5.3.1 GPU-based Parallel Framework

In this section, we present three generalized techniques to introduce GPU parallelism (GPU ||) for any FHE computations. Then, we adopt them to implement and optimize the arithmetic operations. Notably, our CPU || framework is also described in Section 5.2.3.

5.3.1.1 Proposed Techniques for Parallel HE Operations

This section introduces general techniques adopted for the GPU-based parallel framework.

Parallel TFHE Construction: We depict the boolean circuit computation in Figure 5.2. Here, each LWE sample comprises of two variables: $\mathbf{a}$ and b. It is noteworthy that $\mathbf{a}$ is a 32-bit integer vector defined by the secret key size (m) which has a lower memory requirement compared to other FHE implementations (Section 5.6). In our parallel TFHE construction, we only store the vector $\mathbf{a}$ on the GPU's global memory.

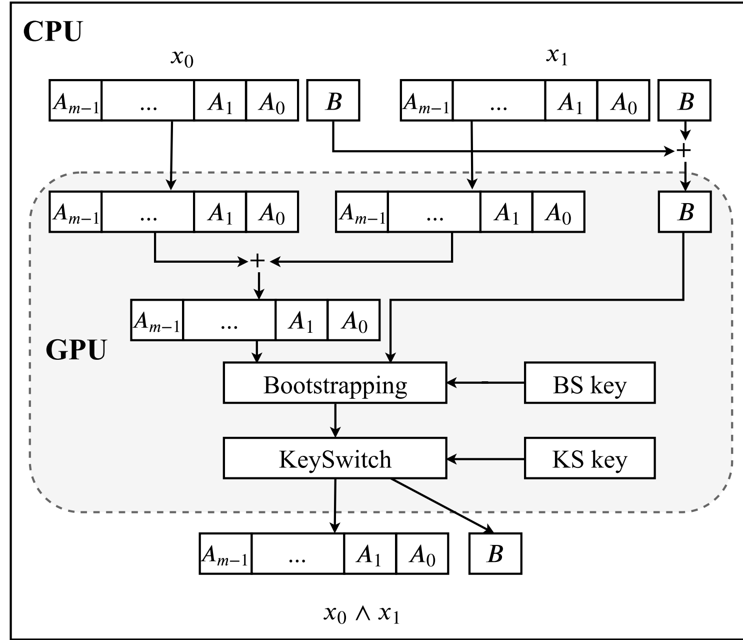


Figure 5.2: Arbitrary operation between two bits where BS, KS key represents bootstrapping and key switching keys, respectively

In addition to all vector operations inside the GPU, we also employ the native cuda enabled FFT library (cuFFT) which uses the parallel cuda cores for FFT operations. Here, the parallel batching technique from cuFFT supports multiple FFT operations to be executed simultaneously. However, cuFFT also limits such parallel number of batches. It keeps the batches in an asynchronous launch queue, and processes a certain number of batches in parallel. This number of parallel batches solely depends on the hardware capacity and specifications [165].

Bit Coalescing (BC): Bit Coalescing combines n-LWE samples in a contiguous memory to represent n-encrypted bits. The encryption of a n-bit number, $X \in \mathbb{B}^n$ requires n-LWE samples (ciphertext), and each sample contains a vector of length m. Instead of treating the vectors of ciphertexts separately, we coalesce them altogether (dimension $1 \times mn$) as illustrated in Figure 5.3.

The intuition behind such construction is to increase parallel operations by extending the vector length in a contiguous memory. Coalescing the vectors increases the vector length but we incorporate more threads to maximize parallelization and

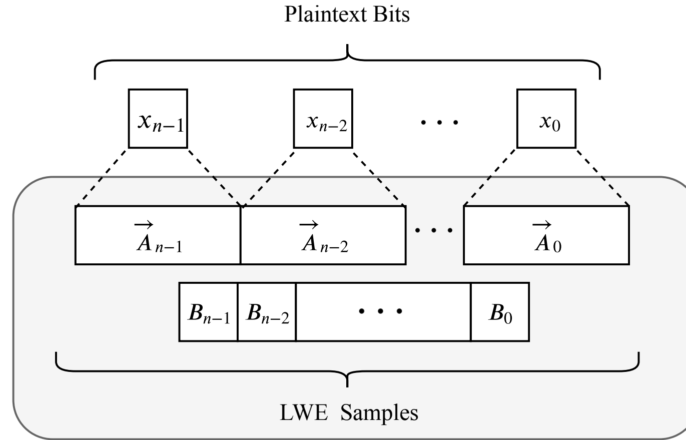


Figure 5.3: Coalescing n-LWE samples (ciphertexts) for n-bits

reduce the execution time.

Compound Gate: Since addition is used in most arithmetic circuits, we propose a new gate structure, *Compound Gates* which allows further parallel operations among encrypted bits. These gates are a hybrid of two gates, which takes two 1-bit inputs as an ordinary boolean gate but gives two different outputs. The motivation behind this novel gate structure comes from the addition circuit. For $R = A + B$, we compute r_i and c_i with the following equations:

$$r_i = a_i \oplus b_i \oplus c_{i-1} \tag{5.1}$$

$$c_i = a_i \wedge b_i \mid (a_i \oplus b_i) \wedge c_{i-1} \tag{5.2}$$

Here, r_i, a_i, b_i, and c_i denotes i^{th}-bit of R, A, B, and the carry, respectively. Figure 5.1 illustrates this computation for an n-bit addition.

While computing the equations 5.1, and 5.2, we observe that `AND` (AND) and `XOR` (OR) are computed on the same input bits. As these operations are independent, they can be combined into a single gate, which then can be computed in parallel. We name these gates as *compound gates*. Thus, $a \oplus b$ and $a \wedge b$ from Equation 5.1 and 5.2

can be computed as,

$$s, c \;= \underbrace{a \oplus b, a \wedge b}_{CONCAT}$$

Here, the outputs of $s =$AND and $c =a$ORb are concatenated. The compound gates construction is analogous to the task-level parallelism in CPU, where one thread performs AND, while another thread performs OR.

In GPU ||, the compound gates operations are flexible as AND or OR can be replaced with any other logic gates. Furthermore, the structure is extensible up to n-bits input and $2n$-bits output.

5.3.1.2 Algebraic Circuits on GPU

The presents different algebraic circuit constructions in GPU-based parallel framework using the general techniques.

Addition: *Bitwise Addition (GPU$_1$):* From the addition circuit in Section 5.2.3.1, we did not find any data-level parallelism. However, we noticed the presence of task-level parallelism for `AND` and `XOR` as mentioned in the compound gates construction. Hence, we incorporated the compound gates to construct the bitwise addition circuit. We also implemented the vector addition circuits using GPU$_1$ to support complex circuits such as multiplications (Section 5.3.1.2).

Number-wise Addition (GPU$_n$): We consider another addition technique to benefit from bit coalescing. Here, we operate on all n-bits together. For $R = A + B$, we first store A in R ($R = A$). Then we compute, $Carry =$AND, $R = R$ORB, and $B = Carry \ll 1$, for n times.

Here, we utilize compound gates to perform AND and RORB in parallel. Thus, in each iteration, the input becomes two n-bit numbers, while in bitwise computation the input was two single bits. On the contrary, even after using compound gates, the bitwise addition (Equations 5.1 and 5.2) has more sequential blocks (3) than the number-wise addition (0). We analyze both in Section 5.4.3.

Numeric Increments and Decrements: We also propose half-adders for numeric increments and decrements which is required for several operations in string search.

(a) Half Adder (b) Half Subtractor

Figure 5.4: 1-bit Increment and Decrement using Half-adder or subtractor where the $\mathbf{x}_i$ is the input Bit and the Carry bit is propagated into the next bit's operation

For example, algorithms 7 and 8, 9, we need to perform increments and decrements of an encrypted number, respectively. We use half adders and subtractors to perform the operations. In Figure 5.4, we show the difference of the operations. For the half adder, we perform the XOR and AND operation for all input bits for the encrypted number while the other input is set to 1 (or 0) under encryption. The only difference for the half-subtractor is that the input bit is inverted before the AND operation which represents the carry bit.

The sign bit for these encrypted numbers (most significant bit), also goes through the same operation as the rest of the bits. However, in this work, we cannot protect the increment against overflow as the number of bits for each encrypted numbers are set prior to the execution. For example, if we are incrementing a 16-bit encrypted number, and it gets a value of $2^{15}+1$ (1-bit reserved for the sign bit), it will not get a correct decrypted value. On the other hand, while decrementing by 1 for Algorithm 8 and 9, we will eventually get into negative numbers which is represented by the sign bit. Therefore, we perform an OR operation in Algorithm 6 on Line 10.

Multiplication

Naive Approach: According to Section 5.2.2, multiplications have AND and $\ll$ operations which can be executed in parallel. It will result in n-numbers where each number will have $[n, 2n]$-bits due to the $\ll$. We need to accumulate these uneven sized numbers which cannot be distributed among the GPU threads. Furthermore, the addition presents another sequential bottleneck while adding and storing ($+=$) the results in the same memory location. Therefore, this serial addition will increase

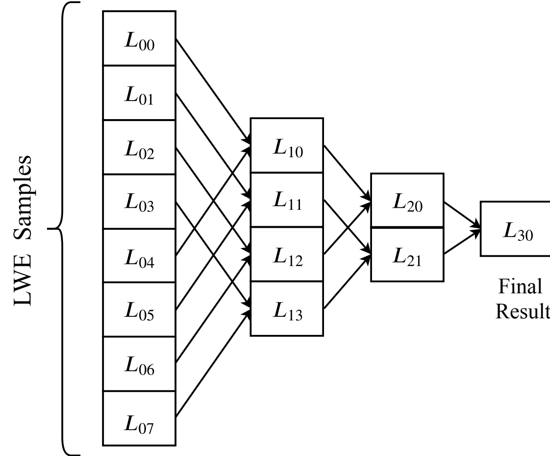


Figure 5.5: Accumulating $n = 8$ LWE samples (L_{ij}) in parallel using a tree-based reduction

the execution time. In the framework, we optimize the operation by introducing a tree-based approach.

In this approach, we divide n-numbers (LWE vectors) into two $n/2$ vectors. This two $n/2$ vectors are added in parallel. We repeat the process as we divide the resultant vectors int two $n/4$ vectors and add them in parallel. The process continues until we get the final result. Notably, the tree-based approach requires $\log n$ steps for the accumulation. In Figure 5.5 for $n = 8$, all the ciphertexts underwent AND and $\ll$ in parallel, and waited for addition. Here, L_{ij} represents the LWE samples (encrypted numbers), i is the level, and j denotes the position.

Karatsuba Multiplication: We used Karatsuba's algorithm with some modifications in our framework to achieve further efficiency while performing multiplications. However, this algorithm requires both addition and multiplication vector operations which tested the efficacy of these components as well. We modified the original Algorithm 5 to introduce the vector operations and rewrite the computations in

Algorithm 6: Determine if Input Number is Greater than Zero

Input: Encrypted number **x** with $|\mathbf{x}|$ bits, boolean flag *hasSign* if **x** has sign bit

Output: One bit representing whether **x** is greater than 0, *result*

```
2  Procedure greaterThanZero(x,hasSign)
3      i ← 0
4      result ← ℰ(0)
5      while i < |x|−1 do
6          result ← result ŌR x[i]
7          i ← i + 1
8      end
9      if hasSign then
10         result ← result ĀND (NŌT x[|x|])
11     end
12     return result
```

Line 9-12 as:

$$\langle Temp_0,\ Temp_1 \rangle = \langle X_0,\ X_1 \rangle + \langle Y_0,\ Y_1 \rangle$$

$$\langle Z_0,\ Z_1,\ Z_2 \rangle = \langle X_0,\ X_1,\ Temp_0 \rangle \cdot \langle Y_0,\ Y_1,\ Temp_1 \rangle$$

$$\langle Temp_0,\ Temp_1 \rangle = \langle Z_2,\ Z_1 \rangle + \langle 1,\ Z_0 \rangle$$

$$Z_2 = Temp_0 + (Temp_1)'$$

In the above equations, X_0, X_1, Y_0, Y_1, Z_0, Z_1, and Z_2 are taken from the algorithm. $\langle \dots \rangle$ and $\cdot$ are used to denote concatenated vectors and dot product, respectively. For example, in the first equation, $Temp_0$ and $Temp_1$ store the addition of X_0, Y_0 and X_1, Y_1. It is noteworthy that in the CPU || framework, we utilized task-level parallelism to perform these vector operations as described in Section 5.2.3.

5.3.1.3 Bit-wise Operations

In this section, the general bit-wise operations required for determining the string distances are discussed. These algorithms will inherit the aforementioned algorithms and extend them accordingly based on the corresponding use-cases:

Greater than Zero

Our string distance methods on encrypted data relies on Algorithm 6, to check whether *input* > 0. Here, the algorithm takes an encrypted number as an input and checks whether it is greater than zero. This allows us to judge whether there are any set bits on the encrypted version of the number. In order to output that result, in line 6, an encrypted bit-wise OR ($\bar{OR}$) operation is performed between an encrypted bit $X[i]$ and the current $result$.

The final result also considers the sign bit as the number can be negative. Here the sign bit is set as the most significant bit (MSB) or $X[|X|]$ which is inverted and placed on another OR operation with the $result$ variable. To get whether the input is less than 0 is also can be achieved by this bit. Notably, the value of the $result$ is kept encrypted throughout the computations which is utilized in the upcoming algorithms.

Longest Consecutive Ones

Algorithm 7 finds the longest or maximum consecutive set bits of value 1 from an encrypted number or bit-stream. Here, the input encrypted number X is left shifted once on each iteration till we reach the end of the bit stream ($|X|$ many times). The core operation happens on line 6 where the encrypted X is left shifted by one. Following the left shift, we also perform an encrypted bit-wise $A\bar{N}D$ operation with the previous X. Then, we find whether the newly formed X has any 1-bit (whether $X > 0$) and increment a counter. This counter is the representative of the $result$ which uses the aforementioned Algorithm 6 and increments by one if $X > 0$. It is important to note that, this algorithm will only be used only for Set Maximal Distance where encrypted haplotypes are inputted as X. Since `greaterThanZero` takes the sign bit into consideration which is not required for Set Maximal operations, line 10 is ignored in this case.

Lets assume we have an encrypted number $\mathbf{x} = \mathcal{E}(011101)$ with sign bit $\mathcal{E}(0)$ and it contains 3 consecutive ones. Here, the result bit is set to 1 since $\mathbf{x} > 0$. In the first iteration, we perform an encrypted AND operation of $\mathbf{x}$ (011101) and $\mathbf{x} << 1$ (111010). Since the resulting $\mathbf{x}$ is greater than 0, the encrypted $result$ number is

Algorithm 7: Find Longest Consecutive Ones

```
  Input: Encrypted number x
  Output: result representing the number of the longest consecutive ones
2 Procedure maxConsecutiveOnes(X)
3   numbits ← |x|
4   result ← greaterThanZero(x, false)
5   while numbits > 0 do
6     x ← x AÑD (x ≪ 1)
7     result ← result + greaterThanZero(x, false)
8     numbits ← numbits − 1
9   end
10  return result
```

Algorithm 8: Get Minimum Number among $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ encrypted positive numbers ($x_i \geq 0$)

```
  Input: Positive Numbers x_1, x_2, ..., x_n
  Output: Minimum encrypted number result
2 Procedure getMin(x_1, x_2, ..., x_n)
3   numinter ← 2^|x_0|
4   result ← 0
5   while numinter > 0 do
6     gtZero ← E(1)
7     foreach x_i ∈ {x_1, ..., x_n} do
8       gtZero ← gtZero AÑD greaterThanZero(x_i)
9       x_i ← x_i − 1
10    end
11    result ← result + gtZero
12    numinter ← numinter − 1
13  end
14  return result
```

incremented. In the following iteration, $\mathbf{x} = \mathcal{E}(011000)$ is multiplied (AND) with $\mathcal{E}(110000)$ which results in 010000. The result number is incremented again. However, in the subsequent iterations ($|\mathbf{x}|$ many times), the $\mathbf{x}$ values are set to 0 and result is not incremented anymore. Finally, the result from Algorithm 7 is retrieved as $\mathcal{E}(3)$.

Algorithm 9: Get Maximum Number among $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ encrypted positive numbers $x_i \geq 0$

Input: Positive numbers $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$
Output: Maximum encrypted number *result*

```
2  Procedure getMax(x_1, x_2, ..., x_n)
3      numiter ← 2^|x_i|
4      result ← E(0)
5      while numiter > 0 do
6          gtZero ← E(0)
7          foreach x_i ∈ {x_1, ..., x_n} do
8              gtZero ← gtZero ŌR greaterThanZero(x_i)
9              x_i ← x_i − 1
10         end
11         result ← result + gtZero
12         numiter ← numiter − 1
13     end
14     return result
```

Finding Minimum and Maximum Number

Algorithm 8 and 9 are two mirroring algorithms where we target the minimum and maximum number from n numbers, respectively. Here, the encrypted numbers, $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ are decremented by one for each bit ($|\mathbf{x}_i|$ bit size). Then, we check if it has any set bit or its value equals to 0. For finding the minimum number, if *all* numbers are greater than zero, then the encrypted result variable is incremented. On the other hand, to find the maximum number, we performed an encrypted $\bar{OR}$ operations to check if *any* of the numbers are greater than 0.

Here, we need to perform a numeric decrement for both algorithms to achieve $\mathbf{x}_i \leftarrow \mathbf{x}_i - 1$. In this case, a binary half subtractor is employed which is described next. However, these decrements will incur underflow as input numbers $\mathbf{x}_i$s can get negative at any given iteration. Therefore, on Algorithm 6, we have an operation with MSB as `greaterThanZero`($\mathbf{x}_i$) outputs a single bit representing $\mathbf{x}_i > 0$. To retrieve the minimum or maximum among n numbers, this single bit, *gtZero* on Algorithm 8 and 9 is added with the result for all *numbits* times. The final result is under encryption and utilized in Edit Distance Approximation (Section 5.3.2.2) and Set

Algorithm 10: Hamming Distances between a query and encrypted sequences

```
    Input: Encrypted target sequence y and query q
    Output: Encrypted distances between y and q, hd
2   Procedure HammingDistance(q, y)
3   |   result ← q XŌR y
4   |   foreach bit r ∈ result do
5   |   |   hd ← hd + r
6   |   end
7   |   return hd
```

Maximal Matches (Section 5.3.2.3).

Alternative Approach: To retrieve the minimum or maximum numbers, we can also utilize full-adders. For example, the maximum between x and y can be determined by performing $x - y$. If the sign bit is not set, then $x > y$. Similarly, the minimum between two numbers can also be denoted by the sign bit as well. We consider this method due to the exponential number of iterations considering the number of bits in line 3 on Algorithm 9 and 8. Since, $|x| > 16$ will require many rounds of computations under encryption, we utilize the aforementioned algorithms for $|x| \leq 16$.

5.3.2 Secure String Search Operations

5.3.2.1 Hamming Distance

Hamming Distance $hd(\mathbf{q}, \mathbf{y})$ represents the bit-wise difference of the input query $\mathbf{q}$ and stored sequence $\mathbf{y}$. Therefore, we do an encrypted X$\bar{\text{O}}$R operation between $\mathbf{q}$ and $\mathbf{y}$ where the result will have set bits (value of 1) on all occasions of mismatches. Now, we need to perform a summation of all these bits on the XOR result to get the Hamming distance (definition 5.2.1). Notably, we assume that the query $\mathbf{q}$ and the encrypted sequences are of the same length.

In Algorithm 10, we outline the mechanism to generate the hamming distance hd where it contains encrypted distance value for one target sequence $\mathbf{y}$ and query $\mathbf{q}$.

Algorithm 11: Banded edit distance on encrypted sequence

```
    Data: query q, sequence y, and band length b
    Result: b-banded Edit Distance d(q, y) [166]
 1  m ← |q|+1
 2  set each element of matrix d_{m×m} to E(0)
 3  for i ← 1 to m do
 4      d[i, 0] ← E(i);
 5      d[0, i] ← E(i);
 6  end
 7  for i ← 1 to m do
 8      if i − b < 1 then low ← 1;
 9      else low ← i − b;
10      if i + b > m then high ← m;
11      else high ← i + b;
12      for k ← low to high do
13          same_bit ← q[i − 1] XNOR y[k − 1]
14          sub ← d[i − 1, k − 1] + same_bit
15          ins ← d[i, k − 1] + 1
16          del ← d[i − 1, k] + 1
17          d[i, k] ← getMin(sub, ins, del)
18      end
19  end
20  return d[m, m];
```

This can be iterated through all sequences and performs the XOR operation between the query $\mathbf{q}$ and $\mathbf{y}_i$ sequence. Subsequently, it also adds the bits to formulate the hamming distance on line 5. Since, the *result* variable is under encryption, the addition (or increment) is oblivious as we perform the operation for every encrypted bit in *result*.

5.3.2.2 Edit Distance Approximation

Edit distance is more complicated than Hamming Distance as it considers more than the bit-wise difference (insertion, deletion and subtraction). Furthermore, under plaintext, it has a $O(m^2)$ complexity where m is the length of the sequence. Therefore, to reduce the complexity, we opt for Banded Edit Distance [166, 155] where we only compute on a band of fixed size.

Algorithm 11 outlines the proposed method where we set a fixed parameter b along with the encrypted input sequences $\mathbf{q}$ and $\mathbf{y}_i$. Apart from the initialization, we also calculate the variables *low* and *high* that dictates the number of expensive operations in line 11. Here, we calculate whether the $\mathbf{q}[i]$ and $\mathbf{y}_i[k]$ bits are the same or not using an encrypted XNOR gate. If they are not the same then, the encrypted number $d[i-1, k-1]$ needs to be incremented which is done with the half-adder. Since we do not know the output of *same_bit*, we push that bit as carry and initialize the substitution variable. Similarly, the insertion and deletion values are set from the existing distance matrix. Finally, we calculate the minimum `getMin`(ins, del, sub) to predict the distance at that specific position. This is set as the the new value of $d[i, k]$. Here, the three half-adder operations are run in parallel before the minimum operation.

5.3.2.3 Set Maximal Distance

Set maximal distance or match (SMM) represent the length of the longest matching substring in two sequences [106]. This allows a health-care researcher to identify genomic sequences that have more genes in common and probably be identical in their physical attributes. The distance also has applications over similar patients queries [154], secure positional Burrows-Wheeler Transformation [167, 103] etc.

The proposed secure set maximal match using Homomorphic Encryption operation depends on Algorithm 7, `maxConsecutiveOnes`. Initially, we perform an encrypted XNOR between two sequences $\mathbf{y}_i$ and query $\mathbf{q}$. Here, XNOR operation (NOT XOR) sets a value of 1 to the positions where the sequences are matching. Now, from this XNOR result, we can perform the `maxConsecutiveOnes` algorithm and get the highest number of set bits that are grouped together.

Suppose for a query $\mathbf{q} = 01100111$ and some input sequence $\mathbf{y}_i = 10000110$ where $\mathbf{y}_i \in \mathbf{Y}$, then $\mathbf{q}\,\text{XNOR}\,\mathbf{y}_i$ will be 00011110. Now, if we perform `maxConsecutiveOnes`$(\mathbf{q}\,\text{XNOR}\,\mathbf{y}_i, false)$, then the output should provide us with the encrypted result of 3. This result denotes the number of set bits on the encrypted XNOR operation, hence the set maximal distance between $\mathbf{q}$ and $\mathbf{y}_i$.

Algorithm 12: Thresholded Set Maximal Matching

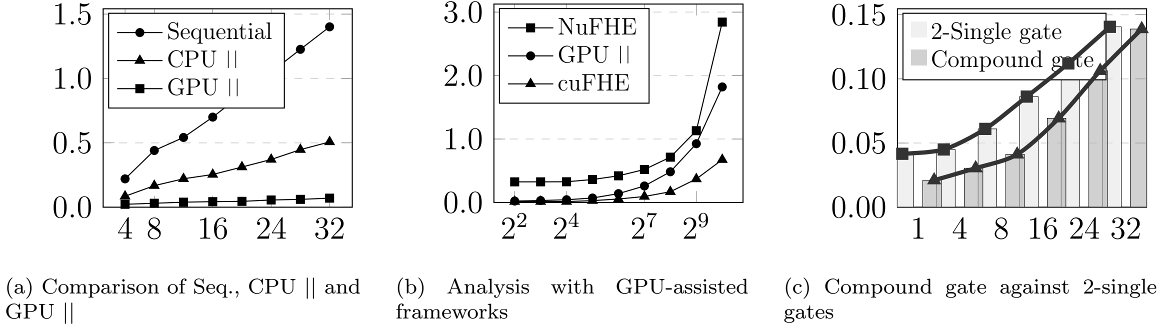
Input: Encrypted query $\mathbf{q}$, encrypted sequence $\mathbf{y}_i \in \{\mathbf{y}_1, \ldots, \mathbf{y}_n\}$ and threshold t

Output: Encrypted SMM distance between $\mathbf{q}$ and $\mathbf{y}_i$ if it is greater than some value t

```
2 Procedure SMMDistance(q, y_i, t)
3     enc_t ← E(t)
4     result ← maxConsecutiveOnes(q XNOR y_i, false)
5     smm_distance ← getMax(result, enc_t)
6     gt_threshold ← AND all bits in(smm_distance XNOR enc_t)
7     smm_distance ←! gt_threshold AND smm_distance
8     return smm_distance
```

Threshold SMM: In a threshold version of this match, we need to output only the distances that is beyond an input threshold t. Here, an extra operation proceeding the `maxConsecutiveOnes` is required where a simple numeric comparison with threshold t would output the result. Therefore, we can use an encrypted MUX operation [93] for this comparison. However, encrypted MUX is an expensive operation and we can replace it with a subtraction. Therefore, we negate the $\mathcal{E}(t)$ value from the resulting `maxConsecutiveOnes`$(\mathbf{q}$ XNOR $\mathbf{y}_i, false)$. Then, we use the `greaterThanZero` algorithm on the result which represents if the SMM distance is beyond the threshold t.

We outline the algorithm in Algorithm 12 where the threshold value is encrypted at first. The matching bits of the query and the sequence is calculated next with the XNOR operation. Subsequently, we perform the comparison operation with a maximum between *result* and $\mathcal{E}(t)$. If $\mathcal{E}(t) \geq result$, then *gt_threshold* is set as the *OR* of all bits among the **smm_distance** XNOR **enc_t** bits. Here, XNOR represents whether two vectors are same or not and doing another logical OR among them. Lastly, we perform AND operation with the distance. If the value of *gt_threshold* is 0, the we get all unset bits on the output whereas set maximal distance in the other case.



(a) Comparison of Seq., CPU || and GPU ||

(b) Analysis with GPU-assisted frameworks

(c) Compound gate against 2-single gates

Figure 5.6: Performance analysis of GPU-accelerated TFHE with the sequential and CPU || frameworks (5.6a), and comparison with the existing GPU-assisted libraries (5.6b). Figure 5.6c presents the performance of compound gates against 2-single gate operations while x-axis and y-axis represents bit size and time in seconds

Table 5.2: Computation time (ms) for Bootstrapping, Key Switching and Misc. for sequential and GPU framework

Bit Size	Sequential					GPU			
n	**Bootstrapping**	**Key Switch**	**Misc.**	**Total**		**Bootstrapping**	**Key Switch**	**Misc.**	**Total**
2	68.89	17.13	27.04	113.05		19.64	2.65	0.45	22.74
4	138.02	34.18	47.97	220.17		18.86	2.69	0.08	21.63
8	275.67	68.31	96.48	440.46		27.83	2.69	0.06	30.58
16	137.25	137.25	425.22	699.72		40.70	2.91	0.44	44.06
32	274.3	274.30	852.51	1401.10		66.74	3.34	0.42	70.50

5.4 Results

The experimental environment included an Intel(R) Core™ i7-2600 CPU having 16 GB system memory with a NVIDIA GeForce GTX 1080 GPU with 8 GB memory [165]. The CPU and GPU contained 8 and 40,960 hardware threads, respectively. We used the same setup to analyze all three frameworks: sequential, CPU ||, GPU ||.

We use two metrics for the comparison: a) execution time and b) speedup = $\frac{T_{seq}}{T_{par}}$. Here, T_{seq} and T_{par} are the time for computing the sequential and the parallel algorithm. In the following sections, we gradually analyze the complicated arithmetic circuits using the best results from the foregoing analysis.

5.4.1 GPU-accelerated TFHE

Initially, we discuss our performance over boolean gate operations, which is deemed as a building blocks of any computation. Figure 5.6a depicts the execution time difference among the sequential, CPU || and GPU || framework for $[4, 32]$-bits. The sequential `AND` operation takes a minimum of 0.22s (4-bit) while the runtime increases to 1.4s for 32-bits.

