MASTER

Privacy-preserving DNA sequence alignment

Asadova, S.E.

Award date:
2017

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
Privacy-Preserving DNA Sequence Alignment

Master Thesis

Sakina Asadova
0959404

Supervisors:
Dr. ir. L.A.M. (Berry) Schoenmakers
Dr. Meilof Veeningen

Assessment committee member:
Dr. B. (Boris) Skoric

Eindhoven, 25th September 2017
Final version
Abstract

A significant increase in Big Data has triggered the need for joint computation, where people or organizations cooperatively exchange private inputs in order to conduct various computational tasks. These tasks can vary in wide range and occur between untrusted entities. As an example, case of competitor organizations working together for some project with need to jointly share some private database information can be shown. In order to protect organizations' valuable and private assets jointly invested computations should be held securely. Nowadays, in order to carry out such computations usually at least one trusted entity should be chosen and informed about private inputs of the both parties. However, if entities are mutually untrusted and there is no party that can be trusted to perform a computation, they need a cryptographic trusted protocol which ensures privacy entirely.

In the literature, above explained problem is solved with secure Multi-Party Computation (MPC) which has a prime importance in cryptography. MPC performs computation in such a way that, output is ensured to be correct and cheating parties will not be able to reveal any information about the inputs of the honest parties. Even though problem of the MPC has been introduced and solved almost 35 years ago, practical real-world applications have been discovered in various research fields and computation domains only in recent years. One of the most sensitive application fields for MPC is the privacy-preserving database queries in healthcare sector. The problem is to investigate if some private search query exists in a database which has private contents (e.g. DNA sequences) need to be kept secret except query result that can be derived.

Solving string matching problem has been widely researched in the literature, both with and without MPC techniques. However, solving approximate string matching problem under tight privacy concerns is not a trivial task to do. We have used a particular algorithm, namely BWT transform to research the problem of sequence comparison and applied MPC techniques in order to investigate applicability of the method and produce privacy-preserving DNA sequence alignment algorithms. We have implemented our protocols in Python using specific framework VIFF supporting MPC, where underlying protocols are based on Shamir secret sharing.

Due to a judicious use of the secret indexing and masking techniques, we were able to implement the protocols in a recursive manner as in original implementation. In particular, we have identified and analyzed two different models for implemented inexact string matching problem: one model with private search query intended to be searched within public reference string and another model with both private search query and private reference string. For example, in real-world use case private search query can be DNA mutations representing particular illness and reference string can be a human genome. In order to highlight the importance of MPC, both models have been verified and ensured to protect obliviousness entirely.

Additionally, one of the major goals of this study is to introduce and analyze a concrete approach to oblivious verification of the inexact string matching. In particular, approach has been achieved by application of the specific cryptographic concept (i.e. zk-SNARK, namely zero-knowledge succinct non-interactive argument of knowledge) which ensures perfect security due to the zero-knowledge proof. This verifiable computation guarantees correctness proof of the computation and provides full protection against private information disclosure by adversarial verifier even with infinite computational power.
Preface

This master thesis project concludes the graduation project step in Computer Science and Engineering master at Eindhoven University of Technology (TU/e), and has been carried out within Philips Research and Coding Theory & Cryptology Research Group of the Mathematics and Computer Science department of TU/e.

First of all, I would like to express my sincere gratitude to my supervisor at TU/e, Dr. ir. L.A.M. (Berry) Schoenmakers for his entire guidance and advices, critical assessment, and constant encouragement throughout this master thesis project. Thank you for the amount of knowledge, experience and time that you have shared with me in every productive meeting that we had for the last six months.

Second, I would like to thank my supervisor at Philips Research, Dr. Meilof Veeningen for the created opportunity to pursue my thesis project in such an advanced environment from both technical and academical perspectives. From very first day of our research, he helped me to sharpen my analytical thinking and clearly define research priorities. I’m grateful for your availability and dedicated time that resulted in successful completion of this project.

Third, I would like to thank to the Digital Security (DS) research group at the Radboud University and EI/PSI at the Eindhoven University of Technology for offering (TRU/e) specialization in Cybersecurity which has been a valuable experience and wonderful scientific journey in my life for the period of two years.

I would like to express my deepest gratitude to my family, for their endless support, for teaching me ‘never give up’ approach, for stimulating me to think beyond borders and ask in order to solve complex problems that I have been tackled during my studies and in particular, during this research.

I would like also to thank Dr. B. (Boris) Skoric for joining my final assessment committee. Thank you for your time and interest.

Finally, I’m deeply grateful to the city of Eindhoven and obtained friends for their kindness, patience and support. Thank you Nermin, Jamila, Burcu, Ilaha, Emin, Peter, Rob, Berkay, Shamil, Michalis, Oyku, Yan, Fadoua, Khalid, Kai, Nico!