In the GPU || framework, bit coalescing facilitates storing LWE samples in contiguous memory and takes advantage of available vector operations. Thus, it helps to reduce the execution time from $0.22 - 1.4s$ to $0.02 - 0.06s$ for 4 to 32-bits. Here, for 32-bits, our techniques provide a $20\times$ speedup. Similar improvement is foreseen in the CPU || framework as we divide the number of bits by the available threads. However, the execution time increases for CPU framework since there is only a limited number of available threads. This limited number of threads is one of the primary motivations behind utilizing GPU.

Then, we further scrutinize the execution time by dividing gate operations into three major components—*a)* Bootstrapping, *b)* Key Switching, and *c)* Miscellaneous. We selected the first two as they are the most time-consuming operations and fairly generalizable to other HE schemes. Table 5.2 shows the difference in execution time between the sequential and the GPU || for $\{2, \ldots, 32\}$-bits. We show that the execution time increment is less compared to the sequential approach.

We further investigated the bootstrapping performance in GPU || framework for the boolean gate operations. Our cuda enabled FFT library takes the LWE samples in batches and performs the FFT in parallel. However, due to the h/w limitations, the number of batches to be executed in parallel is limited. It can only operate on a certain number of batches at once and next batches are kept in a queue. Hence, a sequential overhead occurs for a large number of batches that can increase the execution time.

Under the same h/w setting, we benchmark our proposed framework with the existing GPU-based libraries (cuFHE and NuFHE). Although our GPU || framework outperforms NuFHE for different bit sizes (Figure 5.6b), the performance degrades

for larger bit sizes w.r.t. cuFHE. As the cuFHE implementation focuses more on the gate level optimization, we focus on the arithmetic circuit computations. In Section 5.4.3, we analyze our arithmetic circuits where our framework outperforms the existing GPU libraries.

5.4.2 Compound Gate Analysis

According to Section 5.3.1.1, the compound gates are used to improve the execution time for additions or multiplications. Since, the existing frameworks do not provide these optimizations, we benchmark the compound gates with the proposed single gate computations. Figure 5.6c illustrates the performance of one compound gate over 2-single gates computed sequentially. We performed several iterations for different number of bits $(1, \ldots, 32)$ as shown on the X-axis while the Y-axis represents the execution time. Notably,a 32-bit compound gates will have two 32-bit inputs and output two 32-bits.

Here, bit coalescing improves the execution time as it takes only 0.02s for one compound gates evaluation, compared to 0.04s on performing 2-single gates sequentially. However, Figure 5.6c shows an interesting trend in the execution time between 2-single gates and one compound gates evaluation. The gap favoring the compound ones tends to get narrower for higher number of bits. For example, the speedup for 1-bit happens to be $0.04/0.02 = 2$ times whereas it reduces to 1.01 for 32-bits. The reason behind this diminishing performance is the *asynchronous launch queue* of GPUs.

As mentioned in Section 5.3.1.1, we use batch execution for the FFT operations. Hence, the number of parallel batches depends on the asynchronous launch queue size of the underlying GPU which can delay the FFT operations for a large number batches. This ultimately adversely affects the speedup for large LWE sample vectors. Nevertheless, the analysis shows that the 1-bit compound gates is the most efficient, and we employ it in the following arithmetic operations.

Table 5.3: Execution time (sec) for the n-bit addition

Frameworks	**16-bit**	**24-bit**	**32-bit**
Sequential	3.51	5.23	7.04
cuFHE [53]	1.00	1.51	2.03
NuFHE [54]	2.92	3.56	4.16
Cingulata [55]	1.10	1.63	2.16
Our Methods			
CPU ||	3.51	5.23	7.04
GPU_n||	0.94	2.55	4.44
GPU_1||	0.98	1.47	1.99

Table 5.4: Execution time (sec) for vector addition

Length	**16-bit**				**32-bit**		
ℓ	**Seq.**	**CPU ||**	**GPU ||**		**Seq.**	**CPU ||**	**GPU ||**
4	13.98	5.07	1.27		28.05	10.02	2.56
8	27.86	9.96	1.78		56.01	19.29	3.58
16	55.66	19.65	2.82		111.3	38.77	5.70
32	111.32	38.99	5.41		224.31	77.18	11.22

5.4.3 Addition

Table 5.3 presents a comparative analysis of the addition operation for 16, 24, 32-bit encrypted numbers. We consider our proposed frameworks: sequential, CPU ||, and GPU ||, and benchmark them with cuFHE [53], NuFHE [54] and Cingulata [55]. Furthermore, we present the performance of two variants of addition operation: GPU_n|| (number-wise) and GPU_1|| (bitwise) as discussed in Section 5.3.1.2.

Table 5.3 demonstrates that GPU_n|| performs better than the sequential and CPU || circuits. The GPU_n provides a 3.72× speedup for 16-bits whereas 1.58× for 32-bit. However, GPU_n|| performs better only for 16-bit additions compared to GPU_1||. For 24 and 32-bit additions, GPU_1|| performs around 2× better than GPU_n||. This improvement in essential as it reveals the algorithm to choose between GPU_1|| and GPU_n||.

Although, both addition operations (GPU_n|| and GPU_1||) utilize compound gates, they differ in the number of input bits (n and 1 for GPU_n|| and

$GPU_1||$, respectively). Since the compound gates performs better for smaller bits (Section 5.4.2), the bitwise addition performs better than the number-wise addition for 24/32-bit operations. Hence, we utilize bitwise addition for building other circuits.

NuFHE and cuFHE do not provide any arithmetic circuits in their library. Therefore, we implemented such circuits on their library and performed the same experiments. Additionally, we considered Cingulata [55] (a compiler toolchain for TFHE) and compared the execution time. Table 5.3 summarizes all the results, where we found our proposed addition circuit ($GPU_1||$) outperforms the other approaches.

We further experimented on the vector additions adopting the bitwise addition and showed the analysis in Table 5.4. Like addition, the performance improvement on the vector addition is also noticeable. The framework scales by taking similar execution time for smaller vector lengths $\ell \leq 8$. However, the execution time increases for longer vectors as they involve more parallel bit computations, and consequently, increase the batch size of FFT operations. The difference is clearer on 32-bit vector additions with $\ell = 32$ which takes almost twice the time of $\ell = 16$. However, for $\ell \leq 8$, the executions times are almost similar due to the parallel computations. In Section 5.4.2 we have discussed this issue which relies on the FFT batch size. Notably, Figure 5.6c also aligns with this evidence as the larger batch size for FFT on GPUs affects the speedup. For example, $\ell = 32$ will require more FFT batches compared to $\ell = 16$ which requires more time to finish the addition operation. We did not include other frameworks in Table 5.4, since our GPU $||$ performed better comparing to the others in Table 5.3.

5.4.4 Multiplication

The multiplication operation uses a sequential accumulation (reduce by addition) operation. Instead, we use a tree-based vector addition approach (discussed in Section 5.3.1.2) and gain a significant speedup. Table 5.5 portrays the execution times for the multiplication operations using the frameworks. Here, we employed all available threads on the machine. Like the addition circuit performance, here GPU $||$ outperforms the sequential circuits and CPU $||$ operations by a factor of ≈ 11 and

Table 5.5: Multiplication execution time (sec) comparison

Frameworks	**16-bit**	**24-bit**	**32-bit**
Naive			
Sequential	120.64	273.82	489.94
CPU ||	52.77	101.22	174.54
GPU ||	11.16	22.08	33.99
cuFHE [53]	32.75	74.21	132.23
NuFHE [54]	47.72	105.48	186.00
Cingulata [55]	11.50	27.04	50.69
Karatsuba			
CPU ||	54.76	-	177.04
GPU ||	7.6708	-	24.62

Table 5.6: Execution time (min) for vector multiplication

Length	**16-bit**				**32-bit**		
ℓ	**Seq.**	**CPU ||**	**GPU ||**		**Seq.**	**CPU ||**	**GPU ||**
4	8.13	3.25	0.41		32.56	12.15	1.61
8	16.29	6.17	0.75		65.12	23.48	2.96
16	32.62	11.93	1.40		130.31	46.39	5.62
32	65.15	23.58	2.68		260.52	92.44	10.79

≈ 14.5, respectively for 32-bit multiplication.

We further implemented the multiplication circuit on cuFHE and NuFHE. Table 5.5 summarizes the results comparing our proposed framework with cuFHE, NuFHE, and Cingulata. Our GPU || framework is faster in execution time than the other techniques. Notably, the performance improvement is scalable with the increasing number of bits. This is due to tree-based additions following the reduction operations and computing all boolean gate operations by coalescing the bits altogether.

Besides, we also analyze vector multiplications available in our framework and present a comparison among the frameworks in Table 5.6. We found out an increase in execution time for a certain length (*e.g.*, $\ell = 32$ on 16-bit or $\ell = 4$ on 32-bit), which is similar to the issue in vector addition (Section 5.4.3). Hence, the vector operations from $\ell \leq 16$ can be sequentially added to compute arbitrary vector operations. For example, we can use two $\ell = 16$ vector multiplication to compute $\ell = 32$ multiplication

Table 5.7: Execution time (in seconds) for variable size query and target sequence m for different distance metrics

Method	m					
	8	16	32	64	128	256
Hamming Distance	2.89	11.84	47.95	189.81	758.73	3035.0
Set-Maximal	3.76	13.3	51.24	195.72	771.08	3061.48
Set-Maximal (with t)	7.15	20.67	64.43	223.14	827.76	3173.34
Edit Distance	662	2,577	9,989	39,022	154,194	612,435

resulting around 11 mins. In the vector analysis, we did not add the computations over the other frameworks since our framework surpassed their achievements for a single multiplications.

5.4.5 Karatsuba Multiplication

In Table 5.5, we provide execution time for 16 and 24-bit Karatsuba multiplication over encrypted numbers as well. In the CPU || construction of the algorithm, the execution time does not improvement, rather it increases slightly. We observed that for both 16 and 32-bit multiplication, Karatsuba outperforms naive GPU|| multiplication algorithm on GPU by 1.50 times. Karatsuba multiplication can also be considered a complex arithmetic operation as it comprises of both addition, multiplication, and vector operations. However, the CPU || framework did not provide such difference in performance as it took more time for the `fork-and-join` threads required by the divide and conquer algorithm.

5.4.6 String Search Operations

In Table 5.7, we report the execution time for the three string search operations. Here, we report the execution time in seconds where we change the size of the genomic data. The values of $m = \{8, 16, \ldots, 256\}$ denote the number of genes for the query q and target y.

The results show that Hamming distance requires the least amount of time. It is also clear from the definition 5.2.1 as it requires an XOR operation. The set-

maximal matches (definition 5.2.3) needs more operations as `maxConsecutiveOnes` in Algorithm 7 employs the half-adder for all bits. Furthermore, the threshold-version of SMM takes more time since we need to perform the `getMax` operation. For example, for a target and query sequence size of $m = 128$, it takes around 14 and 13 minutes for with and without an existing threshold. However, for lower size of m, we can see that the time difference is more significant as it takes 3.76 seconds to perform SMM, compared to 7.15s.

Edit distance (5.2.2 takes the highest amount of time for the same genome size m. For example, for a sequence of size $m = 32$, edit distance under FHE takes around 2 hours whereas hamming or set maximal matches take less than a minute. Notably, in these methods, we use Algorithm 9 and 8 for $m < 32$ whereas use the alternative (subtraction) method for larger sequences.

5.5 Discussion

In this section, we provide answers to the following questions about our proposed framework:

Is the proposed framework sufficient to implement any computations? In this article, we show how to implement boolean gates properly using GPUs to gain performance improvement. We then show how to compute addition, multiplication, and matrix operations using the proposed framework. Implementing more complex algorithms such as *secure machine learning* [168, 169] are beyond the scope of this work. In future work, we will investigate how to further optimize the framework for machine learning algorithms.

For GPU $||$ framework, how do we compute on encrypted data larger than the fixed GPU memory? The fixed GPU memories and their variations in access speeds are limitations for any GPU $||$ application. Similar problems also occur in deep learning while handling larger datasets. The solution includes batching the data or using multiple GPUs. Our proposed framework can also avail such solutions as it can easily be extended to accommodate larger ciphertexts.

How can we achieve further speedup on both frameworks? On the CPU $||$

Table 5.8: A comparative analysis of existing Homomorphic Encryption schemes for different parameters on 32-bit number.

	Year	**Homomorphism**	**Bootstrapping**	**Parallelism**	**Bit security**	**Size (kb)**	**Add. (ms)**	**Mult. (ms)**
RSA [48]	1978	Partial	×	×	128	0.9	×	5
Paillier [49]	1999	Partial	×	×	128	0.3	4	×
TFHE [32]	2016	Fully	Exact	AVX [50]	110	31.5	7044	4,89,938
HEEAN [51]	2018	Somewhat	Approximate	CPU	157	7,168	11.37	1,215
SEAL (BFV) [52]	2019	Somewhat	×	×	157	8,806	4,237	23,954
cuFHE [53]	2018	Fully	Exact	GPU	110	31.5	2,032	1,32,231
NuFHE [54]	2018	Fully	Exact	GPU	110	31.5	4,162	1,86,011
Cigulata [55]	2018	Fully	Exact	×	110	31.5	2,160	50,690
Our Method	-	Fully	Exact	GPU	110	31.5	1,991	33,930

framework, we have attempted most H/W or S/W level optimizations to the best of our knowledge. However, our GPU || framework partially relied on the global GPU memory, which is slower than its counterparts. This is critical as different device memories offer variant read/write speeds. Notably, shared memory (`L1`) is the fastest memory after register. Our implementation uses a combination of shared and global memory due to the ciphertext size. In the future, we would like to utilize only the shared memory, which is much smaller but should provide better speedup compared to the current approach.

How the bit security level would affect the reported speedup? The current framework is analogous to the existing implementation of TFHE [170] providing 110-bit security which might not be sufficient for some applications. However, our GPU || framework can accommodate any change for the desired bit security level. Nevertheless, such change will change the execution times as well. For example, any less security level than 110-bits will result in faster execution and likewise for a higher bit security. We will include and analyze the speedup for the dynamic bit security levels in future.

5.6 Related Works

5.6.1 Parallel Frameworks for FHE

In this section, we discuss the other HE schemes from Table 5.8 and categorize schemes based on their number representation: *a)* bit-wise, *b)* modular and *c)* approximate.

Bitwise Encryption usually takes the bit representation of any number and encrypts accordingly. The computations are also done bit-wise as each bit can be considered independent from another. This bit-wise representation is crucial for our parallel framework as it offers less dependency between bits which we can operate in parallel. Furthermore, it provides faster bootstrapping and smaller ciphertext size, which can be easily tailored for the fixed memory GPUs. This concept is formalized and named as GSW [42] around 2013, and it was later improved in subsequent works [161, 32, 6].
Modular Encryption schemes utilize a fixed modulus q which denotes the size of the ciphertexts. There have been many developments [171, 172] in this direction as they offer a reasonable execution time (Table 5.8). The addition and multiplication times from FV [46] and SEAL [52] show the difference as they are much faster compared to our GPU-based framework.

However, these schemes do a trade-off between the bootstrapping and the efficiency as they are often designated as somewhat homomorphic encryption. Here, in most cases, the number of computations or the level of multiplications are predefined as there is no procedure for noise reduction. Furthermore, the encrypted data evidently suffers from larger ciphertexts as the value of q is picked from large numbers.

For example, we selected the ciphertext modulus of 250 and 881 bits for FV-NFLlib [46] and SEAL [52], respectively. The polynomial degrees (d) were chosen 13 and 15 for the two frameworks as it was required to comply with the targeted bit security to populate Table 5.8. It is noteworthy that smaller q and d will result in faster runtime and smaller ciphertexts, but they will limit the number of computations as well. Therefore, this modular representation requires to fix the number of homomorphic operations limiting the use cases.
Approximate Number representations are recently proposed by Cheon *et al.* (CKKS [39]) in 2017. These schemes also provide efficient Single Instruction Multiple Data (SIMD) [173] operations similar to the modular representations as mentioned above. However, they have an inexact but efficient bootstrapping mechanism which can be applied in less precision-demanding applications. The cryptosystem also incurs larger ciphertexts (7MB) similar to the modular approach as we tested it for $q = 1050$ and $d = 15$. Here, we did not discuss HELib [174], the first

cornerstone of all HE implementations since its cryptosystem BGV [172] is enhanced and utilized by the other modular HE schemes (such as SEAL [52]).

The goal of this work is to parallelize an FHE scheme. Most HE schemes that follow modular encryption are either somewhat or adopt inexact bootstrapping. Besides, their expansion after encryption requires more memory. Hence, we choose the bitwise and bootstrappable encryption scheme: TFHE.

Hardware Solutions are less studied and employed to increase the efficiency of FHE computations. Since the formulation of FHE [38] with ideal lattices, most of the efficiency improvements are considered from the standpoint of asymptotic runtimes. A few approaches considered the incorporation of existing multiprocessors (*e.g.*, GPU) or FPGAs [175] to achieve faster homomorphic operation. Dai and Sunar ported another scheme LTV [176] to GPU-based implementation [47, 177]. LTV is a variant of HE that performs a limited number of operations on a given ciphertext.

Lei *et al.* ported FHEW-V2 [161] to GPU [178] and extended the boolean implementation to 30-bit addition and 6-bit multiplication with a speed up ≈ 2.5. Since TFHE extends FHEW and performs better than it predecessor, we consider TFHE as our baseline framework.

In 2015, a GPU based HE scheme CuHE [47] was proposed. However, it was not fully homomorphic as it did not have bootstrapping, hence we do not include it in our analyses. Later in 2018, two GPU FHE libraries cuFHE [53] and NuFHE [54] were released. Both the libraries focused on optimizing of the boolean gate operations. Recently, Yang *et al.* [179] benchmarked cuFHE and its predecessor TFHE, and analyzed the speedup which we also discuss in this chapter (Table 5.8).

Our experimental analysis shows that only performing the boolean gates in parallel is not sufficient to reduce the execution time of higher level circuit (i.e., multiplication). Hence, besides employing GPU for homomorphic gate operations, we focus on arithmetic circuit. For example, we are 3.9 times faster than cuFHE in 32-bit multiplications.

Recently, Zhou *et al.* improved TFHE by reducing and performing the serial operations of bootstrapping in parallel [180]. However, they did use any hardware acceleration to the existing FHE operations. We consider this work as an essential

future direction that can be integrated to our framework for better executing times.

5.6.2 Secure String Distances in Genomic Data

One of the earlier attempt with a secure multi-party setting, Jha *et al.* [181] proposed a privacy preserving genomic sequence similarity in 2008. Their paper showed three different methods to mirror the Levestein distance algorithm using a garbled circuit. However, for a sequence of 25 nucleotides, it took around 40 seconds to compute the distance metric between two strings. In 2015, Wang *et al.* [182] proposed an approximation of the original edit distance in a more realistic setting where the authors utilized a reference genomic sequence to compute the edit distance. However, we have analyzed its accuracy on one of our earlier works [155] and showed the accuracy drops for longer input sequences.

In a recent attempt, Shimzu *et al.* [183] proposed a Burrows-Wheeler transformation for finding target queries on a genomic dataset. The authors attempted the set-maximal matches using oblivious transfer on a 2-party privacy setting. However, we employ completely different cryptographic technique as we do not require the researcher to stay active upon providing their encrypted queries. Therefore, the whole computation can be offloaded to a cloud server and harness its full computational capacity. One of the first attempts with FHE to compute edit distance was conducted by Cheon *et al.* [39]. Given with the advances in 2017, their cryptographic scheme was impressive though taking 16.4 seconds to compute a 8×8 block of string inputs. However, the underlying techniques have improved allowing larger string comparison using FHE techniques as we show in this work.

5.7 Conclusion

In this chapter, we constructed the algebraic circuits for FHE, which can be utilized by arbitrary complex operations. Furthermore, we explored the CPU-level parallelism for improving the execution time of the underlying FHE computations. Our notable contribution is the proposed GPU-level parallel framework that utilizes

novel optimizations such as bit coalescing, compound gate, and tree-based vector accumulation. Experimental results show that the proposed method is 20$\times$ and 14.5$\times$ faster than the existing technique for computing boolean gates and multiplications respectively (Table 5.1).

Part II

Privacy Preserving Outputs from Genomic Data Analysis

Chapter 6

Online Algorithm for Differentially Private Genome-wide Association Studies

In this part, we describe the privacy aspects of genomic data analysis, specially regarding sensitive outputs. Here, the proposed solutions rely on the Differential Privacy (Section 2.2.3) protection. We utilize differentially private mechanisms due to its ability to provide theoretical guarantee over the published data or the results and bounding the privacy loss in the process. The proposed methods are also generalized and can be applied to other sensitive data types.

6.1 Introduction

A surge of technical contributions facilitated the storage and computation of a large volume of electronic healthcare records for scientific research. These healthcare data are accumulated every day in different formats (and of different modalities) from hospitals, clinics, and research organizations located in multiple geographical locations. From Internet of Things (IoT) devices tracking our body weights to the Intensive Care Unit (ICU) monitoring the critical conditions of a patient, all these interconnected components are constantly generating a large volume of data. Such

rich healthcare datasets are stored and often disseminated as datasets for future research on a specific disease (*i.e.*, cancer) or to improve health care systems in general [184, 185].

Among different healthcare data types, the genomics data is unique as it represents our hereditary information and susceptibility to certain diseases. Regardless of the scientific significance of human genomics data, they are intrinsically private and should not be shared without any protective mechanism [26, 27, 186]. Therefore, this sensitive information requires rigorous sensitization or privacy mechanism before any public dissemination [187]. Currently, access to these datasets demands consent forms, certifications, and lengthy administrative processes which impede timely scientific discoveries [186] as they are often protected by the law (*i.e.*, PIPEDA [188], HIPAA [189]).

Therefore, there have been several attempts to attain the privacy of the human genomics data acceding some cryptographic approaches. One of the pragmatic cryptographic mechanism is *Differential Privacy* (DP) [74, 30] as it provides rigorous privacy guarantee [75] through the theoretical and quantifiable privacy bounds on the disclosure of arbitrary data/query results. Furthermore, it provides a certain accuracy threshold based on the privacy mechanism. DP algorithms, however, have not seen its due applicability partly because of the complexity and intricate parameterization. In this work, we manifest the difference between different settings of DP algorithms, emphasizing on the privacy-utility tradeoff.

The DP algorithms often fall short on their accuracy expectations due to the noise added to the process or individual query outputs [190, 20, 191]. The accuracy of the statistical tests is essential to statistical tests on human genomics data or Genome-Wide Association Studies (GWAS). Noisy but private results can be misleading and render the study useless. Therefore, in this work, the accuracy of the proposed methods is analyzed precisely along with acceptable privacy guarantee.

The proposed methods also investigate *online algorithms*, which are employed on sequential inputs that are not known prior to the algorithms. For example, an online sorting algorithm will not get all input numbers simultaneously as they will appear sequentially. In this case, genomic data appear serially, any optimal decision

taken at a specific time might prove to be sub-optimal at a later time considering a different continuation of the input. Similarly, in arbitrary genomics analysis, a DP mechanism might not know the queries from a researcher. This is because the researcher is looking for some Single Nucleotide Polymorphisms (SNPs), which are associated with an unknown disease. Therefore, s/he performs an exploratory search sequentially on a list of SNPs based on the outputs or previous GWAS. As the queried domain is unknown to the DP algorithm, an online mechanism is required [192], which adheres the data privacy providing the maximum accuracy.

6.1.1 Related Works in Differentially Private GWAS

There have been several notable attempts employing DP algorithms in multiple genomics data analysis. The contributions of these works often lie either in altering the underlying functions or addition of differentially private noise into the calculations. Another approach is to use a preprocessing step that allows privacy-preserving outputs. Nevertheless, we propose a generalized DP noise generation mechanism that is simply added to the intermediary statistics of GWAS without any preprocessing.

One of the earlier attempts proposed a differentially private logistic regression, perturbing the objective functions [193]. In this work, we do not change the GWAS functions but dynamically change the inputs to them. Wang *et al.* [94] proposed a more generalized framework to share genomics data with DP guarantees. In this work, the authors partition the data into blocks and construct a tree-like structure. Then, the privacy-preserving methods (anonymization and DP) are applied to it before dissemination. Notably, this work aligns with private data sharing, whereas we target the privacy of the outputs from genomics data computations. In addition, we also do not require any complex preprocessing of the dataset. Uhlerop *et al.* [194] also proposed an approach to *publish* the minor allele frequencies using ϵ privacy guarantee.

Johnson and Shmatikov [195] proposed a private χ^2 test, where the researchers did not know the number of queries they would require prior to the analysis (similar to our online query mechanism). Their method handled GWAS datasets containing the

disease association or case-control studies as the researcher performed an exploratory ab initio search and utilized distance-score mechanism. Similarly, Sei and Ohsuga [196] proposed two methods for the χ^2 test. Their *RandCHI* method applied a Laplace noise based on the global sensitivity to the final χ^2 value. The other method *RandCHIDist* utilized the noisy χ^2 values to correctly predict the statistical significance and reject the null hypothesis. In comparison to both of these works, our mechanism only operates on the in-between outputs from the count queries and can be utilized towards arbitrary GWAS function.

In 2015, Tramèr *et al.* [197] considered a weaker adversarial model proposing a DP mechanism for membership privacy. The authors showed that for a constant pre-fixed privacy budget and adversaries with bounded priors, their solution would incur lower noise into the data. We do not make any assumptions on adversarial capacities or fix the privacy budgets for GWAS. In 2016, Simmons and Berger [198] defined the search for significant SNPs on DNA by modeling it as an optimization problem and relaxed the problem while solving it in constant time. It is noteworthy that the previous work utilized the χ^2 values to determine the order of the SNPs succeeding in the earlier attempts [194, 195, 199].

On a different work, Simmons *et al.* [7] proposed differential techniques for two new GWAS: EigenStrat and Linear Mixed Model (LMM), extending the earlier approach from [198]. Both these methods are utilized for population stratification, which shows the difference in allele frequencies in a subset of a population compared to the entire population. The proposed method relies on private χ^2 statistics, which are then utilized for EigenStrat statistics. Their proposed method outputs the top-k highly correlated SNPs for a given disease based on the EigenStrat and LMM statistics. In this work, we are only interested in benchmarking the accuracy of the corresponding top-k SNP list as they are generated from our proposed method and [198].

In 2017, Wang *et al.* [8] proposed DP mechanism to privately perform a family-based study: Transmission Disequilibrium Test (TDT). Here, the authors employed the shortest Hamming distance score and exponential mechanism to ensure a quantifiable privacy guarantee over the TDT statistics. However, any distance-based

method will require additional pre-processing, which might prove to be expensive for large-scale genomics data, as mentioned in [8]. Though [8] considers the privacy of family linkages (whereas we consider individual), we compare our method with this work in terms of accuracy of the TDT results.

The federated or de-centralized model for GWAS operations is also targeted in SAFETY [200], where the authors showed a hardware-oriented approach to securely aggregate the results from multiple data owners. SAFETY bridges recent cryptographic technique, homomorphic encryption (HE), which allows computations under encryption [46] and proprietary Intel Software Guard Extensions (SGX) to securely aggregate data in untrusted cloud servers. The authors demonstrated the efficacy of using HE in SGX and later utilized for four different types of GWASs. In this work, we incorporate these GWASs along with three new studies. However, this work is fundamentally different than [200] as SAFETY operates on encrypted inputs in a de-centralized setting and does not offer any privacy guarantee. Meanwhile, we propose both centralized and de-centralized models that consider the privacy of the data and outputs with a novel DP mechanism. In other words, we focus on the privacy of the participants in this work, whereas, in SAFETY, the *data, and computational security* on a cloud environment were the main concern.

In this work, we will not discuss theoretical differential privacy approaches, which are related in the online setting [192, 201]. Furthermore, we avoid the details of DP mechanisms applied to the traditional database SQL queries [78, 202]. Nevertheless, some other attempts are utilizing DP mechanisms combined with different cryptographic approaches in the context of genomics data [203, 204]. We will only discuss the related works that operate in the privacy-preserving GWASs area using DP algorithms solely. We propose a generalizable DP mechanism for genomics data executing arbitrary GWAS by managing the privacy budgets and answering sequential online queries without any prior assumptions.

Table 6.1: A summary of the accuracy results for seven GWAS with different privacy budgets (details on Section 6.4)

GWAS	Global	Local	ϵ
LD	99.7	99.6	0.7
HWE	88.9	81.1	3.3
CATT	94.7	89.7	2.76
FET	91.4	84	6.4
Benchmark	Ours	[7, 8]	
EigenStrat	90	80	7
LMM	80	70	7
TDT	69	25	7

6.1.2 Contributions

In this chapter, we propose a method that suits an online GWAS setting that targets an online GWAS setting, where the targeted SNPs (or query parameters) are not known or fixed from the researchers prior to a study. To illustrate the efficacy and benchmark of our proposed model, we selected seven different queries based on the following Genome-Wide Association Studies (GWAS): 1. Linkage Disequilibrium (LD), 2. Hardy-Weinberg Equilibrium (HWE), 3. Cochran-Armitage Test for Trend ($CATT$), 4. Fisher's Exact Test (FET), 5. EigenStrat, 6. Linear Mixed Models (LMM), and 7. Transmission Disequilibrium Test (TDT). We demonstrate the accuracy (mismatch with non-private/original results) of these studies using an ensemble of DP algorithms with different privacy budgets. Hence, the contributions of the work can be summarized as follows:

- We propose an online algorithm that dynamically manages the privacy budget ϵ of a differential privacy mechanism. We demonstrate the efficacy of such mechanisms in both centralized and decentralized trust settings using the global and local differential mechanisms, respectively.
- Our method employs harmonic series to classify the privacy budgets to uneven groups offering higher accuracy results for the seven online GWAS functions in both global and local privacy models.

- Furthermore, we incorporate the *Bin Packing* to optimize the privacy loss allowing sequential and parallel compositions for a set of queries on a partitioned genomics dataset.

- Finally, we implement the proposed algorithms and exhibit the accuracy (summary in Table 6.1) for the aforementioned seven GWAS tests. The proposed algorithms achieve over 80% accuracy in the global setting, while the performance remains competitive in the distributed local model.

6.2 Preliminaries

In this section, we describe our problem settings and the corresponding privacy or trust models, bin packing and some other required backgrounds. In Table 6.3, we outline the related mathematical notations used throughout the chapter.

6.2.1 Problem Description

In GWAS, a data analyst intends to perform statistical queries on a database $\mathcal{DB}$ containing genomics data from many different individuals (Figure 6.1). Ideally, these data owners are geographically distributed and collect their participants' genomics data pertaining to different populations. Therefore, considering each data owner has their own $\mathcal{DB}_i$, the data analysts desire to retrieve results for their GWAS on the whole $\mathcal{DB} = \mathcal{DB}_1 + \ldots + \mathcal{DB}_n$ (for n data owners).

Formally, we define a genomics $\mathcal{DB}$ containing m individuals represented by $r_1, \ldots, r_m$ records, where each record has d positions (or Single Nucleotide Polymorphisms) $(a_1, \ldots, a_d)$, where $r_i \in \mathbb{R}^d$. Furthermore, the records r_i also have demographic information and disease association (phenotype), which are required in GWAS studies. We show an example of the dataset in Table 6.2, where the analyst will execute his/her queries on $\mathcal{DB}$ defining the demographics, phenotype information and the nucleotide value in position a_i. Notably, the proposed work does not target a privacy-preserving mechanism to publish $\mathcal{DB}$ but achieves privacy on the query outputs.

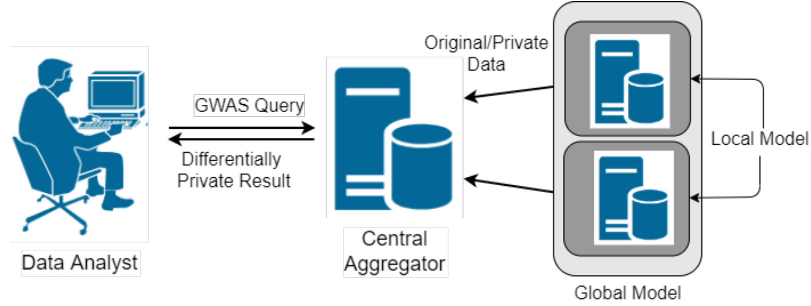


Figure 6.1: Privacy preserving Genome-wide Association Studies (GWAS) models where data owners share the data (or results) with a central aggregator

Table 6.2: Sample Data ($\mathcal{DB}$) representation for GWAS

	Sequence				**Demographics**			**Phenotype**
r_i	a_1	a_2	a_3		**Age**	**Sex**	**Ethnicity**	**Cancer**
1	CC	AG	AA	...	54	M	Caucasian	Positive
2	CA	-	TA	...	50	F	Asian	Negative
3	AC	AA	AT	...	35	M	Hispanic	Positive
4	-	CT	CC	...	33	F	Asian	Positive
			⋮	⋮	⋮	⋮		
m	AA	GA	-	...	38	F	Asian	Negative

Here, we consider the queries from the analyst to be completely unknown to the data owners. For example, a query, q can be $q = \{a_1^q, \ldots, a_d^q\}$ which can arbitrarily cover any SNP or phenotype based on the study. Since the demographic features often carry some importance in genomic studies [205], they can also come as additional parameters on q. This form of input queries to $\mathcal{DB}$ can be perceived as 'online' since the proposed differetially private algorithm handles them serially without any clue about the subsequent queries. Therefore, the proposed privacy-preserving mechanism is termed as an online algorithm.

6.2.2 Privacy Models

In this work, our primary privacy goal is to produce a privacy-preserving output to GWAS results. In other words, rather than publishing the whole dataset $\mathcal{DB}$, we are targeting a privacy-preserving GWAS mechanism that outputs differentially private results. Since this mechanism will add noise to the intermediate calculations, the

accuracy or correctness of the private GWAS needs to be at compared against the non-private or plaintext GWAS as well.

While considering the privacy of the genomic data in $\mathcal{DB}$, two different privacy models: a) Global and b) Local. In the *global model*, all data owners send their genomics data to the Central Aggregator (CA) *without any privacy* guarantee over the data. CA handles the different incoming queries from the analysts while the data owners can become unavailable after sharing. The aggregator is trusted unequivocally by all parties (data owners and analysts) as it is responsible for protecting the privacy of the underlying genomics data.

On the contrary, in the local model, the data owners do not trust a central authority over its data and guarantee privacy prior to sharing any in-between results. Therefore, instead of the aggregator, the data owners employ a differential privacy mechanism for each query and need to communicate with CA. For example, each query from the analyst is propagated by CA to the individual data owners. Then, the results for these queries are sent to CA with a privacy guarantee, which is aggregated and returned to the analyst.

Notably, in both models, we expect the data owners and CA to be honest-but-curious but *not malicious* against the protocols [206]. Since CA can be a third-party server (*e.g.*, cloud), it will adhere to the outlined protocols but may try to learn all possible information from incoming messages or intermediate logs from the program. However, we did not consider any collusion attacks as we detail this in Limitation (Section 6.5.3).

6.2.3 Genomic Data and GWAS

Our genomic data consist of nucleotides (A, T, C, or G) that are located at different positions on a chromosome. The different variants on a specific position or locus are called alleles. These genetic variations in DNA sequences have significant influences on diseases and phenotypes. The most common form of genetic variants is called Single Nucleotide Polymorphisms (SNPs), which refer to an alteration of a single nucleotide on a DNA block, as denoted in Figure 6.2. Here, for a particular DNA,

Table 6.3: Notations used in the proposed method

Notation	Description
$\mathcal{A}_i$	differentially private algorithm
ϵ_i	privacy budget of $\mathcal{A}$
p_i	selection probability of $\mathcal{A}$
k	number of queries from the analyst
n	number of DP algorithms considered
m	number of individuals in $\mathcal{DB}$ or
b_i	bins for online bin packing with size $(0, m]$
d	number of SNPs per individual in $\mathcal{DB}_i$

we see the major allele `C` is replaced by `A` in one or both strands. Notably, this permutation is normal and happens in every thousand nucleotides on average.

Figure 6.2: Single Nucleotide Polymorphisms (SNPs) in DNA, where `C` and `A` are major and minor allele respectively

However, they often prove to be interesting as some of these SNPs are unique to a population or result in a physical trait or expression. Furthermore, they are employed to find genetic markers and identify a gene's functions or association with a disease. Therefore, an analyst is often interested in whether a target SNP (*i.e.*, SNP1) is correlated with a specific disease.

Genome-Wide Association Studies (*GWAS*) are statistical tests on genomic data that answer the questions mentioned above, like susceptibility towards a particular disease or physical traits by analyzing these genetic variations. For example, GWAS on breast cancer will require a dataset with the case, control individuals (with/out

disease), and the targeted SNPs, which are suspected to be associated with the disease. Here, the number of subjects or individuals in the dataset is crucial as it leads to a more reliable and robust result from the underlying statistical tests. Therefore, researchers tend to collect or utilize the largest available dataset on specific SNPs for their desired analysis.

In this chapter, we target the aforementioned statistical tests in Section 6.1.2 to analyze the proposed algorithms. The details and significance of these statistical tests in GWAS can be found in [207] as we do not discuss the details of these GWAS functions in this work. However, it is noteworthy that all these statistical functions can be decomposed into arbitrary database queries, where each query can be answered with a DP guarantee. Therefore, the final result from the GWAS tests will be private, and its accuracy will demonstrate the efficacy of the proposed algorithms.

6.2.4 Online Algorithms and Bin Packing

Online Algorithms deal with serial inputs that are unknown to the algorithm prior to execution. In our problem setting (Section 6.2), the GWAS queries from the analysts are similarly not known beforehand. Therefore, the central aggregator needs to handle these queries in an online manner ensuring the privacy of the genomic data. Hence, the privacy of the DP algorithm needs to be adjusted according to the serially incoming queries as the analyst wants to retrieve arbitrary statistics from $\mathcal{DB}$ for his/her exploratory ab initio study. Apart from the privacy preliminaries, we utilized another algorithmic concept named **Bin Packing**:

Definition 6.2.1. (Bin Packing) In a bin packing problem with n items $a_1, \ldots, a_n$ where $a_i \in (0, 1]$, the goal is to pack them in the minimum number of bins where each bin is of unit (1) size.

In the online version, the input items are not known initially to the algorithm as they come serially. Fundamentally, the vanilla bin packing is an optimization problem consisting of n items of fixed size $(0, 1]$ and bins with a fixed capacity. It can be formulated as a minimization problem, where we minimize the number of bins and fit

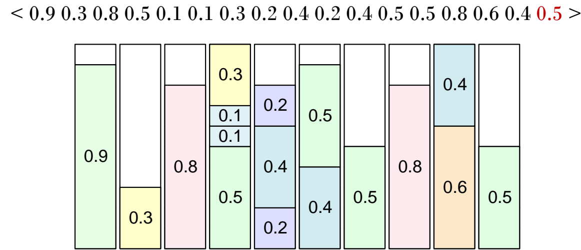


Figure 6.3: Packing 17 items of different size $(0, 1]$ according to next fit algorithm

every item with their variable size. In the online variant of bin packing, the sequence of items is considered to be arriving continuously while the items' sizes are unknown to the algorithm. Hence, the target is to utilize the least number of bins to accommodate these items. Notably, the offline bin packing is an *NP-Hard* problem [208].

In Figure 6.3, we show an online bin packing with items of $\{0.9, 0.3, \ldots, 0.5\}$ sizes where the maximum capacity of a bin is 1 and item sizes are $\in (0, 1]$. The two constraints are: a) the individual item sizes are not known prior to their arrival at time t, and b) re-arrangement of the items are not allowed. Figure 6.3 shows an example of the Next Fit algorithm, where only one bin is kept open at a given time. Therefore, the current item will be stored in the already opened bin only if it had sufficient space. For example, at $t = 2$ we cannot store items with size 0.3 (30% of $\mathcal{DB}$) on the first bin as it already contains items with 0.9 size. Hence, we open a new bin and close the first bin. The leftover space 0.1 in the closed bin is not used in the future.

On the contrary, according to the First Fit algorithm, the items are put into the *first* bin that has sufficient space for it. Nevertheless, we relate the online bin packing to our private GWAS scenario where queries (or functions) f appear online. As with each result (even with noise), we disclose some information about our $\mathcal{DB}$, which is represented by the size of that particular item. Here, items are the equivalent of online queries. Therefore, we can quantify the privacy loss for each GWAS query based on the number of opened bins.

6.3 Methods

In this section, we discuss our differentially private (DP) mechanisms for different GWAS settings. As our proposed methods are generalizable for GWAS studies requiring statistical queries, we divide the proposed methods into three parts: a) the generalized GWAS mechanism, b) Global, and c) Local Model using differential privacy. Here, Section 6.3.1 discusses the GWAS operation along with the dataset partition strategies while the intricacies of generating the differentially private noise is detailed in Section 6.3.2 and Section 6.3.3.

6.3.1 Differentially Private GWAS

6.3.1.1 Dataset Partition

The genomic datasets contain some demographic features (*i.e.*, age, sex), which can be utilized in the genomics association study. Therefore, prior to the privacy-preserving mechanism, we partition the dataset based on the available different demographic attributes on the dataset. For example, individuals aged between 50 to 60 can be in one group. Here, this grouping (age 50-60) is done according to the histogram or frequency of the individuals available in the dataset. These partitions are also extended on composite attributes like age 50-60, male and Hispanic.

The dataset partitioning is a preprocessing step that is executed before any GWAS queries are considered. As the researchers specify their inclusion criteria for their GWAS study and only if their criteria match the dataset partitions, it is considered for the analysis. Notably, the query criteria can lie between multiple groups, in which we consider all such partitions.

Also, this is not a mandatory step as queries can blanket the whole $\mathcal{DB}$. However, such partitions will only allow lower privacy costs which we discuss later in Theorem 6.3.3 (Section 6.3.4.2). Therefore, the privacy analysis of the proposed algorithms should consider the whole dataset for the worst-case scenario, which is the absolute maximum the researcher may query for.

Table 6.4: Contingency table for SNP1 with `C` and `A` as the major and minor allele, respectively

Value, i	Case	Control	Total
0 (CC)	10	12	22
1 (CA/AC)	12	11	23
2 (AA)	2	1	3
Total, S	24	24	48

6.3.1.2 GWAS Operation

GWAS functions depend mostly on the statistics retrieved from the genomics datasets that contain different SNPs. During this process, a summary or contingency table is needed that outlines the relationship of the targeted SNP and case/control variables. For example, the GWAS contingency table will contain the different count statistics of one (or multiple) targeted SNP(s) based on the query parameter(s) and may contain a disease association (*i.e.*, CATT, FET).

Table 6.4 denotes an example contingency table for `SNP1` where case and controls implies the presence or absence of the disease. It is a summarized version of the dataset (Table 6.2) which replaces the nucleotides with their counts. For example, if `C` and `A` are the major and minor allele for `SNP1` respectively, then there are three possible permutations (*i.e.*, CC, CA/AC, AA) available. These combinations are represented as 0, 1 and 2 in this table.

Following a query from the analyst, this contingency table is generated by the CA from the underlying dataset according to the query parameters (SNP, disease etc.). However, CA has no knowledge of these parameters or the list of the targeted SNPs. The server will only get queries, $q = \{a_1^q, \ldots, a_d^q\}$, sequentially and the corresponding GWAS function. For example, if a_1^q =`SNP1`, then CA will build Table 6.4 and later add differentially private noise as discussed later in Section 6.3.2.

We outline an example of a contingency table for GWAS function: Cochran-Armitage Test for Trend (CATT) which identifies whether a certain SNP is related with a disease or not. To get the CATT statistics on `SNP1` (from Table 6.2), the frequency of the three possible alleles CC, CA/AC, and AA (or 0, 1 or 2, respectively)

are required. CA needs to calculate m_0, m_1, m_2 upon the arrival of the query reporting the counts of the individuals who has these alleles on their `SNP1`. Furthermore, CA also gets $m_0^{case}, m_0^{control}$ which represents the case-control instances. For example, in Table 6.4, $m_0^{case} = 10, m_1^{control} = 11$ whereas the sums are represented as $S_{case} = 24, S_{control} = 24$. The equation for CATT is:

$$T = \sum_{i=1}^{3} w_i (m_i^{control} S_{case} - m_i^{case} S_{control}),$$
$$\chi^2 = \frac{T^2}{Var(T)}$$

where w_i are weights which are set as $i-1$ for linear trends (co-dominant model). Finally, CA utilizes the χ^2 value to determine null or alternative hypothesis based on a p-value test and sends the binary result to the analyst. The corresponding statistics for the p-value test can also be sent to the analyst as they can compute the result themselves or re-use these values for another GWAS query. Nevertheless, the values to construct the contingency table is queried in an online manner which are unknown to the CA beforehand.

We add differentially private noise to these count query values of alleles as the query arrives (except randomized response in Section 6.3.3.2). Therefore, the analyst receives a differentially private output from CA on each query. Since, CA as no clue on the query parameters or their sequence, it is termed as 'online' as we propose generalized algorithms to output privacy-preserving results.

In the global model, CA knows the original statistics (w/o noise) from the data owners. On the contrary, the local model avoids this issue as the data owners ensure a privacy guarantee and send noisy values to the aggregator. Here, each query is forwarded to the data owners as they send their individual (noisy) contributions for the contingency table. CA merely aggregates the values from the data owners, and the further privacy-preserving mechanism is not considered.

6.3.2 Global Model $\mathcal{A}$

In this section, the proposed generalized differentially private mechanism is described which is later extended in the local (decentralized) model in Section 6.3.3.

6.3.2.1 Initialization

The initialization step is required prior to the query execution where an ensemble of DP algorithms is created, which collectively defines the original algorithm $\mathcal{A}_{BP} \subseteq \{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n\}$. Hence, it results in n different DP algorithms $\mathcal{A}$, where each $\mathcal{A}_{BP}(i)$ will differ in terms of their privacy budgets according to *harmonic series*. Details on the privacy budget, ϵ is available in Section 2.2.3.3.

Harmonic distribution of budget ϵ: We utilize harmonic series to determine the different privacy budgets for our $\mathcal{A}_{BP}(i)$'s. Harmonic is a divergent series, which denotes that the partial sum of the series (of k terms) offers a finite limit [209]. The series is defined as follows,

$$\sum_{k=1}^{\infty} = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \ldots$$

Furthermore, the partial summation of any fixed sequential k numbers from this series will result in a harmonic number while the infinite sum will result in ∞ (divergent series).

The central server has a maximum and minimum privacy budget defined by $\epsilon_{min}, \epsilon_{max} > 0$. We can define n different classes between this range $[\epsilon_{max}, \epsilon_{min})$ such that each class, $i \in [1, n]$ will be $(\frac{\epsilon_{max}}{i+1}, \frac{\epsilon_{max}}{i}]$ given $\frac{\epsilon_{max}}{i+1} \geq \epsilon_{min}$. Here, each class will have their own domain of privacy budgets and is disjoint with the preceding one.

Lemma 6.3.1 (Number of classes). For $\epsilon_{min}, \epsilon_{max} > 0$ and $\epsilon_{max} \geq \epsilon_{min}$, there will be $n \geq \frac{\epsilon_{max} - \epsilon_{min}}{\epsilon_{min}}$ number of classes available for $\mathcal{A}_{BP}$.

Proof. In each class, the privacy budget is being reduced at most $(\frac{1}{i} - \frac{1}{i+1})$ amount.

Thus, at n^{th} class $(i = n)$, the budget will be $\frac{\epsilon_{max}}{n+1}$ which is at most ϵ_{min}. Thus,

$$\frac{\epsilon_{max}}{n+1} \leq \epsilon_{min}$$
$$\Rightarrow n \geq \frac{\epsilon_{max}}{\epsilon_{min}} - 1 = \frac{\epsilon_{max} - \epsilon_{min}}{\epsilon_{min}}.$$

□

Notably, we can replace the harmonic series values with another mathematical series (*i.e.*, geometric series) while defining the privacy budgets. We selected harmonic series as it allows an uneven distribution of the ϵ's. This adds different privacy classes in our method which differ in the level of privacy and ultimately utility which we discuss next.

Suppose we have $\epsilon_{min} = 0.2 = \frac{1}{5}$ and $\epsilon_{max} = 1$, hence $n \leq \frac{1-0.2}{0.2} = 4$. Therefore, we will have 4 classes of privacy budgets to pick where the maximum and minimum is defined as, $\{(\frac{1}{2}, \frac{1}{1}], (\frac{1}{3}, \frac{1}{2}], (\frac{1}{4}, \frac{1}{3}], (\frac{1}{5}, \frac{1}{4}]\}$. Here, the first class will have a ϵ_{min} and ϵ_{max} as $\frac{1}{2}$ and 1, respectively.

Ensemble of $\mathcal{A}_{BP}(i)$: We use these categories of global privacy budget from harmonic series to initiate different classes of DP algorithms $\mathcal{A}_{BP}(i)$. Here, $\mathcal{A}_{BP}(1), \mathcal{A}_{BP}(2), \ldots, \mathcal{A}_{BP}(n)$ DP algorithms are defined that operate on $\epsilon_1, \ldots, \epsilon_n$ budgets where ϵ_i is initially chosen between $[\frac{\epsilon_{max}}{i}, max(\frac{\epsilon_{max}}{i+1}, \epsilon_{min}))$, at random. There will be at most $\frac{\epsilon_{max} - \epsilon_{min}}{\epsilon_{min}}$ algorithms according to Lemma 6.3.1.

The necessity of $\mathcal{A}_{BP}(i)$ ensemble can be viewed as the block-stacking problem [210] where items with the same dimensions are stacked on top of each other. Similarly, in our scheme, we support the less noisy (and more accurate) DP algorithms with multiple noise-prone algorithms. Therefore, the harmonic series can be replaced with any uneven divergent series as long as it adheres to Lemma 6.3.1. Next, we introduce the selection criteria of these algorithms upon answering each query.

Selection Probability, p_i: Each DP algorithm $\mathcal{A}_{BP}(i)$ will have an associated *selection probability,* p_i where $\sum_i^n p_i = 1$. This probability will determine if any

particular algorithm is more likely to answer an incoming query at a given time, t. Initially, this p_i is set uniformly *s. t.* every algorithm has an equal chance to answer the currently imposed query. Let the central aggregator has set $\epsilon_{max} = 1$ and $\epsilon_{min} = 0.2$. Therefore, we will have 4 classes of DP algorithms where ϵ can be picked randomly from,

$$\epsilon \in_{i=1}^{4} \left\{ \left[\frac{\epsilon_{max}}{i}, max(\epsilon_{min}, \frac{\epsilon_{max}}{i+1}) \right) \right\}$$
$$= \left\{ \left[\frac{1}{1}, \frac{1}{2} \right), \left[\frac{1}{2}, \frac{1}{3} \right), \left[\frac{1}{3}, \frac{1}{4} \right), \left[\frac{1}{4}, \frac{1}{5} \right) \right\}$$

As $n = 4$, there will be four DP algorithms where each $\mathcal{A}_{BP}(i)$ takes a random $\epsilon(i)$ value from the range $[\epsilon_{min}, max(\epsilon_{min}, \epsilon_{max}/(i+1))]$. Now, all the selection probability will be set uniformly as $p_i = 1/n = 1/4$. This denotes that each algorithm $\mathcal{A}_{BP}(i)$ is equally likely at $t = 0$. This probability changes according to the generated noise which we describe next.

6.3.2.2 Query Execution

After initializing all $\mathcal{A}_{BP}(i)$ with $\{\epsilon(i), p_i\}$, the aggregator can answer the online queries as they are represented as f_t where $t \in [0, k]$ while k is the maximum number of queries allowed ($t = 0$ denotes the initialization according to Section 6.3.2.1).

Noise Addition with $\epsilon(i)$: For the privacy guarantee, the proposed method rely on the Laplace mechanism as mentioned in Section 2.2.3.3. For an arbitrary query $f_t(\mathcal{DB})$, we pick an $\mathcal{A}_{BP}(i) = \{\epsilon(i), p_i\}$ based on the selection probability, p_i. Now, for an Algorithm $\mathcal{A}_{BP}(i)$, Equation 2.1 will be:

$$f_t(\mathcal{DB})' = f_t(\mathcal{DB}) + Lap(\Delta f_t/\epsilon(i)), \tag{6.1}$$

where Δf is the sensitivity of the query and $\epsilon(i)$ is the predefined privacy budget for $\mathcal{A}_{BP}(i)$. According to the Laplace mechanism,

Theorem 6.3.1 (Privacy bound for $\mathcal{A}_{BP}(i)$). Algorithms $\mathcal{A}_{BP}(i) \in \{\mathcal{A}_{BP}(1), \mathcal{A}_{BP}(2), \ldots, \mathcal{A}_{BP}(n)\}$ individually holds $\epsilon(i)$ differential privacy.

The proof is available below and can be extended for n instances of $\mathcal{A}_{BP}(i)$ with the sequential composition property [77] (more detail in Section 6.3.4 and Lemma 6.3.4.2). This property defines that the n DP algorithms in conjunction also holds $\sum_{i=1}^{n} \epsilon(i)$ differential privacy.

Proof. Let $\mathcal{DB}$ and $\mathcal{DB}'$ are two database only differing in one row, $||\mathcal{DB} - \mathcal{DB}'|| \leq 1$ and f_t is a function such that $f_t(\mathcal{DB}) \rightarrow \mathbb{R}^k$. If $p_{\mathcal{DB}}(x)$, $p'_{\mathcal{DB}}(x)$ are the density functions of one algorithm $\mathcal{A}_{BP}(i)$ with budget $\epsilon(i)$, then for arbitrary points $z \in \mathbb{R}^k$ we have:

$$
\begin{aligned}
\frac{p_{\mathcal{DB}}(x)}{p'_{\mathcal{DB}}(x)} &= \prod_{t=1}^{k} \left(\frac{\exp(-\frac{|f_t(\mathcal{DB}) - x_t|}{\Delta f / \epsilon(i)})}{\exp(-\frac{|f_t(\mathcal{DB}') - x_t|}{\Delta f / \epsilon(i)})} \right) \\
&\leq \prod_{t=1}^{k} \exp(\frac{\epsilon(i)|f_t(\mathcal{DB}) - f_t(\mathcal{DB}'|}{\Delta f}) \\
&= \exp(\frac{\epsilon(i)||f_t(\mathcal{DB}) - f_t(\mathcal{DB}'||_1}{\Delta f}) \\
&= \exp(\frac{\epsilon(i)}{\Delta f}) \\
&\leq \exp(\epsilon(i)).
\end{aligned}
$$

Here, sensitivity $\Delta f \geq 1$ and $||f_t(\mathcal{DB}) - f_t(\mathcal{DB}'||_1 = 1$ as $||\mathcal{DB} - \mathcal{DB}'|| \leq 1$. □

Noise Threshold, T: We define the added noise at time t as $e_t = f_t(\mathcal{DB})' - f_t(\mathcal{DB})$. If this error e_t is greater than a predefined threshold T then we update the selection probability of the corresponding DP algorithm $\mathcal{A}_{BP}(i)$. The value of T can be set as $1/\sqrt{m}$, where m is the size of $|\mathcal{DB}|$ [192]. If the added noise is below T, algorithm 13 does not change p_i.

Algorithm 13: Query ($\Delta f = 1$) Execution

Input: n DP algorithms $\mathcal{A}_{BP}(i) = \{\epsilon(i), p_i\}$, Threshold T, query $f_t(\mathcal{DB})$
Output: Differential Private answer $f_t(\mathcal{DB})'$

2 $\mathcal{A}_{BP}(i) \leftarrow$ Select one DP algorithm from $\mathcal{A}_{BP}$ based on selection probability p_i
3 $\epsilon \leftarrow$ pick $\epsilon(i)$ randomly from $\mathcal{A}_{BP}(i)$
4 $f_t(\mathcal{DB})' \leftarrow f_t(\mathcal{DB}) + Lap(1/\epsilon(i))$
5 **if** $|f_t(\mathcal{DB})' - f_t(\mathcal{DB})| > T$ **then**
6     $p_i^{new} \leftarrow p_i^{old} \dfrac{\epsilon_{min}(i)}{\epsilon_{max}(i)}$
7     $update \leftarrow p_i^{new} - p_i^{old}$
8   **foreach** $j \in [1, n]$ **do**
9   **if** $j \neq i$ **then**
10             $p_j^{new} \leftarrow p_j^{old} + \dfrac{update}{n-1}$
11   **end**
12   **end**
13 **end**

Updating Selection Probabilities: The random noise e_t to answer $f_t(\mathcal{DB})'$ can be high which will decrease the accuracy of the GWAS analysis. Hence, our proposed method reduces the selection probability p_i for such algorithm $\mathcal{A}_{BP}(i)$, which incurs larger noise and erroneous results. This selection probability reduction of a DP algorithm is named as *penalty* hereafter.

Since the range of each DP class depends on the harmonic series with uneven range of privacy budgets, the penalty should accurately reflect this property. In other words, the penalty for any p_i will also depend on its range $\left[\frac{\epsilon_{max}}{i}, max(\epsilon_{min}, \frac{\epsilon_{max}}{i+1})\right)$ such that:

$$\begin{aligned}
p_i^{new} &= p_i^{old} \times \left[1 - \frac{1}{\epsilon_{max}/i}\left(\frac{\epsilon_{max}}{i} - max(\epsilon_{min}, \frac{\epsilon_{max}}{i+1})\right)\right] \\
&= p_i^{old} \times \left[1 - \frac{\epsilon_{max}(i) - \epsilon_{min}(i)}{\epsilon_{max}(i)}\right] \\
&= p_i^{old} \frac{\epsilon_{min}(i)}{\epsilon_{max}(i)}
\end{aligned}$$

Here, the amount of penalty, $p_i^{new} - p_i^{old}$ reflects the range of the privacy budgets

that $\mathcal{A}_{BP}(i)$ operates on as it is normalized by $epsilon_{max}$. Conceptually, the range $\epsilon_{max}(i) - \epsilon_{min}(i)$ (penalty) for the more noisy (or lower ϵ's) algorithms will be lower than the less noisy ones. For example, for $i = 0$ and $i = 4$, the new probability will be proportionate to $\frac{1/2}{1} = 0.5$ and $\frac{1/5}{1/4} = 0.8$, respectively. In other words, the less noisy class ($i = 0$) will loose 50% of its current selection p_i as more noisy class's ($i = 4$) p_i will only loose 20%.