Sakina Asadova, August, 2017
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Contents</strong></td>
<td>v</td>
</tr>
<tr>
<td>List of Figures</td>
<td>vii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>ix</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Thesis Context</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Problem Statement</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Research Question</td>
<td>3</td>
</tr>
<tr>
<td>1.4 Research Framework and Outline</td>
<td>6</td>
</tr>
<tr>
<td><strong>2 Preliminaries</strong></td>
<td>7</td>
</tr>
<tr>
<td>2.1 Suffix Array</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Edit Distance</td>
<td>8</td>
</tr>
<tr>
<td>2.2.1 Sequence Alignment</td>
<td>9</td>
</tr>
<tr>
<td>2.3 Burrows-Wheeler Transform (BWT)</td>
<td>10</td>
</tr>
<tr>
<td>2.3.1 BWT &amp; Suffix Array</td>
<td>11</td>
</tr>
<tr>
<td>2.3.2 Inverse Transform (IBWT)</td>
<td>12</td>
</tr>
<tr>
<td>2.3.3 BWT Backtrack</td>
<td>13</td>
</tr>
<tr>
<td>2.3.4 Long-read Smith-Waterman Alignment (BWA-SW)</td>
<td>19</td>
</tr>
<tr>
<td>2.3.5 Long-read Maximal Exact Match Alignment (BWA-MEM)</td>
<td>20</td>
</tr>
<tr>
<td>2.4 Secure Multi-Party Computation</td>
<td>20</td>
</tr>
<tr>
<td>2.4.1 Security</td>
<td>20</td>
</tr>
<tr>
<td>2.4.2 Motivation</td>
<td>21</td>
</tr>
<tr>
<td>2.4.3 Secret Sharing</td>
<td>22</td>
</tr>
<tr>
<td>2.4.4 VIFF</td>
<td>22</td>
</tr>
<tr>
<td>2.4.5 Operations</td>
<td>23</td>
</tr>
<tr>
<td>2.4.6 Pinocchio-Based zk-SNARKs and Secure Adaptive Function Evaluation</td>
<td>25</td>
</tr>
<tr>
<td><strong>3 Privacy-Preserving Genomic Data Processing</strong></td>
<td>27</td>
</tr>
<tr>
<td>3.1 Related Work</td>
<td>27</td>
</tr>
<tr>
<td>3.2 BWT short-read alignment with multi-party computation</td>
<td>28</td>
</tr>
<tr>
<td>3.2.1 Private Inexact Search</td>
<td>28</td>
</tr>
<tr>
<td>3.2.2 Fully Privacy-Preserving Inexact Search</td>
<td>31</td>
</tr>
<tr>
<td>3.3 Correctness Proof and Adaptive Function Evaluation</td>
<td>35</td>
</tr>
<tr>
<td>3.4 Discussion</td>
<td>39</td>
</tr>
<tr>
<td>3.4.1 Optimizations</td>
<td>39</td>
</tr>
<tr>
<td>3.4.2 Performance Analysis</td>
<td>40</td>
</tr>
<tr>
<td><strong>4 Conclusion</strong></td>
<td>43</td>
</tr>
<tr>
<td>4.1 Limitations and Future Work</td>
<td>44</td>
</tr>
<tr>
<td>4.1.1 VIFF framework</td>
<td>44</td>
</tr>
<tr>
<td>4.1.2 Non-realistic test data</td>
<td>45</td>
</tr>
<tr>
<td>4.1.3 Privacy-Efficiency trade-off</td>
<td>45</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Multi-Party Computation for joint data analytics</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Scenario 1: Private query on private genomic data</td>
<td>4</td>
</tr>
<tr>
<td>1.3</td>
<td>Scenario 2: Private query on public genomic data</td>
<td>5</td>
</tr>
<tr>
<td>1.4</td>
<td>Visualization of the Research Framework</td>
<td>6</td>
</tr>
<tr>
<td>2.1</td>
<td>Ideal World: <em>MPC</em>, 2-party example</td>
<td>20</td>
</tr>
<tr>
<td>2.2</td>
<td>Real World: <em>MPC</em>, 2-party example</td>
<td>21</td>
</tr>
<tr>
<td>2.3</td>
<td>Secure Adaptive Function Evaluation</td>
<td>25</td>
</tr>
<tr>
<td>B.1</td>
<td>Time efficiency of BWT Transform Algorithm with different comparison methods in multi-party scheme</td>
<td>51</td>
</tr>
<tr>
<td>C.1</td>
<td>Time efficiency of Private Inexact Search Algorithm given reference string (length of 70) in one-party scheme</td>
<td>52</td>
</tr>
<tr>
<td>C.2</td>
<td>Time efficiency of Private Inexact Search Algorithm given reference string (length of 10) in one-party scheme</td>
<td>53</td>
</tr>
<tr>
<td>C.3</td>
<td>Time efficiency of Private Inexact Search Algorithm given reference string (length of 20) in one-party scheme</td>
<td>53</td>
</tr>
<tr>
<td>C.4</td>
<td>Time efficiency of Private Inexact Search Algorithm given reference string (length of 20) with all A’s (character variance) in one-party scheme</td>
<td>54</td>
</tr>
<tr>
<td>C.5</td>
<td>Time efficiency of Private Inexact Search Algorithm given reference string (length of 20) in <em>MPC</em> scheme</td>
<td>54</td>
</tr>
<tr>
<td>C.6</td>
<td>Time efficiency of Private Inexact Search Algorithm given reference string (length of 10) in <em>MPC</em> scheme</td>
<td>55</td>
</tr>
<tr>
<td>C.7</td>
<td>Time efficiency of Full Private Inexact Search Algorithm given reference string (length of 10) in one-party scheme</td>
<td>56</td>
</tr>
<tr>
<td>C.8</td>
<td>Time efficiency of Full Private Inexact Search Algorithm given reference string (length of 20) in one-party scheme</td>
<td>57</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Forward transform ($BWT$) & Suffix Array ............................................. 11
2.2 Direct relationship between $BWT$ & $SA$ ................................................. 11
2.3 $BWM$ matrix with rank information ......................................................... 12
2.4 Reverse $BWT$ - Decoding ........................................................................... 13
2.5 Visualization of the Search Algorithm(Exact) steps .................................... 15