This allows the method to reuse the more private DP classes along with less noisy ones. However, the difference of the selection probability for such penalties $p_i^{old} - p_i^{new}$ are also redistributed *evenly* among the other class:

$$p_j^{new} = p_j^{old} + \frac{p_i^{new} - p_i^{min}}{n-1}, \forall i \neq j \tag{6.2}$$

In Algorithm 13 we outline the steps for answering a query $f_t(\mathcal{DB})$. Here, the *update* value denotes the probability that will be equally distributed in line 7. Furthermore, the threshold T was set according to the number of individuals, $1/\sqrt{m}$ which is essential in updating p_i. The runtime of the query execution will be linear to $\mathcal{O}(n)$ for n number of DP algorithms.

6.3.3 Local Model $\mathcal{A}_l$

In the local differential privacy (LDP) model, $\mathcal{A}_l$, we will incorporate the same ϵ splitting and categorization technique, as mentioned in Section 6.3.2.1. Here, the data owners will share individual statistics for each online query (Section 6.3.3.1) or the whole genomics data with a privacy guarantee (Section 6.3.3.2) to the central aggregator. We propose two randomized mechanisms to achieve such a privacy guarantee:

6.3.3.1 Laplace Mechanism

The naive approach to achieve $\mathcal{A}_l$ will be to extend the aforementioned method from the global model in a local setting. Initially, the query is broadcasted to all data owners by the central aggregator, and each owner employs the Laplace mechanism

Algorithm 14: Randomized response on genomics data

Input: n DP algorithms $\mathcal{A}_l(i) = \{\epsilon_i, p_i\}$, genomics sequence of one individual $\mathcal{DB}_u$ from the dataset $\mathcal{DB}$, Threshold T

Output: Local Differentially Private Sequence $\mathcal{DB}_i$

```
2  A_l(i) ← Select one DP algorithm based on p_i
3  ε ← pick ε_i from A_l(i)
4  rand ← uniform(0, 1)
5  noise ← 0
6  foreach randomly selected individuals do
7      if rand < 1/(1 + exp(ε)) then
8          SNP ← SNP'
9          noise ← noise + 1
10     end
11     if noise > T then
12         p_i^new ← p_i^old · ε_min(i)/ε_max(i)
13         update ← p_i^new − p_i^old
14         foreach j ∈ [1, n] do
15             if j ≠ i then
16                 p_j^new ← p_j^old + update/(n − 1)
17             end
18         end
19     end
20 end
```

on the count statistics. Later, the query results are accumulated by the aggregator to output the final result. Here, the data owners will add the noise according to the method demonstrated earlier in Algorithm 13.

However, since data owners set their different privacy budgets for their datasets, the final output from the central party will contain more errors. Furthermore, in the global model, only the aggregator added noise to the whole analysis, whereas in the local model, individual owners are adding noise to the in-between results. Therefore, the central server also accumulates all errors from the data owners, resulting in a more erroneous analysis. Notably, the local noise is added to the contingency table, as detailed in Section 6.3.1.

6.3.3.2 Randomized Response

Another approach to achieve LDP is to use Randomized Response (`RR`) [211]. The technique was proposed in 1965 as a statistical tool to remove potential bias and add probabilistic noise to surveys or consensus results. Originally, for a given boolean question, the oracle can answer in/correctly based on a random coin toss. There have been several variants and applications of this mechanism utilized in the context of differential privacy.

In this work, we utilize `RR` to manage the variable privacy budgets $\epsilon(i)$. In our local model, the data owners alters the nucleotide values of the SNPs based on $\frac{1}{1+\exp(\epsilon)}$ probability. The algorithm 14 denotes the method where we do such perturbation in line 8. For example, if the original SNP value was 0, it will be either 1 or 2 (with same probability) after alteration. Similar to line 5 in algorithm 13, line 11 reduces the probability of a erroneous ϵ class. The noise threshold T is set as $1/\sqrt{m}$ where $m = |\mathcal{DB}|$.

Theorem 6.3.2 ($\sum_{i=1}^{n} \epsilon(i)$ differential privacy)*.* Algorithm 14 is $\sum_{i=1}^{n} \epsilon(i)$ differential private.

Proof. For any local DP algorithm $\mathcal{A}_l(i)$ following Algorithm 14 which randomly picked an ϵ_i, it can change any SNP with $p = \frac{1}{1+\exp(\epsilon_i)}$ probability. On the contrary, the probability for that corresponding SNP keeping its original value will be $p' = 1 - \frac{1}{1+\exp(\epsilon_i)} = \frac{\exp(\epsilon_i)}{1+\exp(\epsilon_i)}$. Notably, as all $\epsilon_i > 0$, we will have $p' > p$. In other words, the probability for a SNP to keep its original value will always be higher as it is necessary for the accuracy of the GWAS.

Now, lets assume an input genomics sequence $\mathcal{DB}_i = \{SNP_1, \ldots, SNP_n\}$ differs in at least one SNP as $\mathcal{DB}'_i = \{SNP'_1, \ldots, SNP_n\}$ where $SNP_1 \neq SNP'_1$ after

following Algorithm 14. Therefore,

$$\begin{aligned}
\frac{\mathcal{P}[\mathcal{A}_l(\mathcal{DB}_i) \in \hat{\mathcal{DB}}]}{\mathcal{P}[\mathcal{A}_l(\mathcal{DB}'_i) \in \hat{\mathcal{DB}}]} &= \frac{\mathcal{P}[\mathcal{A}_l \rightarrow SNP_1] \ldots \mathcal{P}[\mathcal{A}_l \rightarrow SNP_n]}{\mathcal{P}[\mathcal{A}_l \rightarrow SNP'_1] \ldots \mathcal{P}[\mathcal{A}_l \rightarrow SNP_n]} \\
&\leq \frac{\mathcal{P}[\mathcal{A}_l \rightarrow SNP_1]}{\mathcal{P}[\mathcal{A}_l \rightarrow SNP'_1]} \qquad \text{[as only } SNP_1 \neq SNP'_1\text{]} \\
&\leq \frac{p'}{p} = \frac{\frac{\exp(\epsilon_i)}{1+\exp(\epsilon_i)}}{\frac{1}{1+\exp(\epsilon_i)}} \\
&\leq \exp(\epsilon_i).
\end{aligned}$$

As Algorithm 14 utilizes n DP algorithms with $\epsilon_1, \ldots, \epsilon_n$, according to Lemma 6.3.4.2 (composition property), it will lead to $\sum_{i=1}^{n} \epsilon(i)$ differentially private. □

6.3.4 Privacy Composition with Online Bin Packing

The privacy loss from the proposed method can essentially be composed of summing the ϵs from each query. However, in this section, we utilize a modified version of Online Bin Packing [33] to employ the sequential and parallel composition technique [201] to reduce the privacy loss.

6.3.4.1 Online Bin Packing

We alter the traditional bin packing problem which is discussed in Section 6.2.4. Here, we change the size of each bin and add a new constraint. Traditionally, the bins have a capacity of $(0, 1]$ and items can be of $\{s \in \mathbb{R} | 0 < s \leq 1\}$ size. The goal of the problem is to minimize the number of bins to fit all available items. We modified the bin packing problem as,

Definition 6.3.1 (Modified Bin Packing)**.** Suppose, a query q covers a disjoint dataset sampled from the original dataset $\mathcal{DB}$ where it considers n' records where $n' \leq |\mathcal{DB}|$. Now, for bins of maximum size $|\mathcal{DB}|$, *Bin Packing* will require placing these n' items into the minimum number of bins. Additionally, same records cannot get packed into the same bin.

In this modified version, the bins represent the full $\mathcal{DB}$, whereas the items denote the individuals present in the dataset. Hence, we altered the bin size into $(0, |\mathcal{DB}|)$ and made the items non-fractional. For example, if a query outputs $n' = |\mathcal{DB}|$ (full dataset), then it will require a whole bin to fit every item.

These modifications will allow us to use online bin packing to reduce privacy loss via parallel composition. We can process each record from the query output (disjoint dataset) and potentially perform dataset partitions, which we discuss in Section 6.3.1.

The last constraint does not permit the same records to be put together in the same bin. Hence, if any bin contains a certain item (or record), it cannot hold the same item in that bin for any of the upcoming queries. This constraint is available in popular applications of bin packing, such as audio mixtapes or disks. For example, no compiled mixtape will repeat any track as it will waste precious track time.

In Algorithm 15, we show the bin packing mechanism where $|b_i|$ denotes the current size of the ith bin whereas $b_i(\epsilon)$ is the highest privacy budget spent for all items stored in that bin. We can pack incoming (online) items into an existing bin only if it satisfies the three conditions. Firstly, the bin needs to have at least $|n'|$ empty space to accommodate the new items. Then, if $b_i(\epsilon)$ is greater than the query's privacy budget, we can pack these items according to the non-repeating constraint in Line 5 of Algorithm 15.

Furthermore, the privacy budget of a bin, $b_i(\epsilon)$ is updated if the size of the bin $|b_i|$ is smaller than $|n'|$ (current query size). It allows us to utilize the empty spaces of an existing bin with lower ϵ. For example, with the three conditions as mentioned above, a bin with a minimum budget $b_i(\epsilon) = \epsilon_{min}$ will waste the leftover space as we can only add items with ϵ_{min} here. However, if we have a query with higher privacy cost $q(\epsilon) > \epsilon_{min}$ and its size $|n'|$ is larger than the bin's current size $|b_i|$, we can add new items to it. Additionally, we will be updating the ϵ of the bin to $b_i(\epsilon) = \epsilon_{max}$.

Suppose we have a dataset containing 1000 individuals where an analyst is interested in a set of 500 individuals, which is defined by his/her inclusion criteria. As the dataset was partitioned earlier, the private output will be given only on this partitioned dataset of 500 participants. If the query was answered with ϵ_{max} budget randomly, and the ids (as items) used on this query were stored to the first bin, the

budget of the first bin $b_1(\epsilon)$ will be set as ϵ_{max}.

For the next query, the analyst queries for the other 500 individuals partitioned from the original dataset. Suppose the query is answered with ϵ_{min} randomly and can consider packing these with the first bin. However, if there is an intersection on one individual from the first query set with the second one, we cannot place the new items in the same bin. Hence, in that case, a new bin is created that only contains the intersecting item and set $b_2(\epsilon) = \epsilon_{min}$.

Now, if the next query requires any subset ($|n'| < 1000$), we can pack it on this second bin, updating the $b_2(\epsilon)$ with the query budget $q_2(\epsilon)$ if it is larger than ϵ_{min}. Here, the first bin is completely full of 1000 items, which is the maximum capacity. Notably, if the second query required more than 500 individuals, then a new bin was required.
Notably, Line 3 of Algorithm 15 dictates that the bins have at least the query size $|n'|$ as empty space to be considered for packing. This constraint allows us to perform the costly intersections only between the items from query and items in the b_i bin. However, this constraint can be omitted if we can pack fewer records n'' into b_i where $n'' \in n'$. In worst case, b_i already contains n'' and we proceed to next bin b_{i+1}. Nevertheless, the constraint on Line 4 and 7 only allows us to pack into the bins with *higher budgets*. In summary, Algorithm 15 allows us to mix disjoint queries with higher budgets with lower ones as we utilize sequential and parallel composition, which is discussed next.

6.3.4.2 Privacy Analysis

Generally, privacy composition in DP algorithms denotes the sequential composition property. For example, if a DP algorithm with ϵ budget is applied k times sequentially over the same dataset $\mathcal{DB}$, it will correspond to be $\sum^k \epsilon$ differentially private. Formally,

[Sequential Composition [77]] For k queries on $\mathcal{DB}$ answered with $\mathcal{A}_{BP(i)}$ algorithms each providing specific $\epsilon(i)$ privacy guarantee ($s.\ t.\ i \in [1, n]$), any composition $\mathcal{A}_{BP} = \{\mathcal{A}_{BP(1)}(\mathcal{DB}), \ldots \mathcal{A}_{BP(n)}(\mathcal{DB})\}$ will be $\sum_{i=1}^{k} \epsilon(i)$ differentially

private. Furthermore, there is a parallel composition available where we take the maximum privacy loss from n DP algorithms [78]. For example, we partition the original dataset $\mathcal{DB}$ into k independent partitions and have $\mathcal{DB}_1 \dots \mathcal{DB}_k$ disjoint datasets. Now, if we apply DP algorithms with different budgets $\epsilon(i)$ on these datasets, the final privacy loss will be the maximum of these individual budgets.

[Parallel Composition [78]] For k queries on disjoint $\{\mathcal{DB}_1, \dots \mathcal{DB}_k\} \in \mathcal{DB}$ and DP algorithms each with $\epsilon(i)$ loss, the total privacy loss from $\mathcal{A}_{BP} = \{\mathcal{A}_{BP(1)}(\mathcal{DB}_1), \dots \mathcal{A}_{BP(n)}(\mathcal{DB}_k)\}$ will be $\max(\mathcal{A}_{BP}(\epsilon))$ where $n \leq k$. We can extend this Lemma 6.3.4.2 to the following proposition:

Proposition 6.3.1 (Privacy loss of items in a bin). Each bin $b_i \in b$ will follow the parallel composition and have a privacy loss of the maximum of all budgets from the items packed in that specific bin $max(b_i(\epsilon))$.

Theorem 6.3.3 (Privacy bounds of Parallel Composition). For k queries, if l bins $b \in \{b_1, \dots b_l\}$ each with budget $b_i(\epsilon)$ were employed where $l \leq k$, the privacy loss will be $\sum_{i=1}^{l} max(b_i(\epsilon))$.

Proof. The existing bins from the online bin packing keep track of the records covered by each query from the analysis. Here, the records do not represent the outputs of a query but all records that were required by the incoming online queries. As the maximum ϵ used for each bin represents only the privacy loss for that individual bin (Proposition 6.3.1), collectively, all bins will fall under the sequential composition according to Lemma 6.3.4.2.

Here, each bin denotes a copy of the full dataset without any (disjoint) partitions. Therefore, the final privacy loss will accrue all the maximum losses from each bin according to Proposition 6.3.1 and be $\sum_{i=1}^{l} max(b_i(\epsilon))$. □

6.4 Results

In this section, we analyze the accuracy and utility (or accuracy) of the privacy-preserving GWAS functions considering different settings. Specifically, we are interested in understanding the relations among: a) global or local privacy model

(de/centralized setting), b) accuracy of DP algorithms, and c) corresponding privacy loss.

6.4.1 Dataset

For the first four GWAS analysis, we generated a synthetic dataset from publicly available *1000genomes* dataset [212]. We recorded the allele frequency of the underlying population and generated random SNP values based on these ratios. Later they were divided into case, control groups randomly and named as *synthetic data*. Nevertheless, our synthetic dataset had 500 individuals with 2,000 SNPs. For the distributed local model, we randomly split the whole dataset into three data owners. The four aforementioned functions (LD, HWE, CATT, FET) were then executed on this dataset in three settings–*a)* no noise in $f(\mathcal{DB})$, *b)* noisy $f(\mathcal{DB})'$ with $\mathcal{A}$, and *c)* noisy $f_l(\mathcal{DB})'$ with $\mathcal{A}_l$. The accuracies were calculated by taking the absolute difference of $f(\mathcal{DB}) - f(\mathcal{DB})'$. The other three benchmarking GWASs (TDT, EigenStrat, LMM) employed the dataset publicly available from Wang *et al.* [8].

6.4.2 Experimental Setting

We analyze the utility of the proposed methods in terms of accuracy and the accumulated privacy loss from each query. Here, accuracy is defined as the frequency of mismatches between the differentially private and non-private GWAS results. For example, for 100 queries, if our private GWAS outputs do not agree on 10 queries with the regular (w/o privacy) results, the accuracy will be 90%. We iterated this process for a fixed number of times as we implement the following GWAS considering different privacy budgets: *a)* Linkage Disequilibrium (LD), *b)* Hardy-Weinberg Equilibrium (HWE), *c)* Cochran-Armitage Test for Trend ($CATT$), and *d)* Fisher's Exact Test (FET).

Here, one example LD or HWE query from the analyst can be: 'Does rs140068132 holds LD or HWE?'. This GWAS query will require a contingency table (Section 6.3.1.2) where we introduce the differentially private noise and calculate the outlined statistical function. The results for these four GWAS are given as 0 or 1, which can

denote the disease correlation with the query SNP. The accuracy of the result is then determined by taking the absolute difference between the private and non-private binary GWAS output. This process is represented as a single GWAS query as we simulate 1,000 of such queries, randomly selecting a different SNP. Furthermore, as we select the ϵ values randomly from the harmonic series, we iterate these thousand queries for 10 times and report the average here.

Furthermore, three other GWAS tests were included Transmission Disequilibrium Test (TDT), EigenStrat, and Linear Mixed Models (LMM) to benchmark our algorithm with two previous works [7, 8] as their codes were readily available online. We follow the same experimental setup from [7, 8] and compare the accuracy of our method on the same privacy budget. One fundamental difference between the first four and these three functions is the output retrieved from the GWAS. TDT, EigenStrat, and LMM were used to generate a top-k SNP list, whereas previously, we received binary outputs. Here, the accuracies were computed by considering the set intersection between the list coming from non-private and private mechanisms.

Since our DP algorithms offer differentially private noise addition over a fixed budget in an online query setting. This demonstrated the applicability of the proposed method towards arbitrary GWASs. The implementation is readily available on GitHub [213].

We run the GWAS studies for different classes as they differ on the amount of noise it can allow into the GWAS analysis. We vary the $(\epsilon_{min}, \epsilon_{max}]$ values of these classes as they will demonstrate the relationship between ϵ and accuracy:

1. Class A: $(0.5, 3]$ (*low* noise),
2. Class B: $(0.02, 1]$ (*medium* noise), and
3. Class C: $(0.05, 0.25]$ (*high* noise)

Here, Class B, for example, has four DP algorithms (example 6.3.2.1) where ϵ_{min} and ϵ_{max} will be 0.2 and 1, respectively. The individual ϵ values for these four DP algorithms are also selected randomly among the corresponding intervals according to Section 6.3.2.

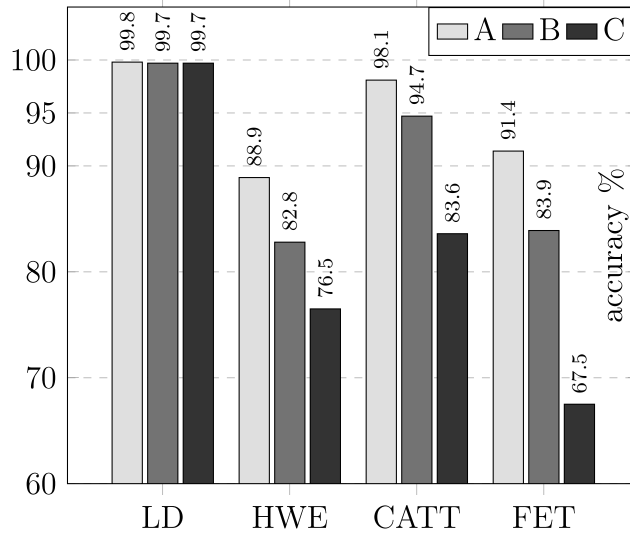


Figure 6.4: Accuracy of GWAS (LD, HWE, CATT and FET) averaged over 1000 queries and 10 iterations with three different privacy budgets

6.4.3 Global Model

Accuracy: Our global model considers a single source of noise added by the CA as it has a centralized trust model (Section 6.3.2). We show the results from 1000 random queries over 10 iterations for LD, HWE, CATT, and FET studies in Figure 6.4. The accuracy values over all four studies agree on the producing lower accuracy from more noisy class. For example, Class C performed inadequately on Fisher's Exact Test (FET) as it attained 67.5% accuracy, which increased to 91.4% for Class A. Similar trend is present in other GWAS studies as well. The detail of these average accuracy values and their standard deviation from 10 separate executions are reported in Table 6.9) as well.

Privacy: We also calculated the corresponding privacy loss to perform these GWASs. Table 6.5 shows the average privacy loss for all three classes and four studies. Analogous to the accuracy results in Figure 6.4, the more accurate results came at a price incurring higher privacy budgets. Notably, we considered all queries done on the whole dataset without any parallel composition, as discussed in bin packing (Section 6.3.4), as we wanted to portray the worst-case privacy loss.

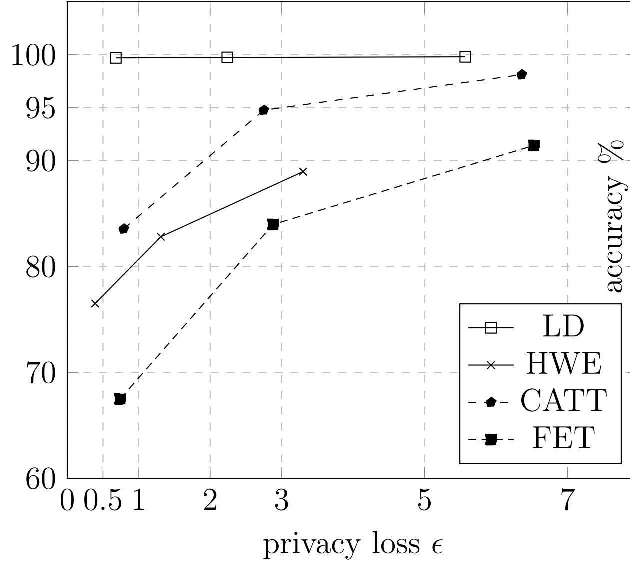


Figure 6.5: Privacy loss and accuracy relation for LD, HWE, CATT and FET where x-axis and y-axis denotes privacy loss ϵ and accuracy respectively

Table 6.5 demonstrates that the privacy losses gradually reduce from A towards C. For example, the highest loss is 6.53 for Fisher's test using class A. Here, losses are also linear to the number of queries required by the underlying GWAS. For example, CATT and FET require more queries to construct their contingency table compared to HWE or LD as they required case, control populations. We discuss this privacy loss in detail in Section 6.5.1.

From Figure 6.4 and Table 6.5, we see that there is reciprocal relation between accuracy and privacy. We compile these values and show the relationship between privacy and utility for all four GWAS studies in Figure 6.5. The figure demonstrates the accuracy decay with a lower ϵ values. For example, the CATT accuracy decreases from 98% to 83% as ϵ decreases by 5.57. Similar downward trends are visible for the other studies as well in Figure 6.5. However, the amount of accuracy (or privacy) loss or gain is not uniform for all GWAS, which is clear from the four studies.

Benchmarking Results: In Table 6.6, we benchmark the global model against Simmons *et al.* [7] which considers privacy-preserving EigenSTRAT and LMM studies. The authors proposed a privacy-preserving GWAS study using the DP mechanism relying on preprocessing the genomics dataset prior to the study. These

studies output a list of top-k SNPs for population stratification [214]. Here, we set $k = \{5, 10\}$ as we generate a top-5 or top-10 list based on our differentially private methods.

We compare the accuracy of both top-10 lists, comparing them with the original non-private top-10 list using the same privacy budgets. The accuracy metric here relies on the *number of matches* in the top-10 most significant SNPs. For a higher budget with class A, we have a 90% match (9 out of 10) compared to the non-private list incoming from EigenSTRAT while our method match 8 among the top-10 on LMM study. However, on the same dataset and privacy budget, Simmons *et al.* [7] offered 80% and 70% accuracy, which is comparable to our results.

6.4.4 Local Model

In our local model, individual data owners add noise before sending intermediary results to the CA. Compared to the global model 6.3.2, the local (decentralized trust) model will have multiple source of noise as we discussed in Section 6.3.3. In Table 6.7, we show the results for the local model from the Laplace mechanism and randomized response, considering the four GWAS functions. The experimental settings and data are the same as accuracy values are averaged on 1,000 random queries over 10 iterations considering three different budget classes. Here, we equally distribute the statistics among *three* data owners. All queries are answered from these three separate datasets as noise values are added according to the Laplace mechanism (Section 6.3.3.1). For the randomized response mechanism, the SNPs on the sequences are perturbed prior to any query, and the statistics are sent to the aggregator (Section 6.3.3.2).

Analogous to the earlier results from the global model, higher privacy budgets lead to a more accurate study, whereas the lowest budgets are erroneous. For example, even with the highest budget in HWE analysis, Laplace and RR both gained around 81% and 86% accuracy. However, accuracy from the Laplace mechanism and RR is apparent for multiple studies as RR performed better in most cases. The laplacian noise addition here is the same as the global model, only differing on the number of

sources. We show the effect of varying the number of data owners in Table 6.10. In comparison, RR perturbs the SNPs (randomly), which is essentially different than both of these methods. This is discussed further in Section 6.5.2.

Benchmarking Results: We also benchmark our local (and global) method with Wang *et al.* [8]'s work, where the authors proposed multiple approaches (*i.e.*, lap, exp stats) for another GWAS: Transmission Disequilibrium Test (TDT). Similar to the EigenStrat or LMM studies mentioned above, it also depends on a contingency table from SNP statistics as it returns a list of top-k SNPs. We incorporate our local and global model to the TDT queries and show the top-10 accuracy in Table 6.8.

Here, our global model performs better in comparison to lap.stats and exp stats [8] as we have an accuracy of 72% . The accuracy results are also comparable in the naive local setting, as it was 64% accurate. We could not benchmark our RR technique as their proposed method used summary statistics from the genomics data, and the original sensitive genomics sequence (data) were not available on the public repository. As our Laplace mechanism performs better than their baseline, we argue that the results from RR should be competitive as well.

6.5 Discussion

Experimental results in Section 6.4 demonstrate the conflict between privacy and accuracy as it shows that accuracy improvements involves privacy loss, which is a parameter controlled by CA or data owners. In this section, we discuss the other intricate details of privacy and utility results and note the limitations of the work.

6.5.1 Privacy

The privacy loss from the proposed private methods are related to the total ϵ value as higher values imply more privacy loss. In this work, we used for a maximum privacy budget (or loss) of 7 for the top-k queries on TDT, EigenStart and LMM studies (Table 6.7, 6.8). It is important to understand the implications of higher ϵ values as the ideal privacy is $\epsilon = 0$, which can prove to the challenging from utility standpoint.

For example, in Definition 2.2.2, $\epsilon = 0$ will result in $\mathcal{P}[\mathcal{A}(\mathcal{DB}_1) \in \hat{\mathcal{DB}}] \leq \mathcal{P}[\mathcal{A}(\mathcal{DB}_2) \in \hat{\mathcal{DB}}]$ which denotes that the probability for choosing $\mathcal{DB}_1$ (original) is equal (or less) compared to $\mathcal{DB}_2$. Details on the privacy budget is further discussed in Section 2.2.3.

On the contrary, higher ϵ values will tilt the balance towards $\mathcal{P}[\mathcal{A}(\mathcal{DB}_1)]$ which denotes that the queries will be answered from the original dataset $\mathcal{DB}_1$. Therefore, the privacy losses were higher for class A compared to B or C as the maximum allowed ϵ was 3. Nevertheless, it did not take $\epsilon = 3$ per query due to the random selections, which is highlighted in Table 6.5. As HWE requires three different queries to get the final binary result, the total loss was only 3.3 for class A. Similarly, class C only required $\epsilon = 0.39$ although the budget $(\epsilon_{max}, \epsilon_{min}]$ were between (0.25, 0.05]. This demonstrates that the privacy loss does not accumulate ϵ_{max} for all intermediate queries but utilizes the different DP algorithms defined over $(\epsilon_{max}, \epsilon_{min}]$.

Nevertheless, the total privacy loss for a GWAS query relies on the number of in-between queries that are performed to get the genomics statistics (*i.e.*, $m_i^{case}, m_i^{control}$ as discussed in Section 6.3.1.2); not solely on the targeted accuracy. For example, the GWAS requiring a higher number of queries will entail more privacy loss. In Table 6.5, FET or CATT requires more ϵ compared to LD or HWE studies under all privacy class. As CATT and FET both require 6 count queries compared to 4 and 3 for LD and HWE, respectively, the privacy losses are equivalent to the number of queries performed. Therefore, for an arbitrary GWAS function, the number of queries needs to be considered while setting the privacy loss.

6.5.2 Utility

Noise susceptibility of GWAS: In this work, we analyze utility in terms of accuracy for all GWAS studies. As discussed earlier, the accuracy of the studies depends on the allowed privacy loss ϵ. However, it is be easily seen that not all GWAS behave similarly in terms of accuracy given the same privacy budgets. Here, accuracy depends on the sensitivity of the functions along with the noise. For example, CATT gains 83.6% accuracy with class C privacy, whereas FET reports around 67.5%. Hence, the formulations for FET is more sensitive compared to CATT and needs more

privacy budget to achieve higher accuracy.