A.1 Time Efficiency of the different Secret Indexing techniques ...................... 50
Chapter 1

Introduction

This master thesis is a graduation project for the Computer Science and Engineering master at Eindhoven University of Technology (TU/e) and has been carried out within Philips Research\footnote{http://www.philips.com/a-w/research/locations/eindhoven.html} and Coding & Crypto group of the Mathematics and Computer Science department of TU/e.

Section 1.1 will introduce the thesis context by defining the motivation behind the notion of Multi-Party Computation (MPC) within DNA sequence alignment domain. The problem statement will be formulated in Section 1.2 by emphasizing the difficulties in MPC involved genomic data processing. The research goal of this project will be then presented in Section 1.3, where we also discuss the importance of the zero-knowledge succinct argument of knowledge (zk-SNARK) of a MPC in our verification method development. Section 1.4 will present the overall research framework and the outline of the following chapters in this thesis.

1.1 Thesis Context

Big Data generation and collection speed has increased exponentially in the current era. Organizations are not only interested with particular data, but also with insights and details about what is going to happen next when this data being analyzed. The demand for unlocking potential of the collected data and rapid developments for this are driving knowledge for organizations from various sectors. These developments include capture, secure storing and analysis of the Big Data. As a result, organizations focus shift to exploration and exploitation of this data\cite{29}. However, this process requires continuous sharing and storing of the confidential data between governmental and commercial organizations for medical, law enforcement and other purposes. In addition, willingness and consent of the data owners are important factors in order to have their private data to be stored. Ensuring data owners that the collected private data is going to be used only in legitimate way and stored in safe databases is the important task for organizations to succeed\cite{9}.

Potential of Big Data analytics and privacy-preserving problem by all means, are widespread in various sectors that deal with personal information. One of the most crucial and intensive areas which needs this protection is a healthcare sector, in particular for privacy-preserving medical database queries. In other words, when collected data contains high sensitivity as in human genome(e.g. DNA sequences or protein sequences), its vulnerable to various attacks even after being anonymized and increases a risk to reveal patient-specific information. Basic operations and their applications on DNA sequences are obtained within computational biology field\cite{16}.

Computational biology is an interdisciplinary field that has become a separate discipline in itself with academic importance dedicated to it. From this perspective, an important notice is due to the computational biology, DNA, RNA and protein sequences rendered as data and they managed and manipulated by data driven tools that reveal hidden and unexpected patterns in them\cite{27}. In recent years, urgent need of securing private data and the progress in bioinformatics engineering
have led to new opportunities as well as challenges.

This research is performed at Philips Research which seeks to engage in clinical research collaborations with DNA-sequencing technologies (i.e. Illumina) and improve large scale analysis of the human genome variances and sequencing technologies\(^2\). For example, rapid profiling difficult cases such as tumors using genomic information is crucial in order to solve complex cancer cases. Due to achieve solution on this, one of the widely used techniques is a new type of the DNA sequencing which aims to find mutations (i.e. sequence disorders). However, sharing high volumes of a sensitive genomic data poses challenges which can be solved by continuously developing ways in order to ensure safety.

Utilizing \textit{MPC} techniques can address these challenges and therefore, achieve distributed secure genomic data processing. This is possible due to the data-oblivious analysis of the data, which means that instructions to carry out computations and accessed memory are not directly related with data. Therefore, computations are being executed jointly among multiple parties\[^{[30]}\]. From this perspective, \textit{MPC} conceptual model might look like the one described in Figure 1.1:

\[\text{Figure 1.1: Multi-party computation for joint data analytics}^{3}.\]

This thesis is a pre-process study for the SODA project by Philips Research which will enable privacy-preserving analytics on Big Data by ensuring several achievements:

- In addition to traditional privacy-efficiency trade-off, analyzing Big Data for real world use-cases by using \textit{MPC} techniques for perfect secrecy;
- Further development of existing tools in order to achieve performance and real privacy improvements simultaneously for relevant stakeholders;
- Explanation of applied techniques during the project to avoid data leakage of data analytics results, hence making data subjects more confident;
- Verification of approach and correctness proof in terms of both functionality and security with hacking challenges and reverse engineering.