Top-K accuracy: In both benchmark GWAS, we have comparable results with [8, 7]. Given the same privacy budgets, we perform similarly for EigenStrat and LMM as our methods attain the lowest of 70% accuracy. For TDT, we actually perform better than the earlier approach for top-10 queries. However, larger noise values from Class B and C DP algorithms reduce the accuracies as they are less than 80%. Especially, the top-10 accuracy is around 50% for some cases on TDT study, which denotes the error susceptibility of TDT for noisy inputs. Nevertheless, with $\epsilon = 7$, we attain 72% according to the outlined setting, which is the best result compared to [8].

Accuracy in global and local model: Fundamentally, the accuracy in the local model should be lower than the global one in our current experimental setup. As every data owners individually add noise into the final results on the local model, these additional noise sources adversely impact the intermediary statistics. Though the privacy is ensured from the individual owners here, the global (/central) model can be less error-prone as the aggregator is the only responsible party for adding impurities into the analysis. Therefore, the global Laplacian mechanism, which has three noise sources compared to its local counterpart (with one source), performs better in all four GWASs. For example, FET attains 91.4% accuracy using Class A, whereas it drops to 81% in the local method. However, Randomized Response is different than randomized noise addition as it performed better against the Laplace mechanism. Even on the local model, it was comparable with the global one as we consider it as essential future work.

6.5.3 Limitations and Future Work

Privacy Preserving Data Publishing: In this work, we target the 'Privacy-Preserving Data Analysis' (PPDA), where the data analysts can only present their queries and get answers in return. However, there is another model with 'Privacy-Preserving Data Publishing' (PPDP), which disseminates the whole data to the analyst (with privacy guarantee). Here, PPDP models are more pragmatic for the researchers compared to the results from PPDA due to their arbitrary queries, and it

allows them to utilize the data at will. However, the privacy risks from publishing the whole dataset are considerable for opting towards the PPDA model. Nevertheless, in the future, we plan to extend the current models to an applicable privacy-preserving genomics data publishing one.

Inference attack: The proposed methods are not immune to the limitations inherited by the PPDA models [215]. For example, if an adversarial analyst repeats certain queries, s/he can infer more information as we did not sanitize or save results for such queries. In summary, we did not consider adversarial queries in this work which is an important future direction.

Abort Mechanism: In this work, we did not consider an abort mechanism based on a fixed privacy budget. For example, CA can set a constant value as the permissible privacy loss. When a set of queries accumulate ϵ's and go beyond that constant value, CA might prefer not to answer any more queries. Similarly, on the local model, individual data owners may also target a specific privacy loss and stop answering any queries after that. In both cases, previously given GWAS query results can be re-used as differentially private mechanisms are immune to post-processing attacks [201]. Nevertheless, this is an important future direction to consider.

Theoretical guarantees over accuracy: In this work, we did not investigate any theoretical accuracy guarantees from the GWAS functions and did not relate it to the privacy loss. We took a more experimental approach with different statistical studies relating the privacy and utility. In the future, we want to analyze the privacy of genomics data given an accuracy guarantee.

Randomized Response on Global Model: As Randomized Response (RR) performed better than the laplacian noise mechanism on the local model, in the future, we intend to use RR primarily. However, the original genomic data from the data owners need to be shared with CA, which will require a different trust model. Nevertheless, it will be interesting to examine the impact of alternating SNPs on the aggregated dataset.

6.6 Conclusion

In this work, we have shown an efficient way of managing the privacy budgets using an ensemble of differentially private algorithms. We further introduced online bin packing concepts to calculate the total privacy loss effectively. Our results for interactive or online settings show the efficacy of our scheme as it provides almost accurate results on Genome-Wide Association Studies. In the future, we would like to experiment on larger datasets and GWAS functions with different sensitivity ($\Delta f \neq 1$), which will emphasize the applicability of these schemes in realistic scenarios.

Availability of materials

The evaluation source code can be found at:
`https://github.com/mominbuet/DifferentialPrivacyGWAS`

Algorithm 15: Privacy composition under bin packing according to Definition 6.3.1

```
   Input: n' records for query q, privacy budget used for query q(ε), and all
          bins B
2  Procedure Pack(b_i,n')
3      res ← array()
4      foreach r_i ∈ n' do
5          if r_i ∉ b_i and |b_i| ≤ |DB| then
6              insert r_i → b_i
7          end
8          else
9              res.add(r_i)
10         end
11     end
12     return res
1  Algorithm main()
2      foreach b_i ∈ B do
3          if |DB| − |b_i| ≥ |n'| then
4              if b_i(ε) ≥ q(ε) then
5                  n' ← Pack(b_i, n')
6              end
7              else if |b_i| ≤ |n'| then
8                  n' ← Pack(b_i, n')
9                  b_i(ε) ← q(ε)
10             end
11         end
12     end
13     if |n'| > 0 then
14         create new bin b'
15         Pack(b', n')
16     end
```

Table 6.5: Privacy loss for each GWAS query on different budget classes without any dataset partitions

	LD	HWE	CATT	FET
Class A	5.57	3.3	6.36	6.53
Class B	2.24	1.3	2.75	2.88
Class C	0.68	0.39	0.79	0.74

Table 6.6: Comparison between Simmons *et al.* [7]'s ($\epsilon_A = 7, \epsilon_B = 5$ and $\epsilon_C = 2$) and our method on EigenSTRAT and LMM GWAS on top-5 SNPs

Work	Mechanism	Budget ϵ		
		A	B	C
Simmons *et al.*	EigenSTRAT	**0.8**	0.8	0.7
	LMM	0.7	0.7	0.5
Our method	EigenSTRAT	**0.9**	0.8	0.8
	LMM	0.8	0.7	0.7

Table 6.7: Local model accuracy using both Laplace and Randomized Response methods for LD, HWE, CATT and FET

GWAS	Laplace			Randomized Response		
	Class A	Class B	Class C	Class A	Class B	Class C
LD	99.84	99.74	99.64	99.86	99.76	99.6
HWE	81.1	76.3	73.1	86.6	83.4	78.6
CATT	96.1	89.7	68.7	97.7	90.6	84.8
FET	84	72.3	53.3	86.1	81	78

Table 6.8: Benchmarking on TDT with Wang *et al.* [8] ($\epsilon_A = 7, \epsilon_B = 5$ and $\epsilon_C = 2$) on top-10 SNPs

Methods	Mechanism	Budget ϵ		
		A	B	C
Wang *et al.*	lap.stats	0.54	0.52	0.51
	exp.stats	0.57	0.56	0.52
Our method	global	0.72	0.63	0.48
	local (Laplace)	0.64	0.52	0.51

Table 6.9: Average and standard deviation of the accuracy and privacy loss values of LD, HWE, CATT and FET over 1,000 random queries and 10 iterations

GWAS	Privacy Class	Accuracy		Loss	
		Average	StdDev	Average	StdDev
LD	A	99.8	0.11	5.57	0.45
	B	99.74	0.12	2.24	0.18
	C	99.7	0.29	0.67	0.15
HWE	A	88.96	1.02	3.3	0.24
	B	82.81	1.95	1.31	0.12
	C	76.51	1.58	0.39	0.09
CATT	A	98.12	0.33	6.36	0.45
	B	94.76	0.45	2.75	0.22
	C	83.56	2.78	0.79	0.17
FET	A	91.43	0.78	6.52	0.55
	B	83.97	1.71	2.88	0.32
	C	67.49	2.33	0.74	0.14

Table 6.10: Local model accuracy using Laplacian methods for HWE, CATT and FET from 3, 6, and 9 data owners

Privacy Class	Class A			Class B			Class C		
Data Owners	3	6	9	3	6	9	3	6	9
HWE	81.1	75.5	74.9	76.3	74.5	74.4	73.1	72.7	70.2
CATT	96.1	95.1	91.7	89.7	86.3	82.8	68.7	64.8	64.2
FET	84	78.9	77.8	72.3	68	63.9	53.3	53.2	50.6

Chapter 7

Generalized Genomic Data Sharing for Differentially Private Federated Learning

7.1 Introduction

Machine Learning (ML) techniques have penetrated every branch of computational studies due to their capability to to find hidden patterns and general abstractions by analyzing significant amount of data. Due to the nature of these methods, the inherent data dependency is paramount. The higher the volume of the data and the greater the variance in the data, the more generalized and useful abstractions these ML models can derive. This particular aspect makes dataset construction one of the most critical aspect of any ML system. However, dataset construction can be tedious and costly. Therefore, it is often considered ideal if multiple parties share their respective dataset in a federated setting to train a model.

Although sharing data to train model can help in constructing a robust model, it is often not practical or desirable due to privacy concerns associated with the data. For instance, sharing medical information of patients (*i.e.*, patient's medical history, diagnosis) or financial information of clients (*i.e.*, credit score, transaction history) are highly sensitive information, hence are prohibited from being shared among parties.

Most importantly, genomic data are one key area of sensitive data, revealing our well-being and disease susceptibility in some cases are also lawfully protected from public access. However, with the proliferation of data storage techniques, more and more organizations have access to significant amount of user data that can be better utilized by machine learning techniques for the advancement of healthcare research.

Federated Learning (FL) methods can reconcile between these seemingly opposing conditions and provide a general framework which can be used to execute ML algorithms over data that are shared among multiple parties and the subset of data that each party holds are mutually exclusive. Federated learning can have two settings namely trusted settings and untrusted settings. In a trusted federated learning setting, all the parties share raw data or aggregate information about the data or model parameters with the central entity which all the parties trust. The central entity trains the model and shares the final parameters with all the parties. An example of this setting is the government or national health authorities who often collaborate with hospitals and clinical organizations in order to determine efficacy of certain clinical trials or drug tests across a wide spectrum of population.

On the other hand, in untrusted settings, the parties don't trust the central entity and employ various privacy preserving methods in order to prevent leakage of private information. A prominent example of this type of setting is when multiple collaborating parties decide to use cloud computing platforms as central authority to perform the model computation. Since cloud computing platforms are owned by third parties who in most cases are not healthcare providers and are considered to have privacy violating properties, the data shared with them are usually encrypted (Homomorphic Encryption) or have noise added to them (Differential Privacy).

Contributions. In this work we propose a generalized data sharing framework for Federated Learning (FL) in an untrusted setting. Our framework has two steps. The first step comprises of differentially private feature selection and the second step is ML analysis on the selected features. The contributions can be summarized as follows:

- We propose a privacy-preserving feature selection mechanism based on variance

ranking of gene expression data to maximize the utility of the ML models. The proposed mechanism is also locally implemented as individual data owners can protect their participants before sharing their data.

- We employ a differentially private histogram-based discretization technique to enable privacy-preserving data sharing among collaborating parties.
- We demonstrate the effectiveness of our proposed technique by experimenting on iDASH 2020 competition dataset [216]. Our proposed method achieves 99.9% accuracy with a privacy budget of 3 and takes 10.84 seconds for training on a federated training setting. In this paper, we also added several other datasets and enhanced the mechanism while testing the method on different settings.

The paper is organized as follows. Section 7.2 describes the required background on the research problem addressed. Section 7.4 contains the proposed methods for privacy-preserving data sharing for federated learning mechanism. Experimental results are shown and discussed in Section 7.5 as we discuss them in Section 7.6. Finally, Section 7.7 presents the conclusion of the work.

7.2 Preliminaries

In this section we will provide a brief overview on some of the core concepts that are used to develop our proposed framework.

7.2.1 Differential Privacy

Differential Privacy has become the de-facto standard for privacy analysis as it provides one of the strongest privacy guarantees. The mathematical rigor of the privacy mechanisms in the domain of differential privacy makes them ideal to use for data sharing under a particular privacy budget. In this section we will describe the basics of differential privacy and explain the particular differential privacy method we used in this work.

Differential privacy was first proposed by Dwork *et al.* in [201]. Let's assume two databases D and D' that differs in at most 1 entry which we denote as $\|D - D'\| \leq 1$. A randomized function $\mathcal{M}$ with domain $\mathcal{N}^{\|D\|}$ is called (ϵ, δ) differentially private if the following equation holds for all $S \subseteq \text{Range}(\mathcal{M})$.

$$Pr[\mathcal{M}(D) \in S] \leq \exp^{\epsilon} Pr[\mathcal{M}(D') \in S] + \delta \tag{7.1}$$

Some of the most popular differential privacy mechanisms that are used in practice are noisy max count, exponential mechanism and laplacian mechanism. In this work we used exponential mechanism which will be described in the following paragraph.

Exponential Mechanism Exponential mechanism is a building block for preserving privacy of queries that have utility values associated with the result of each query. Let us assume that we have a database D on which we can run queries where the corresponding results have a fixed range $\mathcal{R}$. The utility function can be denoted as $u : D \times \mathcal{R} \rightarrow \mathbb{R}$. For a fixed database, the user will prefer that the mechanism in question returns the query result $r \in \mathcal{R}$ with the maximum utility score.

We denote exponential mechanism as $\mathcal{M}_E(D, \mathcal{R}, u)$ that selects output from every possible outputs of the query with probability equal to $\frac{\epsilon u(D,\mathcal{R})}{\Delta u}$ where ϵ is the privacy budget and Δu is defined as follows:

$$\Delta u = \max_{r \epsilon \mathcal{R}} \|u(D, r) - u(D', r)\| \tag{7.2}$$

In equation 7.2 D and D' are two databases that differ by at most one entry. In other words, $\|D - D'\| \leq 1$. Exponential mechanism provides $(\epsilon, 0)$ differential privacy guarantee.

7.2.2 Federated Learning

Federated learning is a ML technique by which a model can be trained on multiple hosts or devices using their own dataset. FL allows multiple parties to build a shared

ML model without sharing the data and thereby addressing issues such as data privacy, data security and data access control. Each host runs the model on their own dataset and shares the parameters or intermediate results. In our framework design, we used Naive Bayes Classifier and XGBoost as machine learning algorithms in the federated setting. The following subsections give a brief details on them.

Naive Bayes Classifier Naive Bayes Classifier is a supervised ML algorithm that uses Bayesian interpretation of probability where probability expresses the degree of belief in an event. Naive Bayes is a conditional probability model. Let us assume that we have a classification problem at hand with k classes $\{C_1, C_2, \ldots, C_k\}$. Consider, a data vector with n features, $x = \{x_1, x_2, \ldots, x_n\}$. For all k classes, Naive Bayes classifier assigns a probability of x being a member of that class. This can be mathematically represented by:

$$\mathbb{P}(C_i \mid x) = \frac{\mathbb{P}(C_i) \times \mathbb{P}(x \mid C_i)}{\mathbb{P}(x)} \text{ for } i \in [0, k) \tag{7.3}$$

In a simplified manner it can be represented as posterior $= \frac{\text{prior} \times \text{likelihood}}{\text{evidence}}$. The class with the maximum probability is considered the result of the classification problem. Naive Bayes requires only a small amount of data to estimate the necessary parameters for classification. This is a big advantage compared to the other ML models used in literature.

Naive Bayes is an algorithm which is easy to implement in federated setting. Firstly, parties calculate their individual components from their data. After that parties can take part in a secure aggregation protocol which can be implemented by homomorphic encryption, secure multiparty computation or trusted hardware. A central server usually manages the aggregation part. Based on the aggregation and subsequent computation, the final model is calculated and the result is shared with all the parties. Several implementations have been proposed over the years for both horizontal and vertical partitioned data. Kantarcioglu et al. proposed a distributed learning protocol where parties compute the probability from their local data and then use secure sum protocol to compute the global result [217]. Vaidya et al. proposed a

similar protocol for vertically partitioned data in [218].

Random Forest and XGBoost Random Forest is a supervised learning algorithm that can be used for both classification and regression problem. During training phase it creates multiple decision trees from different subsets of the data. In the testing phase, the incoming data is predicted by all the decision trees. Majority decisions or voting method is used to reach the final decision for the class prediction. For example, consider the example from Naive Bayes classifier section. We assume that the random forest has b number of trees. Their class predictions are $\{y_1, y_2, \ldots, y_b\}$. Therefore, the predicted class y is given by the following equation:

$$y = argmax(\sum_{j=1}^{k}\sum_{i=1}^{b}(y_i = j)) \tag{7.4}$$

Random Forest classifier removes the bias problem of the decision trees as decision trees tend to overfit the training data. However, they usually can not outperform gradient boosted tree algorithms. Therefore, here, we use train gradient-boosted decision trees using XGBoost [219] on a federated setting and also show the difference with Random Forest algorithm.

In a federated setting, parties each train their own decision tree. These decision trees go through some privacy preserving mechanism such as differential privacy, homomorphic encryption or secure multiparty computation and get aggregated at the central server. The final ensemble is shared with all the parties at the end of the training. Several approaches have been proposed in literature which follow this “locally learn and globally merge" pattern [220, 221, 222].

7.3 Related Works

Federated Learning (FL) frameworks symbolizes a data-distributed machine learning training setting, where the data never leave their location on its raw form [223]. This is a powerful mechanism as it restricts the data owners by enforcing a privacy-preserving method over the sensitive data. Recent works demonstrate the advantages of using

FL techniques over traditional centralized ML models for several health applications [224]. However, FL techniques are venerable to privacy attacks including inference [225], reconstruction [226], and backdoor [227] attack.

Therefore, there have been several academic research that considered the contradicting notions of privacy and data dependency in constructing FL or ML models. Abadi *et al.* showed a stochastic gradient descent model for preserving privacy while training large deep learning models [228]. In this work, the authors considered adding differentially private noise to the training process to prevent the network from memorizing private information about data. However, implementing the proposed method in a federated setting is a significant challenge which was noted by the authors. Geyer *et al.* proposed compressive sensing to minimize error during federated training session [229]. The authors of [229] argued that adding noise proportionally with the number of model's parameters can have detrimental effects on models with large number of parameters. Moreover, the authors did not specify any general model compression scheme for parameter sharing. Yang *et al.* utilized blockchain to enable data sharing in cloud [230]. The authors in [230] used the blockchain for the validation of the privacy budget.

However, generalized protocol for sharing private data to train machine learning models in a federated setting is a challenging problem and not yet fully explored in the literature. Ji *et al.* proposed a differentially private data sharing method for performing logistic regression in biomedical data in [231]. The authors of [231] modified the update step of Newton-Raphson method to perform distributed logistic regression on both public and private data. Li *et al.* also proposed a differentially private data synthesizing method that can enable data sharing among participants [232]. However, this method was not particularly targeted towards federated learning. Some researchers viewed differentially private histogram publication as a secure method of sensitive data release [233]. However, the authors did not explore the viability of this process for machine learning algorithms in a federated settings.

On the other hand, differential privacy has been previously used in genomic data analysis by machine learning algorithms. Chen *et al.* proposed a differential privacy based mechanism for protection against membership inference attack in machine

learning algorithms in [234]. In that work the authors showed that the relationship between privacy budget and the models in question can be represented as a log-like curve which essentially means that the higher the privacy of the model , the lower the accuracy will be of that model in case of genomic data. Azencott *et al.* explored the trade-offs between privacy and accuracy of usage of machine learning models in patient data in [235]. Raisaro *et al.* explored the genomics data privacy protection problem using differential privacy over i2b2 dataset [204]. Olivia *et al.* used differential privacy for secure federated learning over sensitive health data [236]. Their work proved that there are practicality issues in implementation of federated learning where the number of participants are small. There are several surveys for privacy-preserving genomic data operation available to this day [26, 1, 186, 237] that elaborate major privacy research areas and techniques used to tackle privacy issues concerning genomic data.

7.4 Methods

In this section, we describe the proposed methods detailing the privacy-preserving techniques. Firstly, we describe the problem below:

7.4.1 Problem Description

In this problem, we target a federated architecture where genomic data are collected in different location. Due to the privacy constraints, this data cannot be shared publicly or even to other data owners. However, these data owners want to train a ML model collaboratively that considers all data sources.

Figure 7.1 demonstrates a simplified version of the the problem scenario where two data owners intend to jointly train a ML model based on their dataset. The underlying ML algorithm and the hyperparameters are set as public knowledge. However, the genomic data required for the training are sensitive and need to be shared with a privacy guarantee. In summary, the privacy goal of this chapter can best be described as follows: a) protect the sensitive genomic data of the participants and b) output the final results with a privacy guarantee.

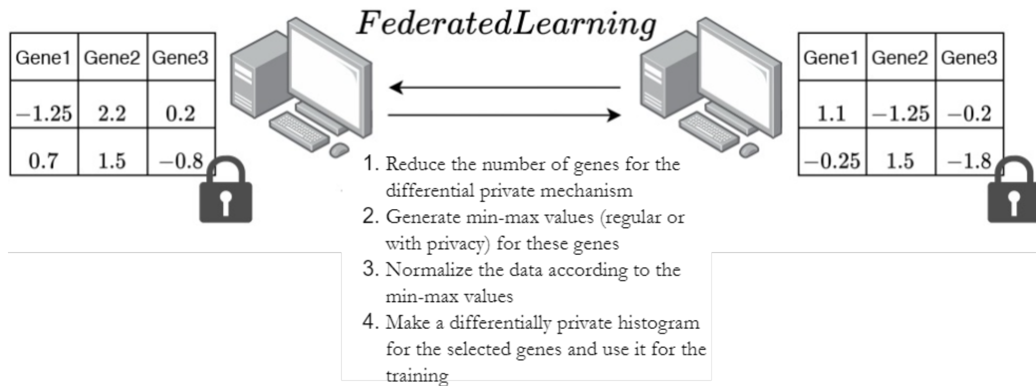


Figure 7.1: Overview of the privacy problem and proposed solution where multiple data owners are targeting to train a model collaboratively using arbitrary machine learning algorithm given a privacy guarantee over the data

In this work, we are considering numeric gene expression data that are categorized into two classes: normal and cancerous. For example, individual data owners can have a dataset of $n \times m$ size (n records with m genes) that are divided into two label groups. The goal here is to propose a cancer prediction model using arbitrary ML technique that can identify cancerous gene set. In other words, given a set of gene expression data from a new participant, the generated model should predict whether it will progress towards breast cancer [238]. We provide an example of the BC-TCGA dataset in Table 7.1

7.4.2 Summary of the proposed method

Our proposed method targets a privacy-preserving data sharing mechanism for any ML task on the genomic data. Firstly, it focuses to reduce the dimension of the data. For example, from a set of m genes, we only consider a smaller subset of size m'. Then, we utilized Differentially Private (DP) mechanisms to discretize the gene expression data.

The proposed DP method converts the numeric gene expression data into bins. Lets assume, the number of bins are set as $b = 10$ where the minimum-maximum range of the gene expression can be $(0, 100]$. Now, if a certain gene expression had a value of 80, it will be under 8th bin given each bin has an individual range of

Table 7.1: Gene Expression data collected at individual data owner (BC-TCGA)

#	$Gene_1$	$Gene_2$	$Gene_3$	...	$Gene_m$	Cancer
1	0.204	-0.242	0.591		0.538	1
2	0.869	0.878	-0.024		0.819	0
3	0.064	-0.814	0.478		0.29	1
⋮				...		
n	-0.186	-0.746	0.701		0.234	1

$(bi, bi + b]$ where $i \in \{0, 9\}$. Notably, gene expression values are numeric and the minimum-maximum value for each gene differs. In the differentially private setting, we select the bins using an exponential mechanism which precedes the ML training.

Such difference in the values of the genes along with the number of bins are of much importance in our proposed method. Therefore, let us define a particular gene expression value g_i to be in the range of $[g_i(min), g_i(max)]$. Data owners are denoted with j as each owner can have their own range of gene expression values defined as $[g_i^j(min), g_i^j(max)]$. In specific, the subscripts define local variables whereas the superscripts denote global ones. In Figure 7.7, we summarize the the differentially private data sharing protocol proposed in this work.

7.4.3 Reducing the Data Dimension

The first fundamental problem to solve here is the large number of genes available in a genomic dataset. It requires any ML algorithm to process a large dimension of data which often results in a poor performance. This is also termed as the curse of dimensionality [239] as some genes do not provide additional value to the analysis.

Therefore, in our first step, we reduce the number of genes that we consider for the underlying ML algorithm. The main challenge here is to select the genes on a federated environment. Since all data owners need to agree on the selected genes, the simplest solution would be to share the data among themselves. However, as the data are sensitive and individual owners are not releasing anything publicly which will require a privacy preserving mechanism.

However, any differentially private mechanism to reduce such large dimensions

will incur heavy privacy cost (in terms of ϵ). For example, if we want to use Principal Component Analysis (PCA) to select the genes in a differentially private manner, we will loose privacy budgets on PCA operations. Resultingly, we will end up with less budget on the ML which is the primary target of this work. Therefore, the variance among the different gene expression values is set as the selection criteria.

Variance among the gene values allows us to realize how these are spread with respect to their mean value. Most importantly, it can be utilized to rank the genes and perform set intersection among the data owners without loosing any privacy budgets. In other words, we assume the rank of the different genes based on their variance at their individual datasets are safe to share.

Individual data owners calculates the variances v_i from the gene expressions g_i for all available genes. Then the genes are sorted according to the highest variance. As the targeted dimension is m', data owners select $2m'$ genes for a set intersection. Then, these $2m'$ number of genes are shared among themselves and only the top m' matching genes are selected for the ML operations.

However, for smaller m' values, there is a risk of lower cardinality of the reduce dataset. For example, if the data owners set $m' = 5$, then there might not be a intersection of size 5 from a set of 10 genes. Therefore, the data owners may opt for larger multipliers for lower m' values or sparse gene expression datasets. We discuss this issue and potential solution in Section 7.6.

7.4.4 Privacy Preserving Mechanism

In our proposed method, the data owners publish a subset of genes for federated learning utilizing a differentially private method. In this section, we discuss the proposed mechanism to use exponential mechanism to privately share the gene expression data.

The proposed mechanism will create a histogram for the subset of genes selected from the reduced dimension (Section 7.4.4.1) that is differentially private and safe to share. Notably, the gene expression values can vary between arbitrary ranges. Hence, to create the histogram, the data owners need to know the min-max values and

agree on a predefined interval that will dictate the number of bins. We propose two approaches: a) Normalized and b) Private min-max, to retrieve the global min-max values which are described below:

7.4.4.1 Regular min-max

In this method, the data owners agree on minimum and maximum values and the interval required for the histogram. The predetermined min-max values will allow them to normalize the gene expressions into a generalized range among all data owners. Let the gene value be g_i while the preset min-max values are $g_i(min)$ and $g_i^(max)$,

$$g_i' = \frac{g_i - g_i(min)}{g_i(max) - g_i(min)} \tag{7.5}$$

Now, if the min-max value for these normalized gene is $g_i'(min)$ and $g_i'(max)$, respectively and the fixed interval for the histogram be agreed as an integer value b, then, the number of bins on the histogram would be:

$$h_i = \lceil \frac{g_i'(max) - g_i'(min)}{b} \rceil \tag{7.6}$$

Using h_i, we can create b bins where each bin will contain the values from $[g_i'(min) + jh, g_i'(min) + jh + h)$ where $j \in [0, b-1]$. This will allow the data owners to discretize the gene expressions into fixed bins. For example, if the normalized gene value is $g_i' = 15$, and $b = 10$ on a $(0, 100]$ min-max range, it will be pointed to the second bin. This process is iterated for all m' genes selected according to Section . Next, these fixed $0, b-1$ values will go through a differentially private process and shared for ML algorithms.

7.4.4.2 Private min-max

Since normalization and arbitrary min-max values limits the shared data and subsequent ML accuracies, we propose another method to generate the histograms. In this case, we utilize a portion of the privacy budget (ϵ) to gather the minimum and maximum values of the selected genes.

Algorithm 16: Generate private histogram for gene expression dataset

Input: Gene expression dataset G of size $n \times m$, reduced dimension size m', privacy budget ϵ

Output: Private histogram H of size $n \times m'$

1 Use variance to select $2m'$ genes from G
2 Collaborate with the other data owners and agree on m' genes
3 $\epsilon' \leftarrow \epsilon$
4 **if** *private min-max* **then**
5 | $\epsilon' \leftarrow \epsilon/2$
6 **end**
7 **foreach** *gene expression g^j∈ $G[m']$* **do**
8 | *calculate the local min-max at individual data owner's site*
9 | *Use regular or private min-max (using ϵ') to generate the global min and max expression values from all data owners*
10 | *create the local histogram H of size $n \times m'$ using the min-max value*
11 **end**
12 $\epsilon' \leftarrow \epsilon'/m'$
13 **foreach** *individual bin value b_ i^j∈ H* **do**
14 | *select a new bin b' with probability* $\propto \dfrac{exp(\dfrac{\epsilon' u(g, b_i^j)}{2\Delta u})}{\sum_{i \in |b|} exp(\dfrac{\epsilon' u(g_i', i)}{2\Delta u})}$
15 | *update $b_i^j \leftarrow b'$*
16 **end**
17 *Share H for any federated learning task*

Here, the data owners use a noisy-min and alternatively noisy-max algorithm to share the private min-max values respectively [201]. As a result, the final min-max value can be agreed from all owners and represented as $\bar{g}_i(min), \bar{g}_i(max)$. Each data owner can then calculate the range of the histogram following equation 7.5. However, we alter the number of bins by 2 as,

$$h_i = \lceil \frac{\bar{g}_i'(max) - \bar{g}_i'(min)}{b'} \rceil \tag{7.7}$$

We used $b' = b-2$ as the $\bar{g}_i(min), \bar{g}_i(max)$ values are not accurate representations of the minimum and maximum, respectively. Due to the additional noise from the noisy min-max procedure, we keep two extra bins that can hold gene expressions that

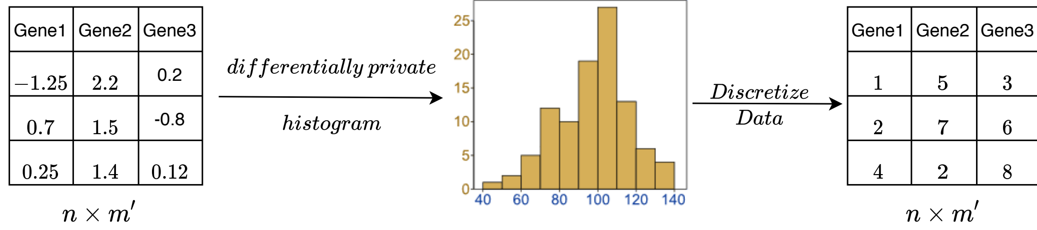


Figure 7.2: Generating differentially private histogram to release genomic data for federated machine learning algorithm

are either less than $\bar{g}_i(min)$ or greater than $\bar{g}_i(max)$.

7.4.4.3 Differentially Private Histogram Publishing

After creating the histogram, the data owners utilize a differentially private mechanism to share the data for training using a specific ML algorithm. Here, we define the utility function according to the histogram values retrieved after the min-max procedure. Notably, this is process is added regardless of the selected method either from Section 7.4.4.1 or 7.4.4.2. The following procedure will make sure that the published data is private and safe to share for ML tasks.

Here, we use exponential mechanism [201] to select the bin to place a gene expression value. The aforementioned process results in a histogram that represents the original values but neatly packed into fixed sized bins. For example, if there are 10 bins between an input range of $[0, 100]$, then a gene with value (normalized) of 12 will be on the second bin. With the exponential mechanism, this bin selection will be probabilistic according to $exp(\epsilon u/\Delta u)$ where u is a utility function.

The utility function $u : \mathcal{R} \rightarrow \mathbb{R}$ is defined as the opposite of the absolute distance between a particular bin number and the exact one. Considering the earlier example, the gene expression value 12 has a distance of 3 with the 4th bin. Therefore, for a total of 10 bins the utility of the 4th bin is considered as 7 whereas the first bin is 10.

Calculating the scores of all the candidate bins, exponential mechanism selects

a bin b_i' according to the following probability:

$$p(b_i) = \frac{exp(\frac{\epsilon' u(g, b_i)}{2\Delta u})}{\sum_{i \in |b|} exp(\frac{\epsilon' u(g_i', i)}{2\Delta u})} \tag{7.8}$$

We are using ϵ' here which is split among the m' dimensions. Notably, for private min-max 7.4.4.2 operation, ϵ is further split into two as we one half is used to retrieve the min-max values whereas the other half is used to generate the private histogram. The sensitivity of the utility function u is set as 1as it is represents a histogram query. We outline the process in Algorithm 16.

In Figure 7.2 we depict this mechanism where the inputs are numeric values of size $n \times m'$. Using Algorithm 16, the data is discretized into fixed range of values. This private histogram is later shared for the federated learning mechanism described below.

Privacy Analysis The privacy of the proposed mechanism relies on the privacy budget ϵ set by the individual data owners. In our proposed privacy preserving method, there are two specific places where we spend the privacy budget ϵ: Firstly, to select the m' genes according to private min-max mechanism (line 9). However, if the data owners normalize their data and utilize the regular min-max method, we do not spend any ϵ in this step. The main consumer of the budget would be the exponential mechanism as shown in line 14 of algorithm 16. Therefore, we prove the following by composition,

Theorem 7.4.1 (Differential privacy proof). The proposed mechanism is ϵ-differentially private.

Proof. Lets assume that the data owners use the private min-max mechanism. Here, the data owners use a portion of their privacy budget $\epsilon' = \epsilon/2$ to generate a noisy version of the gene expression. Noisy min-max method uses a Laplacian mechanism which adds probabilistic random noise to the original data. Since the noise is generated according to ϵ', we can state that the private min-max mechanism is ϵ'-

Table 7.2: Different experimental parameters considered in this work

Dataset	ML Methods	budget ϵ	dimensions
BC-TCGA	Naive Bayes	[0.5, 1, 3, 5, 10]	10
GSE2034	Random Forest		20
GSE25066	-		50

differentially private (according to [201]).If the data owners are using the regular min-max method, then no ϵ is spend in the dimension reduction phase ($\epsilon' = 0$).

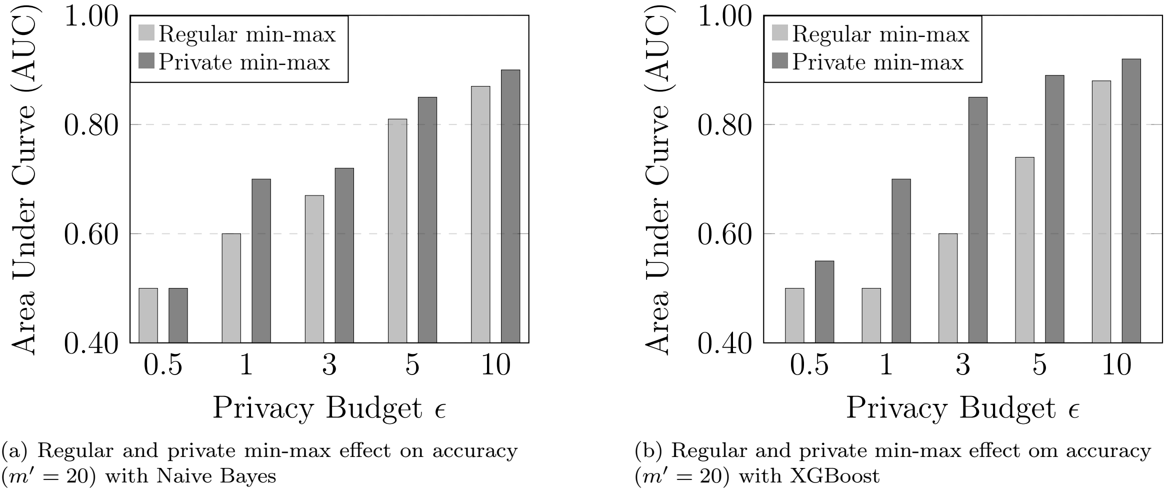
The remaining budget $\epsilon' = \epsilon - \epsilon'$ is split among the top-m' genes such as, $\epsilon' = \epsilon'/m$. Since we use the exponential mechanism to discretize the data, that mechanism would be ϵ'-differentially private as well. Therefore by composition, we have $\epsilon/2 + m'\epsilon/(2m') = \epsilon$ budget spent which proved that the proposed mechanism is ϵ-differentially private. □

7.4.5 Federated Learning Mechanism

The private histogram generated at individual data owners are then utilized for federated learning. In this work, we utilize with three different ML algorithms: a) Naive Bayes Classifier, b) Random Forests and c) gradient boosted decision trees with XGBoost [219] to validate the efficacy of the proposed method.

In Figure 7.7, we depict the privacy preserving mechanisms to generate the histogram that is utilized on the machine learning algorithms. Here, the data owners perform the same steps and share only the summary statistics with a differential privacy guarantee to generate the final dataset. This aggregate data is only forwarded to the ML operation. For example, the statistics required for the Random Forest and Naive Bayes are initially done at the owner's premises (on private data) and then shared among themselves for the final result. For XGBoost, data owners share the private histogram data which is used to train the gradient boosted decision trees. Nevertheless, in all cases, we select one data owner to be the aggregator of these in-between outputs.

Since the underlying histogram data is differentially private, the local statistics are safe to share. Therefore, we do not add any additional privacy-preserving



(a) Regular and private min-max effect on accuracy ($m' = 20$) with Naive Bayes

(b) Regular and private min-max effect om accuracy ($m' = 20$) with XGBoost

Figure 7.3: Accuracy difference with different privacy budgets and methods with a fixed reduced dimension $m' = 20$

mechanisms to further convolute the results. However, since we are proposing a private generalized data sharing mechanism, the ML algorithms are not limited to the ones experimented in this work. We can use the same data in different ML pipeline in a federated setting and achieve privacy over the local data produced at different owner.

7.5 Results

In this section, we describe the results varying several parameters. Since the proposed method is a generalized data sharing mechanism for down the line ML applications, we experiment with different settings as portrayed in Table 7.2. We utilized AWS cloud machines over us-east-1 and us-west-1 regions to conduct the Wide Area Networks (WAN) experiments. The average latency between the servers were around 62 milliseconds. The implementations is written in python which is freely available at [240].

7.5.1 Dataset and Experimental Setup

Since the problem is conceptualized from iDASH 2020 competition [216] which tested the proposed solutions with a single dataset: BC-TCGA [241]. However, we utilized two more breast cancer datasets that was publicly available. These additional datasets allowed us to test the generalization of the proposed method.

1. **BC-TCGA:** 17,814 genes with 424 positive labels and 50 negative labels
2. **GSE2034:** 12,634 genes with 144 and 84 positive and negative patients, respectively
3. **GSE25066:** Similar dataset with 17 negative and 100 positive instances

We use a 10-fold cross validation set for the accuracy results. The training data for the ML models were randomly selected according to a 80:20 split where 80% of the data were used in training. Since the positive and negative number of records are imbalanced in all three datasets, we created balanced sets with equal amount of positive-negative labels for testing first. The training data were split equally into two for both Naive Bayes and Random Forest (trained with XGBoost) in a 2-party setting.

We also experiment with different parameters, privacy budget ϵ, data dimension m' and privacy preserving min-max method (Section 7.4.4.1 and 7.4.4.2). We set the size of the bins as 10 while normalizing the data between $[0, 100]$ for regular min-max (Section 7.4.4.1). We outline these settings in Table 7.2.

7.5.2 Area Under Curve (AUC)

As the outlined ML algorithms will be binary classifiers, we will use Area Under Curve (AUC) as the primary accuracy metric to judge the models. Since AUC consider True Positive and False Positive rates (TPR and FPR) to create the curve (receiver operating characteristic), it genuinely depicts the picture of a binary classifier. It selects different thresholds from $[0, 1]$ and calculates the TPRs and FPRs.

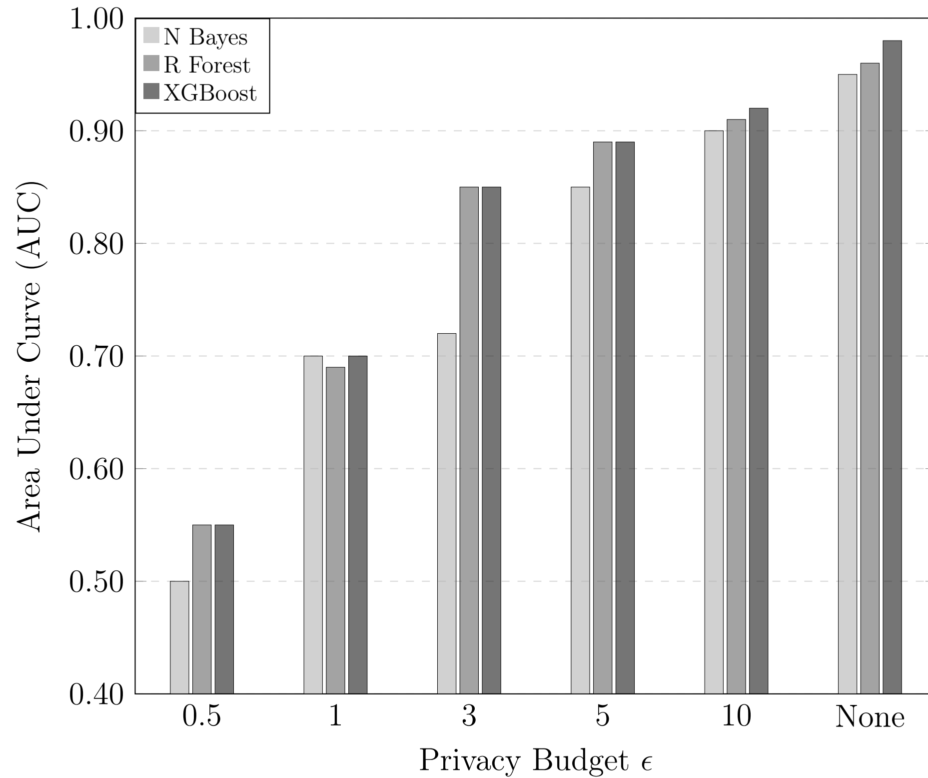


Figure 7.4: AUC results for Naive Bayes, Random Forest and XGBoost on different ϵ values ($m' = 20$) with custom historgram

Here, an AUC value of 1 is the highest meaning that the model is accurately predicting all data instances for all thresholds. On the other hand, an AUC ≤ 0.5 reveals that the model is no better than a random coin toss for any binary classification. For example, a binary classifier always outputs positive value for all data points and can never classify the negative instances. AUC metric will reveal such behavior with lower scores since the false positive rates will be higher. Therefore, AUC allows us to measure the TPR and FPR combinedly which is essential in this problem.

In Figure 7.3, we depict the relation between privacy budget, ϵ and Area Under Curve (AUC) for different methods (regular vs private min-max and Naive Bayes vs XGBoost). In Figure 7.3 a, we show the difference of regular and private min-max method, contributing to the AUC values using Naive Bayes algorithm. It shows that private min-max method provides higher AUC values for the BC-TCGA dataset. The reduced dimension is set to be $m' = 20$ for this experiment. Figure 7.3 b

depicts similar experiment replacing Naive Bayes with XGBoost trained Random Forest algorithm. It shows that that XGBoost method performs better and can deliver maximum AUC for as low ϵ value of 5.

In Figure 7.4, we compare the ML training algorithm: Naive Bayes and XGBoost for different ϵ values. Here, we use the custom histogram since its performance was better than the regular one and tested how good these algorithms performed based different privacy budget. Since we propose a generalized DP mechanism to share genomic data, arbitrary ML algorithms are better suited to test the quality of the privacy-preserving mechanism. The maximum accuracy was achieved by XGBoost of 0.92 AUC whereas the baseline accuracies (w/o privacy) were beyond 0.9 Notably, the competition results are based on Naive Bayes which is available on Table 7.4.

We also conducted an experiment on the size of the reduced dimension m' ranging from 5 to 100 that is available in Figure 7.5. We utilize a budget of $\epsilon = 5$ for both Naive Bayes and XGBoost with $m' = \{5, 10, 20, 50, 100\}$ and depict the AUC values on the y-axis. It seems that larger m's does not provide higher accuracy which is discussed in Section 7.6.

7.5.3 Execution Time

The execution time is split into two parts: a) Training and b) Testing. Notably, the privacy preserving mechanism only effects the training part which is scrutinized more carefully. In Figure 7.6, we depict the training execution time, changing the reduced dimension m' for both Naive Bayes and XGBoost . We utilized two network setting here: a) Same Machine and b) Wide Area Networking (WAN). In our first setting, both parties were setup on the same machine. This was done to check the execution time of the proposed algorithm without any network latency. On the other hand, one party was setup on a AWS Cloud (c5.xlarge instance) in us-east-1 region and the other one on Manitoba, Canada during our WAN experiment. The average latency between these machines were 38ms. We see that the network communication delays the execution due to the latency.

Table 7.3: AUC for *GSE2034* and *GSE25066* datasets using Naive Bayes and Random Forest on different parameters such as reduced dimension $m' \in \{20, 50\}$ and privacy budget $\epsilon \in \{5, 10, 15\}$. The maximum baseline AUC (w/o privacy) on GSE2034 and GSE25066 was 0.71 and 0.79 respectively.

Dataset	Method	m'	ϵ	AUC	Dataset	Method	m'	ϵ	AUC
GSE2034	NB	20	5	0.6	GSE25066	NB	20	3	0.67
			10	0.64				5	0.70
		50	5	0.58			50	3	**0.70**
			10	0.62				5	0.73
	RF	20	5	0.56		RF	20	5	0.62
			10	0.66				10	0.66
		50	5	0.53			50	5	0.53
			10	0.58				10	0.64
			15	**0.65**				15	**0.68**

7.5.4 Other Datasets

In Table 7.3, we show the results for two different breast cancer datasets: GSE 2034 and GSE 2056. Here, we intend to analyze the performance of the proposed mechanism and test its generalizability for different dataset. We see a similar trend in terms of accuracy as higher ϵ values provide higher AUCs. However, the overall AUC values are quite low compared to the competition dataset. Also, the increasing number of dimensions m's, also require more privacy budgets to attain the same level of AUC.

7.6 Discussion

The privacy problem tackled in this work was proposed in a world-wide competition in 2020 where the organizers received a total of 55 solutions from different teams. Apart from the academic research labs, there were several teams affiliated with the industry also submitted their solutions. The organizers provided a sample dataset (BC-TCGA) and set some ground rules [216]:

- All submissions must meet the differential privacy requirement under a given

privacy budget, ϵ

- Solutions will be ranked based on their performances which is measured in terms of prediction accuracy, wall-clock running time, and the communication cost between the two-parties
- All solutions will be tested with same privacy budgets, and ranked accordingly. If two solutions provide the same accuracy (within 1%), they will be further compared against their execution time and then communication cost.
- The code execution must finish between 24 hours

The organizers evaluated these submitted projects with a different train-test dataset (BC-TCGA) on a Intel Xeon E3-1280 v5 processor (4 physical cores) and 64 GB memory. The final results are published in Table 7.4 where the training and testing times are separated. Accuracy represents the percentage of the positive and negative classes detected accurately from a class-balanced test dataset. Here, training times are higher than their testing counterpart as this step generates the model in private which is later tested. Notably, the model evaluation or testing was not required to be private since the input training data while training was already private.

Table 7.4 presents the best comparison of our proposed method with the other submissions. The best performing result proposed by Carpov et al. [242], *Manticore* presents a real number and boolean arithmetic-based secure multiparty computation framework. Splitting the computation between online and offline phase, the framework performs a Principal Component Analysis (PCA) to reduce the features and finally use logistic regression to classify the gene expressions. Importantly, their work takes a different route of using secure-multiparty computations using Garbled circuits whereas we employ differentially private mechanism to share the data. Fundamentally, Carpov et al. [242] and our methods are different in terms of underlying cryptographic techniques and approach whereas both solving the same problem.

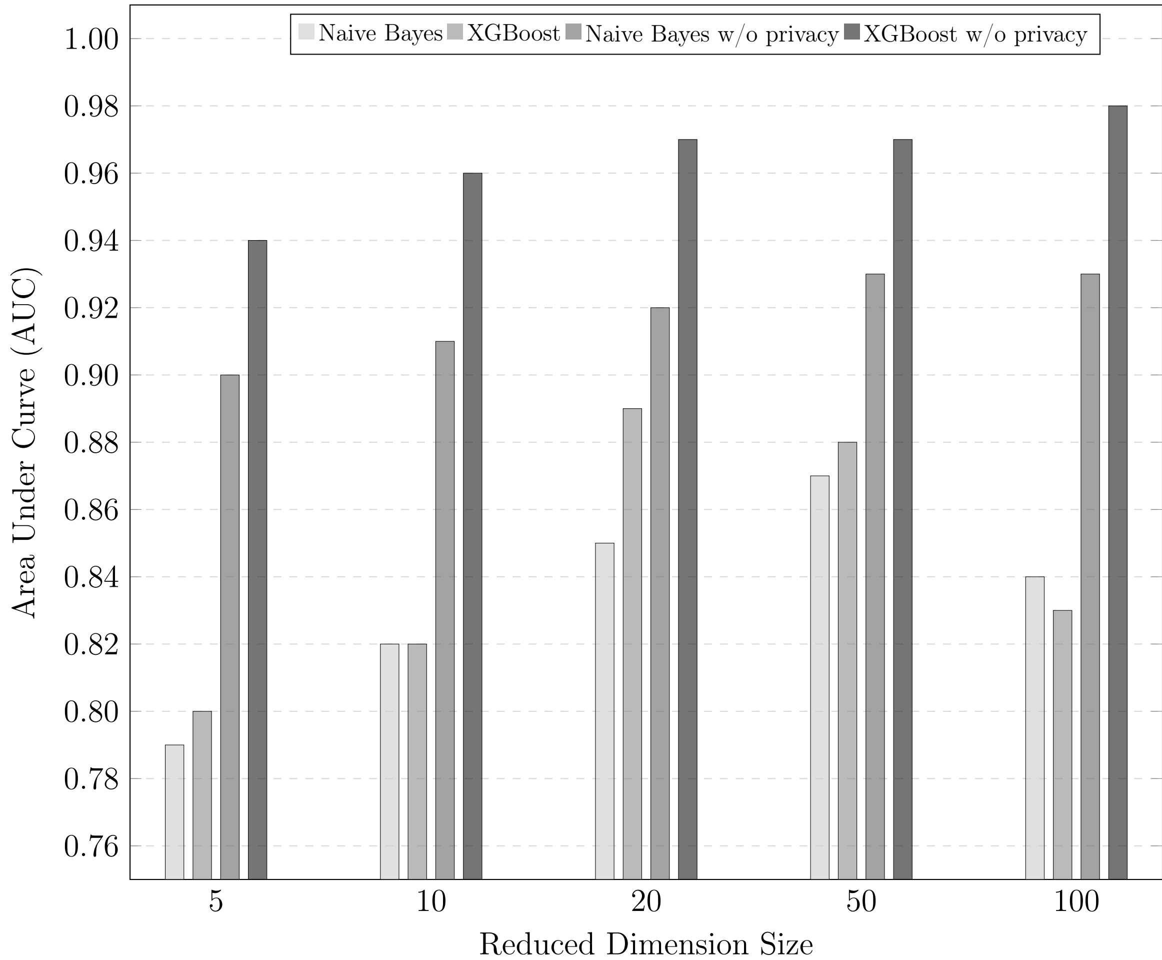


Figure 7.5: Accuracy difference for Naive Bayes and XGBoost on different dimension of data (m') while training privately with $\epsilon = 5$ or without any privacy mechanism

Figure 7.3a and 7.3b shows that the private min-max method offers better accuracy given the same privacy budget ϵ. Since the private histogram constructs the input to the federated ML, customized min-max values of the original data represents the gene expressions better. In comparison, the regular min-max method normalizes the data within a static range of [0, 100] for each party. Therefore, it does not adhere to the overall min-max value from all data sources. This adversely affects the accuracy of the underlying ML model which is clear from both Naive Bayes and XGBoost runs.

Since the output from the histogram data is categorical in nature, we see that XGBoost trained decision trees and random forest provide better AUC values

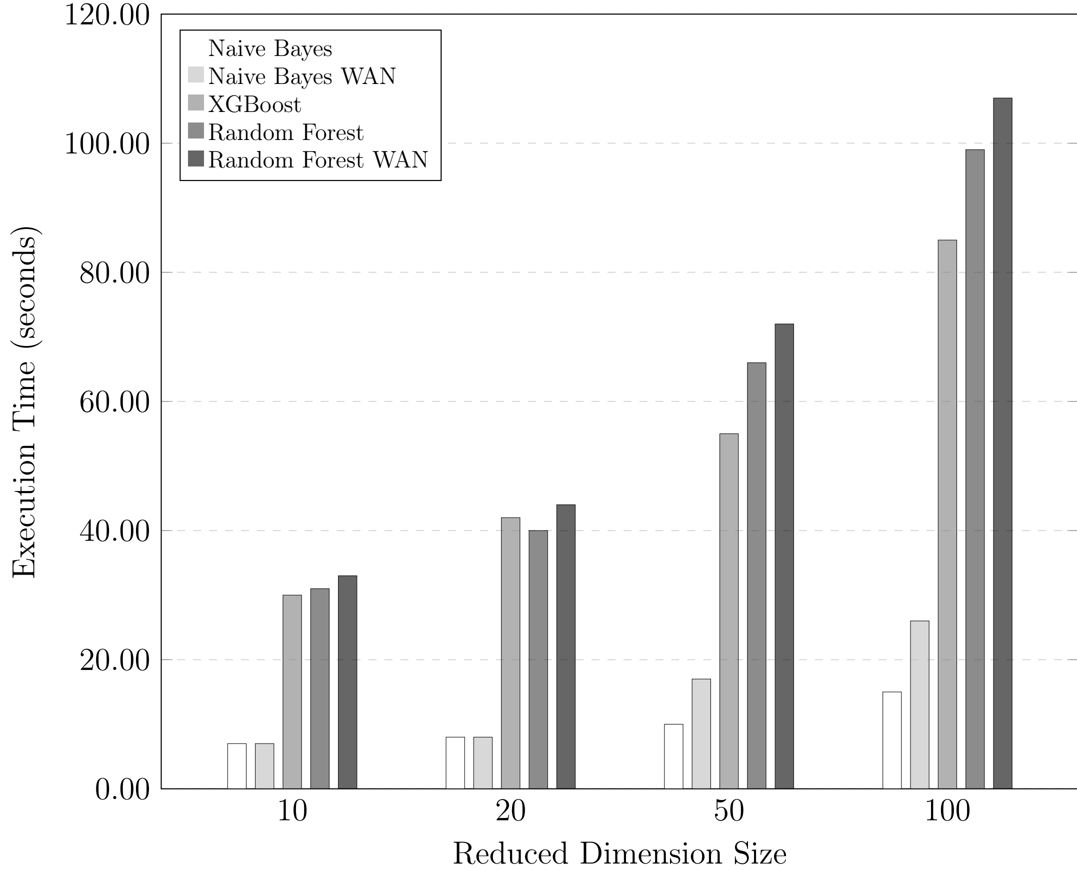


Figure 7.6: Execution time for Naive Bayes Random Forest, and XGBoost on different dimension size and LAN, WAN settings

compared to the Naive Bayes method. In Figure 7.4, we see notable AUC difference among Naive Bayes and the other methods given the same amount of privacy budget ϵ. Different budgets along with fixed a data dimension of $m' = 20$ shows that XGBoost performs better than the other ML techniques. For example, with $\epsilon = 5$, we see 0.91 AUC for XGBoost compared to 0.8 achieved with Naive Bayes. Notably, without any data privacy mechanism over the training phase, we have around 0.95, 0.96, and 0.98 AUC for all three methods respectively.

Figure 7.5 depicts the same behavior in terms of AUC regarding the data dimension, using m' values where XGBoost provides higher AUC results. We also show the non-private accuracy results on the same figure as it shows that larger data dimension leaves a positive impact on the AUC values for XGBoost whereas its the

Table 7.4: Official benchmarking results from the iDASH 2021 participating teams showing the accuracy, privacy budgets, and execution time (in seconds) on BC-TCGA dataset

Team	Affiliation	Accuracy	Running Time (sec)		DP Criteria
			Train	Test	
Manticore [242]	Industry (Inpher)	100%	0.68	0.71	$\epsilon = 3$
DP-FL	Purdue University	99.1%	1.18	0.63	$\epsilon = 3$
DSP	**UofManitoba**	**99.1%**	**10.84**	**2.67**	$\epsilon = 3$
FLR	Industry (Owkin)	99.1%	2.26	0.6	$\epsilon = 3, \delta = 1e-05$
Angle PowerFL	Industry (Tencent)	100%	36.12	1.22	$\epsilon = 3, \delta = 1e-05$
PrivCom	Industry (Baidu)	100%	104.43	2.42	$\epsilon = 3, \delta = 1e-05$
Morse	Industry (Ant Group)	94.87%	168.59	1.99	$\epsilon = 3$
SDS Research	Industry (Samsung)	89.1%	13.47	11.74	$\epsilon = 2, \delta = 1e-05$
Team GersteinLab	Yale University	85.47	243.28	5.16	$\epsilon = 1$

same (around 0.98) for $m' = \{20, 50, 100\}$ with Naive Bayes.

Figure 7.5 depicts the behavior for multiple data dimension m' as both larger and smaller values adversely affect the results. Smaller m's loose too many data points whereas larger m's consume much ϵ that adds excessive noise to the data.Therefore, $m' = 20$ performs well for $\epsilon = 5$ compared to the other m's. We also see the trend of XGBoost having a better performance than Naive Bayes in terms of accuracy similar to our earlier results.

However, given the AUC difference, we also see higher execution time for Random Forest and XGBoost algorithm since it requires more computations compared to Naive Bayes. For example, the execution time for Random Forest and Naive Bayes are 40 and 8 seconds, respectively on a federated training environment. The execution time for the WAN setup also show similar trend as the latency between the two-parties (average 38ms) slow down the training process.