\(^2\)Philips and Illumina Collaboration, for more information refer to: \url{http://philips.to/2vNMepD}\n
\(^3\)adopted from SODA - Scalable Oblivious Data Analytics, for more information refer to: \url{http://ec.europa.eu/newsroom/document.cfm?doc_id=41838}
1.2 Problem Statement

With arrival of the new sequencing technologies, new generation of the sequence alignment tools have been introduced. Most of them are considerably efficient for short read sequence alignment, namely for arranging comparably short sequences to analyze their similarities or differences, as short queries result in low error rate. However, sequences produced by real-world sequencing technologies are not short reads anymore, except the ones are designed specifically for a particular length range of the sequences.

As discussed earlier, to achieve even simple oblivious data-independent computations on genome processing poses challenges, as genomic data is considerably large. However, in order to succeed more complex functionalities and meet objectives, various non-trivial tricks should be involved in computational tasks. Computing edit distance between two different genome sequences, namely finding similarities between them can be considered as one example of the complex functionalities in a secure setting. In other words, procedure involves alignment of the two different genome sequences which is not straightforward task in MPC scheme. Logical solution to address this challenge can be computation of the edit distance between input sequences and then, implementation of the particular two-party solution in a multi-party context.

In order to identify similar regions of DNA, RNA or protein sequences, namely to solve sequence alignment problem and further, achieve optimizations on it, various algorithms have been applied by researchers from wide range of computer science branches. One of these optimization algorithms is BWT transform in particular, due to the usage of dynamic programming recurrence relation.

In this thesis, we discuss privacy-preserving approach in order to achieve provably secure solution of the string matching problem. Specifically, we demonstrate applicability of the secure MPC to the solution of distributed approximate common string search problem by BWT algorithm. The problem statement that will be explored in this thesis can be defined as follows:

**Problem Statement:** Given specific sequence alignment algorithm BWT transform, how to apply multi-party computation techniques in order to generate a privacy-preserving string comparison method.

1.3 Research Question

It has been indicated that addressing sequence alignment problem in MPC context is crucial in order to ensure privacy of the distributed computation model. Security requirements in these computational models directly affect the efficiency of the involved sequence alignment algorithms while enabling joint computation.

Our work might be compared to [11], where authors have developed an oblivious secure comparison protocol in multi-party context to compute edit distance between genomic sequences. In addition, they also have contributed zero-knowledge proof which prevents client detection by revealing information from output and ensures that same input has been used by involved parties. This construction is similar to the framework that we have proposed. On the contrary, in our project we focus on the specific sequence alignment algorithm, namely BWT transform instead of the general sequence comparison methods. Moreover, we check the correctness of the proposed approach by verifying it with zero-knowledge proof concept.

The general objective of this thesis is to understand the role of MPC and the use of data analytics to analyze applicability of it both from functionality and security perspective. Despite the fact that security and privacy-preserving always come with additional cost, in this thesis the goal is to identify crucial requirements for perfect secrecy in limited framework (VIFF) and develop techniques for performance improvements in realistic scenarios with genomic data. Research questions

---

that will be explored in this thesis project can be introduced as follows:

**Research Question:**

1. Given the Virtual Ideal Functionality Framework (VIFF), by applying MPC techniques to particular genomic sequence comparison algorithm - BWT transform, we would like to develop secure distributed protocol which will help to achieve full privacy-preserving sequence comparison within two construction:
   
   (a) **Private query on private genomic data**;
   
   (b) **Private query on public genomic data**.

2. Achieve verification and correctness proof of the proposed approach for all participants by zk-SNARK concept.

It is important to state that, during the implementations of the algorithm we have investigated two sequence comparison models in MPC context as defined above. As a real-life example let us now examine Figures 1.2 and 1.3 by defined scenarios below:

**Scenario 1:**

- Alice has a genomic data, namely digital record of her DNA saved in flash drive or smart-card and she wants to take one specific genomic test. This genomic test is provided by Cloud server which uses private query obtained from Bob. Alice does not trust Bob and wants to protect her privacy. Alice does not have the computational power and query to execute test herself.

- On the other hand, Bob does not want to reveal his private query neither to Alice nor another party and he does not trust the digital device containing DNA record of the Alice. He still wants to have Alice’s genomic data in order to execute genomic test which is important for his research. Bob does not have the computational power and DNA record to execute particular test himself.

- In this scenario, neither Alice nor Bob does not trust each other and both worried about privacy of data. Third party (Cloud) has been used due to its computational power and also not trusted by Alice and Bob. All three parties execute joint computation by running VIFF.
and using power of the Cloud server, query from Bob, DNA record from Alice. Correctness of the computation then, obtained by zero-knowledge proof based on the result that each party gets. Privacy of all parties has been preserved.