Limitations: In Table 7.3, we see that our proposed mechanism is not providing the same performance on other datasets compared to the competition one, BC-TCGA. However, the baseline accuracy on these datasets (around 0.7 AUC) reveal that the maximum achievable accuracy is not too different from our proposed one. We also see that the random forest method is a bit behind in terms of AUC on *GSE25066* compared to Naive Bayes which was not the case for *BC-TCGA*. However, the number

of genomic expressions available in these datasets are too limited which did not allow us to train the model properly resulting a lower AUC value.

The competition results in Table 7.4 denote our execution time to be longer than the other participating teams. Specially our training time includes a pre-processing step (Section 7.4.4), which guarantees the privacy of the individual gene expressions. In future, we are trying to reduce them to become more comparable to the other solutions.

In this work, we worked with numerical data from gene expressions and converted them to a private histogram that can be shared for any federated learning. However, genomic data can also represent nucleotides or Single Nucleotide Polymorphisms (SNPs) containing only fixed digits. Here, we did not test our mechanism for other data types (except gene expressions) where the proposed mechanism may require some modifications. Furthermore, individual data owners may opt for different privacy budget or levels which is not explored in this work.

We also utilized variance as a method to reduce the dimensions of the dataset. This technique might not be optimal for other datasets or population specific gene expression values. Therefore, a centralized dimentionality reduction needed to be investigated. However, this can prove to be difficult as it often requires exhaustive privacy budgets for larger datasets which we deem as an important future work.

During the feature selection (in Section 7.4.4.1), we assumed that each data owner can fix m' common genes from a set of $2m'$ genes. However, there can be less than m' genes from the $2m'$ set after the differentially private set intersection operation. For example, if the gene expression variance values for both data owners are significantly different and produce a completely different set of $2m'$ genes, then we will end up with any gene for the federated learning process. Since we cannot spread a reasonable privacy budget ϵ into the thousands of genes and expect realistic accuracy, we need to formulate a better solution. There are multiple approaches that can be utilized to avoid this which we describe below.

Firstly, for an $m' = \emptyset$, we can increase the multiplying factor of $2m'$ to Cm' where C is a constant that dictates how many genes are selected for the private-set intersection. However, it increases the expense of ϵ as the primary goal of using a

smaller set (*i.e.*, $2m'$) for intersection was to reduce the usage of it in the first place. We can also perform the intersection in multiple rounds. Suppose, we have $< m'$ genes after the private intersection, we resort to spend more budget ϵ for the next round. The privacy budget can also be split unevenly where we use the larger portion on the initial round and eventually spend less on the future ones [5]. However, we did not see $< m'$ genes in all three genomic datasets and different m values while performing the experiments. Nevertheless, it is an important future research direction to find x items from multiple data owners with a differentially private guarantee [243].

7.7 Conclusion

In this chapter, we propose a privacy-preserving data sharing mechanism for a federated learning setting. Initially, the number of genes considered for the ML operation from a gene expression dataset are reduced. Later, we constructed histograms using exponential mechanism on individual data sites that was used for any further operation. We experimented with different parameters and the competition results show the efficacy of the proposed solution. In future, we would like to extend our dimensionality reduction mechanism and employ Principal Component Analysis (PCA) on a privacy-preserving federated setting. It will enhance the accuracy as we conducted some initial experiments and has shown promise on a centralize machine learning scenario. The proposed mechanism can also be utilized into different areas. Once good example can be the online user profile building where user's data is accumulated from multiple sources and privately aggregated. Different use-cases of such models are also available in the literature: clinical trial data sharing [244], graph sharing [245], and time series data release for traffic management [246]. In future, we would like to test the viability of the proposed mechanism into different areas as well.

	Data Owner 1		***Data Owner 2***
	Input:		**Input:**
1 :	Genomic Dataset		Similar Dataset
	of size $n_1 \times m$		with $n_2 \times m$
	Dimension Reduction		
2 :	Calculate the variance v_i		Follow the same steps
3 :	Rank the genes based on v_i	share $2m'$ gene list $\longleftrightarrow$	generate similar rank
4 :	Pick the top-m' genes		Pick the top-m' genes
	Regular Min-Max:		**Regular Min-Max:**
5 :	For each gene expression g_i,		Generate the min-max
	generate values:		values for local dataset
	$g_i(min)$ and $g_i(max)$	share $g_i(min), g_i(max)$ $\longleftrightarrow$	–
6 :	Normalize the gene		–
	expr. according to eqn 7.5		
	Private Min-Max:		**Private Min-Max:**
7 :	For each gene expression g_i,		Mirror the same steps to
	generate the *noisy* values:		generate noisy min and
	with ϵ', $\bar{g}_i(min)$ and $\bar{g}_i(max)$	$\bar{g}_i(min)$ and $\bar{g}_i(max)$ $\longleftrightarrow$	maximum values
8 :	Normalize the gene		–
	expr. according to eqn 7.7,		–
9 :	Generate the histogram for		Follow the same steps
	local dataset $n_1 \times m'$		and generate $n_2 \times m'$
	using Algorithm 16		–
10 :	The ML operations	share $n_j \times m'$ $\longleftrightarrow$	The final model is shared
	are performed only on these	among all data owners	
	private $n_j \times m'$ datasets		–

Figure 7.7: Summary of the differentially private data sharing methods, which is utilized on a machine learning training mechanism

Chapter 8

Conclusion

In this thesis, different mechanisms and algorithms are presented for the privacy of genomic data and any outputs generated from it. To this day, two conferences and three journal articles have been published (Chapter 2, 7, 3 and 6), whereas another journal article (Chapter 5) is currently under review. Following, we summarize the contributions of this thesis.

8.1 Summary

In Chapter 3, we proposed generalized suffix trees for genomic data, constructing them in parallel using external memory. Primarily, we investigated a novel hash-based method to outsource and execute privacy-preserving queries on the GST structure. The proposed parallel constructions and privacy-preserving queries can also be generalized for other data structures (*e.g.*, prefix trees [3], PBWT [31]) and thus can be useful for different genomic data computations.

Chapter 4 discussed privacy-preserving genome data storage and querying using a graph database. Our proposed mechanisms deem scalable compared to the previous attempts as a result of the proposed indexing scheme. Experimental results show that different numbers of SNPs from $\sim 700k$ SNPs (per person) execute within one minute showing the feasibility of our methods. However, the encryption mechanism (offline) is a major bottleneck of the scheme considering frequent database updates. This

can be replaced by the recent state-of-the-art HE mechanisms [46, 145] to improve efficiency.

In Chapter 5, we initially constructed the required circuits for FHE, which can be utilized by arbitrary complex operations. Furthermore, we explored the GPU-level parallelism for improving the execution time of the simple gate operations and arithmetic computations. We also extended those parallel operations to string operations and distance metrics. Experimental results show that the proposed method is 20× and 14.5× faster than the existing technique for computing boolean gates and multiplications, respectively.

Chapter 6 proposed an efficient technique of managing the privacy budgets using an ensemble of differentially private algorithms. We introduced online bin packing concepts to calculate the total privacy loss effectively. Our results for interactive or online settings show the efficacy of our scheme as it provides almost accurate results on Genome-Wide Association Studies. Overall, this chapter proposed a privacy-preserving data analysis framework for genomic data.

In comparison, Chapter 7 proposed a privacy-preserving data sharing mechanism for a federated machine learning setting. Initially, we reduced the data dimension or the number of genes considered for the ML operation. Later, we constructed histograms using an exponential mechanism on individual data sites that were used for any further operation. We experimented with different parameters and the competition results show the efficacy of the proposed solution.

8.2 Future work

Protecting Data Privacy One of the other prominent avenues we did not pursue in this thesis is the hardware-based security techniques, where custom hardware such as Field Programmable Gate Arrays (FPGAs) and Intel SGX processors are employed. In our previous attempts, [200, 247] we tested Intel SGX enabled processors to compute different statistics from genomic data under a strict security guarantee. The capabilities of these processors have increased in the past few years alongside the research in their real-world applications which make them a primary candidate

to extend this work.

Along the same line, our work on GPU-based FHE can be applied to more applications that rely on parallel bit computations. Since the framework cannot break the barrier of the sequential computations of additions (propagate the carry bit), we tested it on string matching problems. There are other index and search operations on genomic data as discussed in Chapter 3 and 4 using garbled circuit-based approach. Therefore, replacing the two-party trust setting with a secure FHE framework can be an important future work.

Protecting Output Privacy While in terms of data privacy, new attacks are surfacing utilizing sensitive consumer data, warranting further applications of the privacy-preserving methods. Differential privacy provides a theoretical guarantee of privacy with a tune-able budget around the mechanism. However, we did not focus on Group Privacy [248, 30] as Chapter 6 and 7 mainly consider each participant's data as private. In future, we would like to extend the work towards the group privacy setting. The privacy-preserving techniques proposed in this thesis are also not specific only to genomic data. They can be generalized towards other sensitive data such as social media [249], medical text [250, 251], and trajectory data. [252].

To conclude, security and privacy issues for any data or application cannot be solved only by privacy-preserving mechanisms or novel algorithms. There is an an urgent need to bridge the gap between the technologies and their proper enforcement using privacy laws and standard policies. Cross-disciplinary research among privacy laws and policies, computer science, and the genomic research community is much needed in this area to propose and mandate best practices.

Bibliography

[1] Md Momin Al Aziz, Md Nazmus Sadat, Dima Alhadidi, Shuang Wang, Xiaoqian ang, Cheryl L Brown, and Noman Mohammed. Privacy-preserving techniques of genomic data—a survey. *Briefings in bioinformatics*, 20(3):887–895, 2019.

[2] Md Momin Al Aziz, Parimala Thulasiraman, and Noman Mohammed. Parallel generalized suffix tree construction for genomic data. In *International Conference on Algorithms for Computational Biology*, pages 3–15. Springer, 2020.

[3] Luyao Chen, Md Momin Aziz, Noman Mohammed, and Xiaoqian Jiang. Secure large-scale genome data storage and query. *Computer methods and programs in biomedicine*, 165:129–137, 2018.

[4] T. Morshed, M. Aziz, and N. Mohammed. Cpu and gpu accelerated fully homomorphic encryption. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 142–153, Los Alamitos, CA, USA, dec 2020. IEEE Computer Society.

[5] Md Momin Al Aziz, Shain Kamali, Xiaoqian Jiang, and Noman Mohammed. Online algorithm for differentially private genome-wide association studies. *ACM Transaction on Computing for Healthcare (to appear)*, 2020.

[6] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster packed homomorphic operations and efficient circuit bootstrapping for tfhe. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 377–408. Springer, 2017.

[7] Sean Simmons, Cenk Sahinalp, and Bonnie Berger. Enabling privacy-preserving GWASs in heterogeneous human populations. *Cell Systems*, 3(1):54–61, 2016.

[8] Meng Wang, Zhanglong Ji, Shuang Wang, Jihoon Kim, Hai Yang, Xiaoqian Jiang, and Lucila Ohno-Machado. Mechanisms to protect the privacy of families when using the transmission disequilibrium test in genome-wide association studies. *Bioinformatics*, 33(23):3716–3725, 2017.

[9] DNA Sequencing Costs. `http://www.genome.gov/sequencingcosts/`. Accessed: 2016-01-11.

[10] Barry Barnes and John Dupré. *Genomes and what to make of them*. University of Chicago Press, 2009.

[11] Susan Brown Trinidad, Stephanie M Fullerton, Julie M Bares, Gail P Jarvik, Eric B Larson, and Wylie Burke. Genomic research and wide data sharing: views of prospective participants. *Genetics in Medicine*, 12(8):486–495, 2010.

[12] Bradley Malin and Latanya Sweeney. Determining the identifiability of dna database entries. In *Proceedings of the AMIA Symposium*, page 537. American Medical Informatics Association, 2000.

[13] Scott Gottlieb. Us employer agrees to stop genetic testing. *BMJ: British Medical Journal*, 322(7284):449, 2001.

[14] Zhen Lin, Art B Owen, and Russ B Altman. Genomic research and human subject privacy. *SCIENCE-NEW YORK THEN WASHINGTON-.*, pages 183–183, 2004.

[15] Nils Homer, Szabolcs Szelinger, Margot Redman, David Duggan, Waibhav Tembe, Jill Muehling, John V Pearson, Dietrich A Stephan, Stanley F Nelson, and David W Craig. Resolving individuals contributing trace amounts of dna to highly complex mixtures using high-density snp genotyping microarrays. *PLoS Genet*, 4(8):e1000167, 2008.

[16] Michael T Goodrich. The mastermind attack on genomic data. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 204–218. IEEE, 2009.

[17] Mathias Humbert, Erman Ayday, Jean-Pierre Hubaux, and Amalio Telenti. Addressing the concerns of the lacks family: quantification of kin genomic privacy. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1141–1152. ACM, 2013.

[18] Latanya Sweeney, Akua Abu, and Julia Winn. Identifying participants in the personal genome project by name. 2013.

[19] Melissa Gymrek, Amy L McGuire, David Golan, Eran Halperin, and Yaniv Erlich. Identifying personal genomes by surname inference. *Science*, 339(6117):321–324, 2013.

[20] Matthew Fredrikson, Eric Lantz, Somesh Jha, Simon Lin, David Page, and Thomas Ristenpart. Privacy in pharmacogenetics: An end-to-end case study of personalized warfarin dosing. In *In Proceedings of the 23rd USENIX Security Symposium (USENIX Security 14)*, pages 17–32, 2014.

[21] Suyash S Shringarpure and Carlos D Bustamante. Privacy risks from genomic data-sharing beacons. *The American Journal of Human Genetics*, 97(5):631–646, 2015.

[22] Jean Louis Raisaro, Florian Tramer, Zhanglong Ji, Diyue Bu, Yongan Zhao, Knox Carey, David Lloyd, Heidi Sofia, Dixie Baker, Paul Flicek, et al. Addressing beacon re-identification attacks: Quantification and mitigation of privacy risks. Technical report, 2016.

[23] Arif Harmanci and Mark Gerstein. Quantification of private information leakage from phenotype-genotype data: linking attacks. *Nature methods*, 13(3):251–256, 2016.

[24] Zhicong Huang, Erman Ayday, Huang Lin, Raeka S Aiyar, Adam Molyneaux, Zhenyu Xu, Jacques Fellay, Lars M Steinmetz, and Jean-Pierre Hubaux. A

privacy-preserving solution for compressed storage and selective retrieval of genomic data. *Genome Research*, 26(12):1687–1696, 2016.

[25] U.s. department of health and human services office for civil rights. `https://ocrportal.hhs.gov/ocr/breach/breach_report.jsf`. Accessed: 2021-11-16.

[26] Muhammad Naveed, Erman Ayday, Ellen W Clayton, Jacques Fellay, Carl A Gunter, Jean-Pierre Hubaux, Bradley A Malin, and XiaoFeng Wang. Privacy in the genomic era. *ACM Computing Surveys (CSUR)*, 48(1):6, 2015.

[27] Md Momin Al Aziz, Md Nazmus Sadat, Dima Alhadidi, Shuang Wang, Xiaoqian Jiang, Cheryl L Brown, and Noman Mohammed. Privacy-preserving techniques of genomic data: a survey. *Briefings in bioinformatics*, 20(3):887–895, 2017.

[28] Mete Akgün, A Osman Bayrak, Bugra Ozer, and M Şamil Sağıroğlu. Privacy preserving processing of genomic data: A survey. *Journal of biomedical informatics*, 56:103–111, 2015.

[29] Jean Louis Raisaro, Gwangbae Choi, Sylvain Pradervand, Raphael Colsenet, Nathalie Jacquemont, Nicolas Rosat, Vincent Mooser, and Jean-Pierre Hubaux. Protecting privacy and security of genomic data in i2b2. Technical report, Institute of Electrical and Electronics Engineers, 2017.

[30] Cynthia Dwork. Differential privacy. In *Proceedings of the 33rd International Conference on Automata, Languages and Programming - Volume Part II*, ICALP'06, pages 1–12, 2006.

[31] Kana Shimizu, Koji Nuida, and Gunnar Rätsch. Efficient privacy-preserving string search and an application in genomics. *Bioinformatics*, 32(11):1652–1661, 2016.

[32] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *Advances in Cryptology–ASIACRYPT 2016: 22nd International Conference*

on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I 22, pages 3–33. Springer, 2016.

[33] Joan Boyar, Shahin Kamali, Kim S Larsen, and Alejandro López-Ortiz. Online bin packing with advice. *Algorithmica*, 74(1):507–527, 2016.

[34] Oded Goldreich. *Foundations of cryptography: volume 2, basic applications*. Cambridge university press, 2009.

[35] Long Cheng, Fang Liu, and Danfeng Yao. Enterprise data breach: causes, challenges, prevention, and future directions. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 7(5):e1211, 2017.

[36] U.S. Department of Health and Human Services Office for Civil Rights. Breach portal: Notice to the secretary of hhs breach of unsecured protected health information.

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

[38] Craig Gentry et al. Fully homomorphic encryption using ideal lattices. In *STOC*, volume 9, pages 169–178, 2009.

[39] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 409–437. Springer, 2017.

[40] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptology ePrint Archive*, 2012:144, 2012.

[41] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *International Conference on the Theory and Application of Cryptographic Techniques (EUROCRYPT)*, pages 223–238, 1999.

[42] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *Advances in Cryptology–CRYPTO 2013*, pages 75–92. Springer, 2013.

[43] Léo Ducas and Daniele Micciancio. Fhew: Bootstrapping homomorphic encryption in less than a second. *EUROCRYPT (1)*, 9056:617–640, 2015.

[44] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Tfhe: Fast fully homomorphic encryption over the torus. *Journal of Cryptology*, pages 1–58, 2018.

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

[46] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptology ePrint Archive*, 2012:144, 2012.

[47] Wei Dai and Berk Sunar. cuhe: A homomorphic encryption accelerator library. In *International Conference on Cryptography and Information Security in the Balkans*, pages 169–186. Springer, 2015.

[48] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[49] Pascal Paillier et al. Public-key cryptosystems based on composite degree residuosity classes. In *Eurocrypt*, volume 99, pages 223–238. Springer, 1999.

[50] Chris Lomont. Introduction to intel advanced vector extensions. *Intel White Paper*, pages 1–21, 2011.

[51] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. Bootstrapping for approximate homomorphic encryption. In *Annual*

International Conference on the Theory and Applications of Cryptographic Techniques, pages 360–384. Springer, 2018.

[52] Microsoft SEAL (release 3.2). `https://github.com/Microsoft/SEAL`, February 2019. Microsoft Research, Redmond, WA.

[53] CUDA-accelerated Fully Homomorphic Encryption Library, September 2019.

[54] NuFHE, a GPU-powered Torus FHE implementation, September 2019.

[55] Cingulata, September 2019.

[56] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):34, 2009.

[57] Miran Kim, Arif Ozgun Harmanci, Jean-Philippe Bossuat, Sergiu Carpov, Jung Hee Cheon, Ilaria Chillotti, Wonhee Cho, David Froelicher, Nicolas Gama, Mariya Georgieva, et al. Ultrafast homomorphic encryption models enable secure outsourcing of genotype imputation. *Cell Systems*, 12(11):1108–1120, 2021.

[58] Md Nazmus Sadat, Al Aziz, Md Momin, Noman Mohammed, Feng Chen, Xiaoqian Jiang, and Shuang Wang. Safety: secure gwas in federated environment through a hybrid solution. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 16(1):93–102, 2019.

[59] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In *In Proceedings of the 27th USENIX Security Symposium (USENIX Security 18)*, pages 1651–1669, 2018.

[60] Ehsan Hesamifard, Hassan Takabi, Mehdi Ghasemi, and Rebecca N Wright. Privacy-preserving machine learning as a service. *Proceedings on Privacy Enhancing Technologies*, 2018(3):123–142, 2018.

[61] Andrew Chi-Chih Yao. Protocols for secure computations. In *FOCS*, volume 82, pages 160–164, 1982.

[62] Moni Naor and Benny Pinkas. Efficient oblivious transfer protocols. In *SODA*, volume 1, pages 448–457, 2001.

[63] Mihir Bellare and Silvio Micali. Non-interactive oblivious transfer and applications. In *Conference on the Theory and Application of Cryptology*, pages 547–557. Springer, 1989.

[64] Tung Chou and Claudio Orlandi. The simplest protocol for oblivious transfer. In *International Conference on Cryptology and Information Security in Latin America*, pages 40–58. Springer, 2015.

[65] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 535–548, 2013.

[66] Yehuda Lindell and Benny Pinkas. A proof of security of yao's protocol for two-party computation. *Journal of Cryptology*, 22(2):161–188, 2009.

[67] Xiao Wang, Hubert Chan, and Elaine Shi. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 850–861. ACM, 2015.

[68] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *In Proceedings of the 20th USENIX Security Symposium*, volume 201, 2011.

[69] Ebrahim M Songhori, Siam U Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. Tinygarble: Highly compressed and scalable sequential garbled circuits. In *2015 IEEE Symposium on Security and Privacy*, pages 411–428, May 2015.

[70] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. `https://github.com/emp-toolkit`, 2016.

[71] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 19–38. IEEE, 2017.

[72] M Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 707–721. ACM, 2018.

[73] Bita Darvish Rouhani, M Sadegh Riazi, and Farinaz Koushanfar. Deepsecure: Scalable provably-secure deep learning. In *In Proceedings of the 55th Annual Design Automation Conference*, page 2. ACM, 2018.

[74] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of cryptography*, pages 265–284. Springer, 2006.

[75] J. Hsu, M. Gaboardi, A. Haeberlen, S. Khanna, A. Narayan, B. C. Pierce, and A. Roth. Differential privacy: An economic method for choosing epsilon. In *2014 IEEE 27th Computer Security Foundations Symposium*, pages 398–410, 2014.

[76] Cynthia Dwork. A firm foundation for private data analysis. *Communications of the ACM*, 54(1):86–95, 2011.

[77] Peter Kairouz, Sewoong Oh, and Pramod Viswanath. The composition theorem for differential privacy. *IEEE Transactions on Information Theory*, 63(6):4037–4049, 2017.

[78] Frank D McSherry. Privacy integrated queries: an extensible platform for

privacy-preserving data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 19–30. ACM, 2009.

[79] Frank McSherry and Kunal Talwar. Mechanism design via differential privacy. In *Foundations of Computer Science, 2007. FOCS'07. 48th Annual IEEE Symposium on*, pages 94–103. IEEE, 2007.

[80] Mathias Humbert, Erman Ayday, Jean-Pierre Hubaux, and Amalio Telenti. Quantifying interdependent risks in genomic privacy. *ACM Transaction on Privacy Security*, 20(1):3:1–3:31, 2017.

[81] Ethan Cerami, Jianjiong Gao, Ugur Dogrusoz, Benjamin E Gross, Selcuk Onur Sumer, Bülent Arman Aksoy, Anders Jacobsen, Caitlin J Byrne, Michael L Heuer, Erik Larsson, et al. The cbio cancer genomics portal: an open platform for exploring multidimensional cancer genomics data, 2012.

[82] Paul Bieganski, John Riedl, John V Carlis, and Ernest F Retzel. Generalized suffix trees for biological sequence data: Applications and implementation. In *HICSS (5)*, pages 35–44, 1994.

[83] Ramesh Hariharan. Optimal parallel suffix tree construction. *Journal of Computer and System Sciences*, 55(1):44–69, 1997.

[84] Martin Farach, Paolo Ferragina, and Shanmugavelayutham Muthukrishnan. Overcoming the memory bottleneck in suffix tree construction. In *Proceedings 39th FOCS*, pages 174–183. IEEE, 1998.

[85] Alan Hodgkinson and Adam Eyre-Walker. Human triallelic sites: evidence for a new mutational mechanism? *Genetics*, 184(1):233–241, 2010.

[86] Donald E Knuth. Sorting and searching. 1973.

[87] Dan Gusfield. Algorithms on stings, trees, and sequences: Computer science and computational biology. *Acm Sigact News*, 28(4):41–60, 1997.

[88] E. Ukkonen. Online construction of suffixtrees. *Algorithmica*, 14(3):249–260, 1995.

[89] Ian Foster. *Designing and building parallel programs: concepts and tools for parallel software engineering*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[90] Ralph C Merkle. Method of providing digital signatures, January 5 1982. US Patent 4,309,569.

[91] Computing Resources. `www.cs.umanitoba.ca/computing`. Accessed: 2019-12-4.

[92] Md Momin Al Aziz. Implementation for Parallel Private GST. https://github.com/mominbuet/ParallelGST. Accessed 2022-03-25.

[93] Toufique Morshed, Dima Alhadidi, and Noman Mohammed. Parallel linear regression on encrypted data. In *In Proceedings of 16th Annual Conference on Privacy, Security and Trust (PST)*, pages 1–5. IEEE, 2018.

[94] Shuang Wang, Noman Mohammed, and Rui Chen. Differentially private genome data dissemination through top-down specialization. *BMC medical informatics and decision making*, 14(Suppl 1):S2, 2014.

[95] Rui Chen, Gergely Acs, and Claude Castelluccia. Differentially private sequential data publication via variable-length n-grams. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 638–649, 2012.

[96] Mikhail J Atallah, Florian Kerschbaum, and Wenliang Du. Secure and private sequence comparisons. In *Proceedings of the 2003 ACM workshop on Privacy in the electronic society*, pages 39–44. ACM, 2003.

[97] Marina Blanton and Mehrdad Aliasgari. Secure outsourcing of DNA searching via finite automata. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 49–64. Springer, 2010.

[98] Juan Ramón Troncoso-Pastoriza, Stefan Katzenbeisser, and Mehmet Celik. Privacy preserving error resilient DNA searching through oblivious automata. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 519–528. ACM, 2007.

[99] Hiroki Sudo, Masanobu Jimbo, Koji Nuida, and Kana Shimizu. Secure Wavelet Matrix: Alphabet-Friendly Privacy-Preserving String Search. *BioRxiv*, page 085647, 2016.

[100] Md Safiur Rahman Mahdi, Mohammad Zahidul Hasan, and Noman Mohammed. Secure sequence similarity search on encrypted genomic data. In *In Proceedings of 2017 IEEE/ACM International Conference on Connected Health: Applications, Systems and Engineering Technologies (CHASE)*, pages 205–213. IEEE, 2017.

[101] Md Safiur Rahman Mahdi, Md Momin Al Aziz, Dima Alhadidi, and Noman Mohammed. Secure similar patients query on encrypted genomic data. *IEEE journal of biomedical and health informatics*, 23(6):2611–2618, 2018.

[102] Yu Ishimaki, Hiroki Imabayashi, Kana Shimizu, and Hayato Yamana. Privacy-preserving string search for genome sequences with FHE bootstrapping optimization. In *In Proceeding of the 2016 IEEE International Conference on Big Data (Big Data)*, pages 3989–3991. IEEE, 2016.

[103] Richard Durbin. Efficient haplotype matching and storage using the positional burrows–wheeler transform (pbwt). *Bioinformatics*, 30(9):1266–1272, 2014.

[104] Y. Yamada, K. Rohloff, and M. Oguchi. Homomorphic encryption for privacy-preserving genome sequences search. In *In Proceedings of 2019 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 7–12, 2019.

[105] Tsung-Ting Kuo, Xiaoqian Jiang, Haixu Tang, XiaoFeng Wang, Tyler Bath, Diyue Bu, Lei Wang, Arif Harmanci, Shaojie Zhang, Degui Zhi, et al. idash

secure genome analysis competition 2018: blockchain genomic data access logging, homomorphic encryption on gwas, and dna segment searching. *BMC Medical Genomics*, 13(Suppl 7), 2020.

[106] Katerina Sotiraki, Esha Ghosh, and Hao Chen. Privately computing set-maximal matches in genomic data. *BMC Medical Genomics*, 13(7):1–8, 2020.

[107] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game, or a completeness theorem for protocols with honest majority. In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, pages 307–328. 2019.

[108] Md Safiur Rahman Mahdi, Md Momin Al Aziz, Noman Mohammed, and Xiaoqian Jiang. Privacy-preserving string search on encrypted genomic data using a generalized suffix tree. *Informatics in Medicine Unlocked*, 23:100525, 2021.

[109] Sandeep Tata, Richard A Hankins, and Jignesh M Patel. Practical suffix tree construction. In *Proceedings of the 13th intl. conf. VLDB*, pages 36–47, 2004.

[110] Yuanyuan Tian, Sandeep Tata, Richard A Hankins, and Jignesh M Patel. Practical methods for constructing suffix trees. *The VLDB Journal*, 14(3):281–299, 2005.

[111] Benjarath Phoophakdee and Mohammed J Zaki. Genome-scale disk-based suffix tree indexing. In *SIGMOD int. conf. on Management of data*, pages 833–844. ACM, 2007.

[112] Amol Ghoting and Konstantin Makarychev. Serial and parallel methods for i/o efficient suffix tree construction. In *Proceedings of the 2009 ACM SIGMOD Int'l Conference on Management of data*, pages 827–840. ACM, 2009.