Figure 1.3: Scenario 2: Private query on public genomic data

**Scenario 2:**

- Alice is a scientist who is interested in exploring public health care data conducted from publicly available database for some research topic.
- She trusts publicly available database that provides DNA record, however she wants to keep her query private in order to preserve confidentiality of the study. As genomic data is large, Alice cannot execute computation and outsource it to the Cloud servers, however she also does not trust Cloud servers that can execute genomic computation.
- Alice gets separate response from each Cloud as a result of genomic computation. The cloud servers run VIFF between each other. Then, verification step is possible due to zero-knowledge proof executed on the collected responses. Alice ensures correctness of the computation by obtained zero-knowledge proof. Privacy of all parties has been preserved.

Moreover, sequence matching procedure is performed for model where only query string or both query and reference sequences are the secret data. To justify this statement, we can think of looking for some mutation sequence within genomic data sequences. Sequenced complete human genome is a public or private data which can be stored in huge genomic archives ideally along with patient-specific treatments and disease outcomes[26]. On the other hand, mutations on genomic sequences are novel rearrangements which can reveal result of the particular string matching problem, in other words can infringe patient’s privacy.

One of the major goals of this final master project is to introduce a concrete approach to oblivious verification of inexact string matching. This is possible due to the zero-knowledge proof that can be
executed by Client on responses that it gets from each individual entity that executes computation. In particular, approach has been succeeded due to use of reverse engineering, namely by storing result of the computations in between, and application of the cryptographic concept (i.e. zk-SNARK) which ensures perfect security based on the zero-knowledge proofs. These verifiable computations guarantee correctness proof of the computation and provide full protection even against adversarial verifier with infinite computational power.

1.4 Research Framework and Outline

Figure 1.4 illustrates research framework of this study.

Following methods have been used in order to address problem statement and give solution to the research questions:

- **In Chapter 2**, importance of the suffix array, edit distance, sequence alignment concept for our domain has been introduced first. Next, BWT transform algorithm and detailed explanation of its significant features have been discussed. As a background information, other two sequence alignment algorithms (i.e. BWA-SW, BWA-MEM) based on the utilized BWT transform algorithm have been discussed. As a next step, we gave the building blocks to achieve oblivious genomic processing in multi-party scheme, namely introduce secure MPC and VIFF framework. Further, zero-knowledge succinct argument of knowledge (zk-SNARK) of MPC will be presented and discussed as a verification scheme.

- **Chapter 3** presents our two versions of oblivious short-read sequence alignment algorithm and oblivious verification in order to ensure correctness of the computations held by these algorithms. Algorithms differ mainly by implemented security models, namely being private query on private data and private query on public data, while presenting a set of steps required for the successful implementation of obliviousness. This chapter will introduce our final approach for the privacy-preserving genomic data processing on MPC context while introducing several invariants with different optimizations. We also present correctness proof algorithm and concrete approach to oblivious verification of string matching algorithm. This verification scheme ensures correctness of the oblivious computations held by particular algorithm.

- **Conclusion of this thesis** will be presented in Chapter 4 that will summarize results collected from performance analysis and entire approach both from functionality and security perspectives, additionally contain limitations of our method and future work discussions.
Chapter 2

Preliminaries

We have formulated our problem statement and research question in the previous chapter. In this chapter we will present preliminary concepts that will be used throughout this thesis project.

In Section 2.1, we will briefly introduce certain data structures, which are used for the assessment of sequence alignment in particular, focusing on the structure of string search. Edit distance concept for the domain of general string matching and search techniques will be presented in Section 2.2 covering approach comparison for time efficiency. We will continue with Section 2.3 in which we analyze entire Burrows-Wheeler Transform algorithm and explain detailed motivation behind each computation. This section will also give information about other algorithms (i.e. BWA-SW, BWA-MEM) based on work principle of BWT algorithm. In Section 2.4 will describe our motivation behind the multi-party computation approach that have been selected in order to achieve oblivious sequence alignment. Additionally, we will introduce zero-knowledge proof concept for verification of the novel oblivious alignment approach that has been used in this project.

2.1 Suffix Array

Authors of [21] have introduced suffix array as a simple, space efficient alternative to the suffix tree which is a trie that stores all suffixes of the given string as their keys and positions in the string as values of these keys. Suffix array (SA) is simply the suffix positions obtained by traversing leaf nodes of the suffix tree from left to right in lexicographic order. Furthermore, SA can be interpreted as lexicographically ordered list of the suffixes for string X, represented by pointers to their entry positions. In order to understand definition above better, let \( i = 0, 1, \ldots, n - 1 \), \( X[i] = a_i \) be the \( i^{th} \) symbol of the \( X \), \( X[i, j] = a_i \ldots a_j \) a substring within \( X \), and \( X[i, n - 1] \) a suffix of \( S \). Suffix array \( SA \) of \( X \) is a permutation of the integers \( 0 \ldots n - 1 \) such that \( SA[i] \) is the start position of the \( i^{th} \) smallest suffix due to the lexicographic order. Minimal space need for the construction of the suffix array is \( n \log n \) in original algorithm[21].

2.2 Edit Distance

In computer science, edit distance (or Levenshtein distance) is utilized to measure the similarity between two strings. Given two strings \( A \) and \( B \), edit distance is the minimal number of edit operations needed to transform string \( A \) to string \( B \) and denoted as \( ED(A, B) \)[23]. In particular, these operations consist of insertion, deletion and substitution on a single character. This concept for string \( A = a_1 \ldots a_i \) and \( B = b_1 \ldots b_j \) can be formulated as follows:

- Insertion of element \( b \) to \( k^{th} \) position, resulting in \( A = a_1 \ldots a_k b a_{k+1} \ldots a_i \)
- Deletion of element from \( k^{th} \) position, resulting in \( A = a_1 \ldots a_{k-1} a_{k+1} \ldots a_i \)
- Substitution, namely change of element at \( k^{th} \) position to element \( b \), resulting in \( A = a_1 \ldots a_{k-1} b a_{k+1} \ldots a_i \)
These are allowed edit operations to transform one string to another. For each particular operation there exists a cost associated with it. Furthermore, to insert an element $b$ in a particular position, operational cost denoted as $I(b)$, while cost of deleting an element $b$ denoted as $D(b)$ and substituting an element $a$ with element $b$ represented as $S(a, b)$. For Levenshtein distance $I(b) = D(b) = S(a, b) = 1$, whereas other costs are also possible in different schemes. The actual series of insertion, deletion and substitution required to achieve optimal edit distance can be defined as edit script or edit path. Each edit script, namely sequence of operations that transforms one string into another has an associated total operational cost which is the sum of costs of insertion, deletion and substitutions that it contains[4].

In this thesis, we have concentrated on two variant of the string matching problem, namely exact and inexact search, which based on original and approximate string matching, respectively. String matching problem is defined as finding one or all occurrences of a search query $W = [w_1 \ldots w_m]$ of length $m$ in original string $X = [x_1 \ldots x_n]$ of length $n$. Therefore, exact search can be defined as follows:

**Definition 2.2.1. Exact Search**

Given a string $X$, a search query $W$, exact search finds starting positions for each consecutive segment of $W$ in $X$, that is, $X=YWZ$.

On the other hand, inexact search can be defined as variant of approximate string matching problem, namely as the method of finding strings that match a particular pattern, namely a search query approximately[23].

**Definition 2.2.2. Inexact Search**

Given a string $X$, a search query $W$ and non-negative integer - bound $T$, inexact search finds starting positions for each substring $W'$ in $X$, where the edit distance $(W, W') \leq T$, that is, $X=YW'Z$.

Moreover, string matching problem being in computer science corresponds to the sequence alignment in computational biology. Therefore, in order to investigate DNA sequence comparison problem in more detail, we proceed to the next step of our explanation, namely to the sequence alignment concept in next section.

### 2.2.1 Sequence Alignment

In biology, sequence alignment is the way of arranging two sequences in order to analyze their similarity and differences over particular alphabet $\sum$. This alphabet is a set of ordered of characters and will be used within this thesis as $\sum=[A, C, G, T]$ where each symbol represents the chemical components of DNA sequences: Adenine (A), Cytosine (C), Guanine (G), and Thymine (T), respectively. The sequence alignment concept can also be applied for non-biological sequences, to measure the edit distance(2.2) between given data. That is, calculating number of changes needed to get one sequence from another.

A sequence over alphabet $\sum$ is an enumerated collection of characters from the $\sum$. The length of the sequence corresponds to the number of elements it consists. One way to denote sequences is to list their elements, furthermore given sequences $A$ and $B$ with the length of $i$ and $j$ respectively, the notations $A = a_1, a_2, \ldots, a_i$ and $B = b_1, b_2, \ldots, b_j$ over alphabet $\sum$ can be used.

An alignment is assignment of gaps to particular positions in $[0, ..., i]$ of $A$, and $[0, ..., j]$ of $B$, in order to line up each element in one sequence with either an element, or a gap in the other sequence. If the sequences $A$ and $B$ share mutual ancestors in one alignment, namely likelihood of these two sequences being genuinely related due to their base pairs, gaps between them can be interpreted as indels, namely insertion or deletion operations in either of the sequences. It also can be interpreted as a gap penalty in which, every time a skip is made in one or the another sequence in order to determine optimum matching between them.