[113] Essam Mansour, Amin Allam, Spiros Skiadopoulos, and Panos Kalnis. ERA: efficient serial and parallel suffix tree construction for very long strings. *Proceedings of the VLDB*, 5(1):49–60, 2011.

[114] Matteo Comin and Montse Farreras. Parallel continuous flow: a parallel suffix tree construction tool for whole genomes. *Jrnl. of Comp. Biology*, 21(4):330–344, 2014.

[115] Guanghui et al. Zhu. DGST: Efficient and scalable suffix tree construction on distributed data-parallel platforms. *Parallel Computing*, 87:87–102, 2019.

[116] Julian Shun and Guy E Blelloch. A simple parallel cartesian tree algorithm and its application to parallel suffix tree construction. *ACM TOPC*, 1(1):8, 2014.

[117] Patrick Flick and Srinivas Aluru. Parallel construction of suffix trees and the all-nearest-smaller-values problem. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 12–21. IEEE, 2017.

[118] Freeson Kaniwa, Otlhapile Dinakenyane, and Venu Madhav Kuthadi. Parallel algorithm for indexing large dna sequences using mapreduce on hadoop. In *2017 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 1576–1582. IEEE, 2017.

[119] Yunhao Li, Jiahui Jin, Runqun Xiong, and Junzhou Luo. A distributed approach for constructing generalized suffix tree on spark by using optimized elastic range algorithm. In *2017 Fifth International Conference on Advanced Cloud and Big Data (CBD)*, pages 117–122. IEEE, 2017.

[120] Marko J Mišić etal. Parallelization of gst algorithm for source code similarity detection. In *24th TELFOR*, pages 1–4. IEEE, 2016.

[121] Charles E Cook, Mary Todd Bergman, Robert D Finn, Guy Cochrane, Ewan Birney, and Rolf Apweiler. The european bioinformatics institute in 2016: Data growth and integration. *Nucleic Acids Res.*, 44(D1):D20–6, January 2016.

[122] Microsoft. Microsoft Azure Machine Learning. Accessed 21 May, 2019.

[123] Cloud storage pricing. `https://aws.amazon.com/s3/pricing/`. Accessed: 2017-12-13.

[124] Zhen Lin, Art B Owen, and Russ B Altman. Genomic research and human subject privacy. *Science*, 305(5681):183, July 2004.

[125] Melissa Gymrek, Amy L McGuire, David Golan, Eran Halperin, and Yaniv Erlich. Identifying personal genomes by surname inference. *Science*, 339(6117):321–324, January 2013.

[126] Peter Claes, Denise K Liberton, Katleen Daniels, Kerri Matthes Rosana, Ellen E Quillen, Laurel N Pearson, Brian McEvoy, Marc Bauchet, Arslan A Zaidi, Wei Yao, and Others. Modeling 3D facial shape from DNA. *PLoS Genet.*, 10(3):e1004224, 2014.

[127] Christoph Lippert, Riccardo Sabatini, M Cyrus Maher, Eun Yong Kang, Seunghak Lee, Okan Arikan, Alena Harley, Axel Bernal, Peter Garst, Victor Lavrenko, Ken Yocum, Theodore Wong, Mingfu Zhu, Wen-Yun Yang, Chris Chang, Tim Lu, Charlie W H Lee, Barry Hicks, Smriti Ramakrishnan, Haibao Tang, Chao Xie, Jason Piper, Suzanne Brewerton, Yaron Turpaz, Amalio Telenti, Rhonda K Roby, Franz J Och, and J Craig Venter. Identification of individuals by trait prediction using whole-genome sequencing data. *Proceedings of the National Academy of Sciences*, 114(38):10166–10171, September 2017.

[128] The privacy conundrum and genomic research: Re-Identification and other concerns. `http://www.healthaffairs.org/do/10.1377/hblog20130911.034137/full/`. Accessed: 2017-11-13.

[129] 14 MEEELLION verizon subscribers' details leak from crappily configured AWS S3 data store. `https://www.theregister.co.uk/2017/07/12/14m_verizon_customers_details_out/`. Accessed: 2017-11-13.

[130] Ebtesam A Alomari Muhammad. A survey of security issues for data sharing over untrusted cloud. *Journal of Emerging Trends in Computing and Information Sciences*, 5(8):609–619, 2014.

[131] D Liu. Efficient processing of encrypted data in Honest-but-Curious clouds. In

2016 IEEE 9th International Conference on Cloud Computing (CLOUD), pages 970–974, June 2016.

[132] Zahidul Hasan, Md Safiur Rahman Mahdi, and Noman Mohammed. Secure count query on encrypted genomic data: A survey. *IEEE Internet Computing*, 2018.

[133] Mohammad Zahidul Hasan, Md Safiur Rahman Mahdi, Md Nazmus Sadat, and Noman Mohammed. Secure count query on encrypted genomic data. *Journal of biomedical informatics*, 81:41–52, 2018.

[134] Nichola Johnson, Olivia Fletcher, Claire Palles, Matthew Rudd, Emily Webb, Gabrielle Sellick, Isabel dos Santos Silva, Valerie McCormack, Lorna Gibson, Agnes Fraser, et al. Counting potentially functional variants in brca1, brca2 and atm predicts breast cancer susceptibility. *Human molecular genetics*, 16(9):1051–1057, 2007.

[135] Justin J Miller. Graph database applications and concepts with neo4j. In *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA*, volume 2324, page 36, 2013.

[136] Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1247–1250. AcM, 2008.

[137] Md Momin Al Aziz, Mohammad Z. Hasan, Noman Mohammed, and Dima Alhadidi. Secure and efficient multiparty computation on genomic data. In *Proceedings of the 20th International Database Engineering & Applications Symposium*, pages 278–283. ACM, 2016.

[138] Alfred V Aho and John E Hopcroft. *The design and analysis of computer algorithms*. Pearson Education India, 1974.

[139] Adrienne Kitts and Stephen Sherry. The single nucleotide polymorphism database (dbsnp) of nucleotide sequence variation. *The NCBI Handbook. McEntyre J, Ostell J, eds. Bethesda, MD: US National Center for Biotechnology Information*, 2002.

[140] G M Church. The personal genome project. *Molecular Systems Biology*, 1(1), 2005.

[141] Matteo Fumagalli, Filipe G Vieira, Thorfinn Sand Korneliussen, Tyler Linderoth, Emilia Huerta-Sánchez, Anders Albrechtsen, and Rasmus Nielsen. Quantifying population genetic differentiation from next-generation sequencing data. *Genetics*, 195(3):979–992, 2013.

[142] Craig Gentry, Kenny A Goldman, Shai Halevi, Charanjit Julta, Mariana Raykova, and Daniel Wichs. Optimizing oram and using it efficiently for secure computation. In *International Symposium on Privacy Enhancing Technologies Symposium*, pages 1–18. Springer, 2013.

[143] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: An extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 299–310. ACM, 2013.

[144] Zachary D Stephens, Skylar Y Lee, Faraz Faghri, Roy H Campbell, Chengxiang Zhai, Miles J Efron, Ravishankar Iyer, Michael C Schatz, Saurabh Sinha, and Gene E Robinson. Big data: astronomical or genomical? *PLoS biology*, 13(7):e1002195, 2015.

[145] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Improving tfhe: faster packed homomorphic operations and efficient circuit bootstrapping. Technical report, IACR Cryptology ePrint Archive 2017, 430, 2017.

[146] Benny Pinkas. Cryptographic techniques for privacy-preserving data mining. *SIGKDD Explor. Newsl.*, 4(2):12–19, December 2002.

[147] R. Ghasemi, M. M. Al Aziz, N. Mohammed, M. H. Dehkordi, and X. Jiang. Private and efficient query processing on outsourced genomic databases. *IEEE Journal of Biomedical and Health Informatics*, 21(5):1466–1472, Sept 2017.

[148] Mustafa Canim, Murat Kantarcioglu, and Bradley Malin. Secure management of biomedical data with cryptographic hardware. *Information Technology in Biomedicine, IEEE Transactions on*, 16(1):166–175, 2012.

[149] Murat Kantarcioglu, Wei Jiang, Ying Liu, and Bradley Malin. A cryptographic approach to securely share and query genomic sequences. *IEEE Transactions on information technology in biomedicine*, 12(5):606–617, 2008.

[150] Anh Pham, Italo Dacosta, Guillaume Endignoux, Juan Ramon Troncoso Pastoriza, Kevin Huguenin, and Jean-Pierre Hubaux. Oride: A privacy-preserving yet accountable ride-hailing service. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1235–1252, Vancouver, BC, 2017.

[151] Miran Kim, Yongsoo Song, and Jung Hee Cheon. Secure searching of biomarkers through hybrid homomorphic encryption scheme. *BMC medical genomics*, 10(2):42, 2017.

[152] Hao Chen, Ran Gilad-Bachrach, Kyoohyung Han, Zhicong Huang, Amir Jalali, Kim Laine, and Kristin Lauter. Logistic regression over encrypted data from fully homomorphic encryption. *BMC medical genomics*, 11(4):81, 2018.

[153] 23AndMe.com. Our health + ancestry dna service - 23andme canada. `https://www.23andme.com/en-ca/dna-health-ancestry`. Accessed: 2020-11-20.

[154] X. Sh. 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*, CCS '15, pages 492–503. ACM, 2015.

[155] Md Momin Al Aziz, Dima Alhadidi, and Noman Mohammed. Secure

approximation of edit distance on genomic data. *BMC Medical Genomics*, 10(2):41, 2017.

[156] Christi J Guerrini, Jill O Robinson, Devan Petersen, and Amy L McGuire. Should police have access to genetic genealogy databases? capturing the golden state killer and other criminals using a controversial new forensic technique. *PLoS biology*, 16(10):e2006906, 2018.

[157] Jung Hee Cheon, Miran Kim, and Kristin Lauter. Homomorphic computation of edit distance. In *International Conference on Financial Cryptography and Data Security*, pages 194–212. Springer, 2015.

[158] Christina Boura, Nicolas Gama, and Mariya Georgieva. Chimera: a unified framework for b/fv, tfhe and heaan fully homomorphic encryption and predictions for deep learning. Technical report, Cryptology ePrint Archive, Report 2018/758, 2018.

[159] Matteo Frigo and Steven G Johnson. Fftw: An adaptive software architecture for the fft. In *In Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP'98 (Cat. No. 98CH36181)*, volume 3, pages 1381–1384. IEEE, 1998.

[160] Zvika Brakerski, Craig Gentry, and Shai Halevi. Packed ciphertexts in lwe-based homomorphic encryption. In *International Workshop on Public Key Cryptography*, pages 1–13. Springer, 2013.

[161] Léo Ducas and Daniele Micciancio. Fhew: Bootstrapping homomorphic encryption in less than a second. Cryptology ePrint Archive, Report 2014/816, 2014. `https://eprint.iacr.org/2014/816`.

[162] Catherine C. McGeoch. Parallel addition. *The American Mathematical Monthly*, 100(9):867–871, 1993.

[163] Anatolii Alekseevich Karatsuba and Yu P Ofman. Multiplication of many-digital numbers by automatic computers. In *Doklady Akademii Nauk*, volume 145, pages 293–294. Russian Academy of Sciences, 1962.

[164] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[165] NVIDIA. Geforce gtx 1080 graphics cards from nvidia geforce.

[166] James W Fickett. Fast optimal alignment. *Nucleic acids research*, 12(1Part1):175–179, 1984.

[167] Kana Shimizu, Koji Nuida, and Gunnar Rätsch. Efficient privacy-preserving string search and an application in genomics. *Bioinformatics, 32(11): 1652–1661*, 2016.

[168] Pengtao Xie, Misha Bilenko, Tom Finley, Ran Gilad-Bachrach, Kristin Lauter, and Michael Naehrig. Crypto-nets: Neural networks over encrypted data. *arXiv preprint arXiv:1412.6181*, 2014.

[169] Hassan Takabi, Ehsan Hesamifard, and Mehdi Ghasemi. Privacy preserving multi-party machine learning with homomorphic encryption. In *29th Annual Conference on Neural Information Processing Systems (NIPS)*, 2016.

[170] Ilaria Chillotti Snowman Nicolas Gama, SC. Tfhe: Fast fully homomorphic encryption library over the torus. `https://github.com/tfhe/tfhe/tree/v1.0.1`, 2017.

[171] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. *SIAM Journal on Computing*, 43(2):831–871, 2014.

[172] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):13, 2014.

[173] Michael J Flynn. Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9):948–960, 1972.

[174] Shai Halevi and Victor Shoup. Algorithms in helib. In *Annual Cryptology Conference*, pages 554–571. Springer, 2014.

[175] Yarkın Doröz, Erdinç Öztürk, Erkay Savaş, and Berk Sunar. Accelerating ltv based homomorphic encryption in reconfigurable hardware. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 185–204. Springer, 2015.

[176] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *In Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pages 1219–1234. ACM, 2012.

[177] Wei Dai, Yarkın Doröz, and Berk Sunar. Accelerating swhe based pirs using gpus. In *International Conference on Financial Cryptography and Data Security*, pages 160–171. Springer, 2015.

[178] Xinya Lei, Ruixin Guo, Feng Zhang, Lizhe Wang, Rui Xu, and Guangzhi Qu. Accelerating homomorphic full adder based on fhew using multicore cpu and gpus. *IEEE Transactions on Industrial Informatics*, 2019.

[179] Hai-bin Yang, Wu-jun Yao, Wen-chao Liu, and Bin Wei. Efficiency analysis of tfhe fully homomorphic encryption software library based on gpu. In *Workshops of the International Conference on Advanced Information Networking and Applications*, pages 93–102. Springer, 2019.

[180] Tanping Zhou, Xiaoyuan Yang, Longfei Liu, Wei Zhang, and Ningbo Li. Faster bootstrapping with multiple addends. *IEEE Access*, 6:49868–49876, 2018.

[181] Somesh Jha, Louis Kruger, and Vitaly Shmatikov. Towards practical privacy for genomic computation. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 216–230. IEEE, 2008.

[182] Xiao Shaun Wang, Yan Huang, Yongan Zhao, Haixu Tang, XiaoFeng Wang, and Diyue Bu. Efficient genome-wide, privacy-preserving similar patient query based on private edit distance. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 492–503. ACM, 2015.

[183] Kana Shimizu, Koji Nuida, and Gunnar Rätsch. Efficient privacy-preserving string search and an application in genomics. *Bioinformatics*, 32(11):1652–1661, 2016.

[184] Robert H Miller and Ida Sim. Physicians' use of electronic medical records: barriers and solutions. *Health affairs*, 23(2):116–126, 2004.

[185] Guy Paré, Louis Raymond, Ana Ortiz de Guinea, Placide Poba-Nzaou, Marie-Claude Trudel, Josianne Marsan, and Thomas Micheneau. Electronic health record usage behaviors in primary care medical practices: a survey of family physicians in canada. *International journal of medical informatics*, 84(10):857–867, 2015.

[186] Alexandros Mittos, Bradley Malin, and Emiliano De Cristofaro. Systematizing genome privacy research: A privacy-enhancing technologies perspective. *Proceedings on Privacy Enhancing Technologies*, 2019(1):87–107, 2019.

[187] Bradley Malin, Kenneth Goodman, et al. Between access and privacy: Challenges in sharing health data. *Yearbook of medical informatics*, 27(01):055–059, 2018.

[188] The Personal Information Protection and Electronic Documents Act (PIPEDA). `https://goo.gl/TScuoW`. Accessed: 2018-09-16.

[189] Peter Kilbridge. The cost of hipaa compliance. *The New England journal of medicine*, 348(15):1423, 2003.

[190] Andreas Haeberlen, Benjamin C Pierce, and Arjun Narayan. Differential privacy under fire. In *USENIX Security Symposium of the 20th USENIX Security Symposium*, 2011.

[191] Md Momin Al Aziz, Reza Ghasemi, Md Waliullah, and Noman Mohammed. Aftermath of bustamante attack on genomic beacon service. *BMC medical genomics*, 10(2):43, 2017.

[192] Moritz Hardt and Guy N Rothblum. A multiplicative weights mechanism for privacy-preserving data analysis. In *Foundations of Computer Science (FOCS), 2010 51st Annual IEEE Symposium on*, pages 61–70. IEEE, 2010.

[193] Fei Yu, Michal Rybar, Caroline Uhler, and Stephen E Fienberg. Differentially-private logistic regression for detecting multiple-SNP association in GWAS databases. In *International Conference on Privacy in Statistical Databases*, pages 170–184. Springer, 2014.

[194] Caroline Uhlerop, Aleksandra Slavković, and Stephen E Fienberg. Privacy-preserving data sharing for genome-wide association studies. *The Journal of privacy and confidentiality*, 5(1):137, 2013.

[195] Aaron Johnson and Vitaly Shmatikov. Privacy-preserving data exploration in genome-wide association studies. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1079–1087. ACM, 2013.

[196] Yuichi Sei and Akihiko Ohsuga. Privacy-preserving chi-squared testing for genome snp databases. In *2017 39th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pages 3884–3889. IEEE, 2017.

[197] Florian Tramèr, Zhicong Huang, Jean-Pierre Hubaux, and Erman Ayday. Differential privacy with bounded priors: reconciling utility and privacy in genome-wide association studies. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1286–1297. ACM, 2015.

[198] Sean Simmons and Bonnie Berger. Realizing privacy preserving genome-wide association studies. *Bioinformatics*, 32(9):1293–1300, 2016.

[199] Fei Yu, Stephen E Fienberg, Aleksandra B Slavković, and Caroline Uhler. Scalable privacy-preserving data sharing methodology for genome-wide association studies. *Journal of biomedical informatics*, 50:133–141, 2014.

[200] Md Nazmus Sadat, Md Momin Al Aziz, Noman Mohammed, Feng Chen, Xiaoqian Jiang, and Shuang Wang. Safety: Secure gwas in federated environment through a hybrid solution. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2018.

[201] Cynthia Dwork, Aaron Roth, et al. The algorithmic foundations of differential privacy. *Foundations and Trends® in Theoretical Computer Science*, 9(3–4):211–407, 2014.

[202] Indrajit Roy, Srinath TV Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. Airavat: Security and privacy for mapreduce. In *NSDI*, volume 10, pages 297–312, 2010.

[203] Jean Louis Raisaro, Juan Ramón Troncoso-Pastoriza, Mickaël Misbach, João Sá Sousa, Sylvain Pradervand, Edoardo Missiaglia, Olivier Michielin, Bryan Ford, and Jean-Pierre Hubaux. M ed c o: Enabling secure and privacy-preserving exploration of distributed clinical and genomic data. *IEEE/ACM transactions on computational biology and bioinformatics*, 16(4):1328–1341, 2018.

[204] Jean Louis Raisaro, Gwangbae Choi, Sylvain Pradervand, Raphael Colsenet, Nathalie Jacquemont, Nicolas Rosat, Vincent Mooser, and Jean-Pierre Hubaux. Protecting privacy and security of genomic data in i2b2 with homomorphic encryption and differential privacy. *IEEE/ACM transactions on computational biology and bioinformatics*, 15(5):1413–1426, 2018.

[205] Greg Gibson. Population genetics and gwas: a primer. *PLoS biology*, 16(3):e2005485, 2018.

[206] AJ Paverd, Andrew Martin, and Ian Brown. Modelling and automatically analysing privacy properties for honest-but-curious adversaries. *Tech. Rep.*, 2014.

[207] William S Bush and Jason H Moore. Genome-wide association studies. *PLoS computational biology*, 8(12):e1002822, 2012.

[208] EG Co man Jr, MR Garey, and DS Johnson. Approximation algorithms for bin packing: A survey. *Approximation algorithms for NP-hard problems*, pages 46–93, 1996.

[209] Harmonic Series. `https://en.wikipedia.org/wiki/Harmonic_series_(mathematics)`. Accessed: 2018-12-01.

[210] Eric W. Weisstein. Block-Stacking problem.

[211] Stanley L Warner. Randomized response: A survey technique for eliminating evasive answer bias. *Journal of the American Statistical Association*, 60(309):63–69, 1965.

[212] Laura Clarke, Xiangqun Zheng-Bradley, Richard Smith, Eugene Kulesha, Chunlin Xiao, Iliana Toneva, Brendan Vaughan, Don Preuss, Rasko Leinonen, Martin Shumway, et al. The 1000 genomes project: data management and community access. *Nature methods*, 9(5):459, 2012.

[213] Differential Privacy GWAS–implementation. `https://github.com/mominbuet/DifferentialPrivacyGWAS`. Accessed: 2020-09-10.

[214] Lon R Cardon and Lyle J Palmer. Population stratification and spurious allelic association. *The Lancet*, 361(9357):598–604, 2003.

[215] Nour Almadhoun, Erman Ayday, and Özgür Ulusoy. Inference attacks against differentially private query results from genomic datasets including dependent tuples. *Bioinformatics*, 36(Supplement 1):i136–i145, 2020.

[216] IDASH PRIVACY and SECURITY WORKSHOP 2021. Idash privacy security workshop 2021, 2021.

[217] Murat Kantarcıoglu, Jaideep Vaidya, and C Clifton. Privacy preserving naive bayes classifier for horizontally partitioned data. In *IEEE ICDM workshop on privacy preserving data mining*, pages 3–9, 2003.

[218] Jaideep Vaidya and Chris Clifton. Privacy preserving naive bayes classifier for vertically partitioned data. In *Proceedings of the 2004 SIAM international conference on data mining*, pages 522–526. SIAM, 2004.

[219] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

[220] Irene Giacomelli, Somesh Jha, Ross Kleiman, David Page, and Kyonghwan Yoon. Privacy-preserving collaborative prediction using random forests. *AMIA summits on translational science proceedings*, 2019:248, 2019.

[221] Yang Liu, Yingting Liu, Zhijie Liu, Yuxuan Liang, Chuishi Meng, Junbo Zhang, and Yu Zheng. Federated forest. *IEEE Transactions on Big Data*, 2020.

[222] Zhuoran Ma, Jianfeng Ma, Yinbin Miao, and Ximeng Liu. Privacy-preserving and high-accurate outsourced disease predictor on random forest. *Information Sciences*, 496:225–241, 2019.

[223] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-Efficient Learning of Deep Networks from Decentralized Data. In Aarti Singh and Jerry Zhu, editors, *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, volume 54 of *Proceedings of Machine Learning Research*, pages 1273–1282. PMLR, 20–22 Apr 2017.

[224] Theodora S Brisimi, Ruidi Chen, Theofanie Mela, Alex Olshevsky, Ioannis Ch Paschalidis, and Wei Shi. Federated learning of predictive models from federated electronic health records. *International journal of medical informatics*, 112:59–67, 2018.

[225] Mengkai Song, Zhibo Wang, Zhifei Zhang, Yang Song, Qian Wang, Ju Ren, and Hairong Qi. Analyzing user-level privacy attack against federated learning. *IEEE Journal on Selected Areas in Communications*, 38(10):2430–2444, 2020.

[226] Zhibo Wang, Mengkai Song, Zhifei Zhang, Yang Song, Qian Wang, and Hairong Qi. Beyond inferring class representatives: User-level privacy leakage from federated learning. In *In Proceeding of the 2019 IEEE Conference on Computer Communications*, pages 2512–2520. IEEE, 2019.

[227] Eugene Bagdasaryan, Andreas Veit, Yiqing Hua, Deborah Estrin, and Vitaly Shmatikov. How to backdoor federated learning. In *International Conference on Artificial Intelligence and Statistics*, pages 2938–2948. PMLR, 2020.

[228] Martin Abadi, Andy Chu, Ian Goodfellow, H Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 308–318, 2016.

[229] Robin C Geyer, Tassilo Klein, and Moin Nabi. Differentially private federated learning: A client level perspective. *arXiv preprint arXiv:1712.07557*, 2017.

[230] Mu Yang, Andrea Margheri, Runshan Hu, and Vladimiro Sassone. Differentially private data sharing in a cloud federation with blockchain. *IEEE Cloud Computing*, 5(6):69–79, 2018.

[231] Zhanglong Ji, Xiaoqian Jiang, Shuang Wang, Li Xiong, and Lucila Ohno-Machado. Differentially private distributed logistic regression using private and public data. *BMC medical genomics*, 7(1):S14, 2014.

[232] Haoran Li, Li Xiong, Lifan Zhang, and Xiaoqian ang. Dpsynthesizer: differentially private data synthesizer for privacy preserving data sharing. In *In Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, volume 7, page 1677. NIH Public Access, 2014.

[233] Haoran Li, Li Xiong, Xiaoqian ang, and nfei Liu. Differentially private histogram publication for dynamic datasets: An adaptive sampling approach. In *In Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, CIKM '15, page 1001–1010, New York, NY, USA, 2015. Association for Computing Machinery.

[234] June Chen, Wendy Hui Wang, and Xinghua Shi. Differential privacy protection against membership inference attack on machine learning for genomic data. *bioRxiv*, pages 2020–08, 2020.

[235] C-A Azencott. Machine learning and genomics: precision medicine versus patient privacy. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 376(2128):20170350, 2018.

[236] Olivia Choudhury, Aris Gkoulalas-Divanis, Theodoros Salonidis, Issa Sylla, Yoonyoung Park, Grace Hsu, and Amar Das. Differential privacy-enabled federated learning for sensitive health data. *arXiv preprint arXiv:1910.02578*, 2019.

[237] Abukari Mohammed Yakubu and Yi-Ping Phoebe Chen. Ensuring privacy and security of genomic data and functionalities. *Briefings in bioinformatics*, 21(2):511–526, 2020.

[238] Yixin Wang, Jan GM Klijn, Yi Zhang, Anieta M Sieuwerts, Maxime P Look, Fei Yang, Dmitri Talantov, Mieke Timmermans, Marion E Meijer-van Gelder, Jack Yu, et al. Gene-expression profiles to predict distant metastasis of lymph-node-negative primary breast cancer. *The Lancet*, 365(9460):671–679, 2005.

[239] Mario Köppen. The curse of dimensionality. In *In Proceedings of the 5th Online World Conference on Soft Computing in Industrial Applications (WSC5)*, volume 1, pages 4–8, 2000.

[240] Md Momin Al Aziz.

[241] Ahsan Huda, Adam Castaño, Anindita Niyogi, Jennifer Schumacher, Michelle Stewart, Marianna Bruno, Mo Hu, Faraz S Ahmad, Rahul C Deo, and Sanv J Shah. A machine learning model for identifying patients at risk for wild-type transthyretin amyloid cardiomyopathy. *Nature communications*, 12(1):1–12, 2021.

[242] Sergiu Carpov, Kevin Deforth, Nicolas Gama, Mariya Georgieva, Dimitar Jetchev, Jonathan Katz, Iraklis Leontiadis, M Mohammadi, Abson Sae-Tang, and Marius Vuille. Manticore: Efficient framework for scalable secure multiparty computation protocols. *IACR Cryptol. ePrint Arch.*, 2021:200, 2021.

[243] Liyan Shen, Xiaojun Chen, Dakui Wang, Binxing Fang, and Ye Dong. Efficient and private set intersection of human genomes. In *In Proceedings of the 2018 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 761–764. IEEE, 2018.

[244] Duy Vu and Aleksandra Slavkovic. Differential privacy for clinical trial data: Preliminary evaluations. In *In Proceedings of the 2009 IEEE International Conference on Data Mining Workshops*, pages 138–143. IEEE, 2009.

[245] Alessandra Sala, Xiaohan Zhao, Christo Wilson, Haitao Zheng, and Ben Y Zhao. Sharing graphs using differentially private graph models. In *In Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 81–98, 2011.

[246] Liyue Fan, Li Xiong, and Vaidy Sunderam. Differentially private multi-dimensional time series release for traffic monitoring. In *Data and Applications Security and Privacy XXVII*, pages 33–48. Springer, 2013.

[247] Md Nazmus Sadat, Xiaoqian Jiang, Md Momin Al Aziz, Shuang Wang, and Noman Mohammed. Secure and efficient regression analysis using a hybrid cryptographic framework: Development and evaluation. *JMIR medical informatics*, 6(1), 2018.

[248] Edward J Bloustein. *Individual & group privacy*. Routledge, 2018.

[249] Kazi Wasif Ahmed, Md Momin Al Aziz, Md Nazmus Sadat, Dima Alhadidi, and Noman Mohammed. Nearest neighbour search over encrypted data using intel sgx. *Journal of Information Security and Applications*, 54:102579, 2020.

[250] Md Momin Al Aziz, Tanbir Ahmed, Tasnia Faequa, Xiaoqian Jiang, Yiyu Yao, and Noman Mohammed. Differentially private medical texts generation using generative neural networks. *ACM Trans. Comput. Healthcare*, 3(1), oct 2021.

[251] Tanbir Ahmed, Md Momin Al Aziz, and Noman Mohammed. De-identification of electronic health record using neural network. *Scientific reports*, 10(1):1–11, 2020.

[252] Dong Li, Jiahui Wu, Junqing Le, Xiaofeng Liao, and Tao Xiang. A novel privacy-preserving location-based services search scheme in outsourced cloud. *IEEE Transactions on Cloud Computing*, pages 1–1, 2021.