Available alignment methods correspond to either consecutive selection of each letter from sequences, or introducing indel to the matching element in the first sequence and place suitable match
in the another sequence, preferably. Alignment is also possible in other way around, namely by intro-
ducing an indel to the corresponding element in the second sequence and place opposite element
in the first sequence. An example to the sequence edit operations can be represented as:

**Sequence:** G C C C T A G C G  
**Substitution:** G C C C A T A G C G  
**Insertion:** G C C C C T T A G C G  
**Deletion:** G C C - T A G C G

In order to estimate best alignment between two DNA sequences, further maximizing the number
of matches and minimizing the number of spaces and mismatches, scoring method is efficient.
Moreover, a score for each possible alignment can be determined by adding no points for matching
characters and penalizing points for gaps and mismatches within sequence. This is possible by
both *global* and *local* sequence alignment techniques. *Global* sequence alignment aims to find the
best alignment, namely shortest edit distance between an entire sequence $A$ and entire sequence $B$,
recursively. In a scheme where matches are not awarded by any point, while mismatches and spaces
penalized by one point an optimal global alignment example can be represented as follows:

| Sequence $A$: G C C C T A G C G |
| Sequence $A'$: G C C C T A G C G |
| Sequence $B$: G C G C A A T G |
| Sequence $B'$: G C G C - A A T G |

- One gap in $B'$ (contrarily, an insertion in $A'$), five matches and three mismatches.

    \[
    \text{Edit distance: } [5 * 0] + [1 * 1] + [3 * 1] = 4
    \]

Yielded score for alignment is the best that can be obtained within given scoring scheme$^1$.

On the contrary, there are no constraint such as alignment of whole sequences in *local* sequence
alignment. It finds a region with highest similarity between input sequences that may or may not
be related. It is appropriate to mention that, best local alignment score is greater or equal to the
best global alignment score within the same scoring scheme. Major reason behind this the fact that,
global sequence alignment is also a local alignment.

On the other hand, it is interesting to mention another type of sequence alignment, which is
*semi-global* alignment. Given one short and one relatively longer input sequence, semi-global align-
ment aims to answer question whether short sequence can be found within the longer sequence
even with possible errors$^2$. This problem is a variant of the inexact search problem introduced in
Definition 2.2.2.

In order to align short and less varying sequences, sequence alignment can be computed by
hand. However, sometimes the vague surviving similarities and lengthy structure of the sequences
make it difficult to validate relation among them solely by human effort. Instead, computer-based
sequence alignment procedure is required by using algorithms constructed by human knowledge in
order to align sequences and also, reflect the patterns among them which can be extremely difficult
to determine algorithmically.

Computational approaches applied for sequence alignment include several dynamic programming
methods, which can be slow but computationally simple, ensuring that no duplicate work has been
involved. These advanced techniques can solve algorithmic optimization problems based on the
bottom-up approach, namely by finding optimal solutions to subproblems first. Computer-based
solution to the sequence alignment problem is possible by several efficient software tools that apply
advantages of the dynamic programming. As an example, BWA is a software tool to align relatively
similar sequences against a large reference sequences, e.g. the human genome. Tool performs based
on the three algorithms: BWA-backtrack($BWT$ transform), BWA-SW and BWA-MEM$^3$.

---

$^1$Dynamic programming and sequence alignment, for more information refer to: https://ibm.co/2v72HDt

$^2$Introduction to sequence alignment, available in: https://www.slideshare.net/Kubuldinho/introduction-to-sequence-alignment

2.3 Burrows-Wheeler Transform (BWT)

Algorithm was invented by David Wheeler and Michael Burrows in 1994. The main concept of the algorithm is to rearrange a character string into sequence of the non-divergent characters. Burrows-Wheeler Transformation (BWT) became popular after its recognition as compression tool, as in particular compressed indexes based on the BWT were probably the simplest and most space efficient ones. Currently, it is one of the most popular index structure for genomic data processing, used by a wide range of the alignment tools\(^4\). BWT algorithm is an efficient approach in order to align short sequencing reads (up to 100 base-pairs, bp) to the reference genome (sequence, from now on).

\(BWT\) consists of a reversible permutation of the characters in a sequence. Algorithm tends to have the convenient property as to group characters which appear contiguously in substrings. This feature is useful for data compression, as a sequence which consists of the identical characters in several locations is relatively easy to compress\(^3\). Property can be exploited also in sequence alignment, namely in a transformation of the reference string which contains several substrings that occur often, where one character can be obtained in a repetitive manner.

In order to analyze entire transformation procedure, we distinguish between two categories of a transform: forward transform or BWT, and a backward transform (inverse) or IBWT which constructs the original sequence back from the transformed one. Forward transform can be obtained in three major steps:

- Given sequence \(X\), append a special sign ‘$’ that appears nowhere else within sequence. It is alphabetically prior to any character of the \(X\).

- Rotations: get the cyclic shifts of the sequence \(X\), that is taking an element from one end of the \(X\) and sticking it on to other end, and store each rotation in a row of \([n \times n]\) matrix given \(X\) length of \(n\).

- Sort and construct BWT: get lexicographic sort of the rows in matrix with respect to order of \($ < A < C < G < T\) over alphabet \(\sum = \{\$, A, C, G, T\}\). Resulted matrix is the Burrows-Wheeler Matrix (BWM). The transformed sequence (\(B\)) is the concatenation of the last symbols of the cyclic rotations, namely the last column of the BWM read from top to bottom.

It would be appropriate to mention that, every column of the matrix BWM is the permutation of the transform \(B\). In particular, the first column of the matrix BWM, calling it \(F\), can simply be constructed by lexicographic sort of the transform \(B\).

All major steps of the forward transform, BWT are depicted in Table 2.1. As an example, constructing suffix array and transform \(B\) for original sequence \(X=AGGAGC\) is illustrated. Sequence \(X\) is circulated to generate seven rotations, which are then lexicographically sorted. After sort phase, the positions of the first characters in each row of the BWM construct the suffix array \((6, 3, 0, 5, 2, 4, 1)\). As a final step, concatenation of the last characters of the cyclic shifts from top to bottom forms BWT transform as \(B=CG$GGAA\).

---

### 2.3.1 BWT & Suffix Array

In order to analyze apparent relationship between BWT and suffix array (SA), it important to observe that, suffixes represented in Table 2.1 obtained by deleting everything after special character ‘$’ in the reference sequence $X$. These suffixes are still in sorted order and SA represents the start positions for them. Moreover, sorting the rows in BWM matrix is the same as sorting suffixes of the original sequence $X$.

The direct relation between transform $B$ and SA can be represented as follows:

$$X'[SA[i]] = B[i], \text{ where } X' = $X \text{ for } i \in [0 \ldots \text{len}(B)) \text{ in transform } B \quad (2.1)$$

### Table 2.2: Direct relationship between BWT & SA

<table>
<thead>
<tr>
<th>Reference String $X$</th>
<th>Suffix Array (SA)</th>
<th>$X[\text{suffix position}] = \text{transform}(B)$</th>
</tr>
</thead>
</table>

Optimized simple BWT implementation computing BWT transform and suffix array as output represented below:

**Algorithm 1 BWT transform**

1. procedure $BWT$(INPUT: $X$, OUTPUT: $B, SA$)
   ▶ Pre: $X$: ’$’+ reference string
   ▶ Post: $B$: transform, last column of the BWM, $SA$: suffix array
2. for each $i \in \{0, \ldots, \text{len}(X)\}$ do
   3. tab$_i = X[i + 1, \text{len}(X)] \| X[0, i + 1] \| i$ ▶ create table of cyclic rotations
4. end for
5. $s_{\text{tab}} \leftarrow \text{lexicographic sort of tab}$ ▶ sorted table
   ▶ get transform
6. $B \leftarrow s_{\text{tab}_0,\text{len}(X)-2}, \ldots, s_{\text{tab}_{\text{len}(X)-1,\text{len}(X)-2}}$ ▶ get suffix array
7. $SA \leftarrow s_{\text{tab}_{\text{len}(X)-1,\text{len}(X)-1}}$ ▶ get suffix array
8. return $B, SA$
2.3.2 Inverse Transform (IBWT)

The relationship between transform $B$ and suffix array $SA$ described above in Equation 2.1, is also exploited by inverse transformation $IBWT$. Output of the Forward Transform Algorithm (see Algorithm 1), namely suffix array $SA$ and transform $B$ is the input for the $IBWT$ algorithm represented below:

Algorithm 2 IBWT inverse transformation

1: procedure $IBWT$ (INPUT: $B, SA$, OUTPUT: $X$)
   \[\triangleright\] Pre: $B$: last column of the BWM, $SA$: suffix array
   \[\triangleright\] Post: $X$: reference string without ‘$’
2: $X' \leftarrow [0, \ldots , 0]$ 
3: for each index $\in \{0, \ldots , \text{len}(SA)\}$ do
4:    $\text{target index} = SA[index]$
5:    $X'[\text{target index}] \leftarrow B[index]$
6: end for
7: $X \leftarrow X'_{1}, \ldots , X'_{\text{len}(X')-1}$ \[\triangleright\] eliminate ‘$’
8: return $X$

It is important to observe a clear connection between Equation 2.1 and Algorithm 2 which obtains original reference string $X$ from output $X' = $X, by simply eliminating the first character. Suffix array contains positions of suffixes, however it can be used for inverse transformation due to the $L-F$ mapping which we will define below:

Prior to explain $L-F$ mapping and the complete process behind $BWT$ algorithm[18], it is important to mention two observations about cyclic rotations of the characters in string $X$:

- Given $F$ as a first column and $L$ as a last column of the $BWM$, $L[i]$, being the last character in the $i^{th}$ row of $BWM$, precedes the first character $F[i]$ of the the same row in the string $X$. In particular, this relation can be interpreted in original string as $X=[..L[i]F[i]..][12]$.

- We define the rank of a character $ch$ as number of times that character occurs previously within $X$. Further, first occurrence of $ch$ becomes $ch_0$, the second occurrence of $ch$ becomes $ch_1$, etc. Symbol ‘$’ is not ranked, as it can occur only once and not within string.

In Table 2.3, $BWM$ has been represented including ranks which do not affect lexicographical order. $L-F$ mapping states that, $i^{th}$ occurrence of character $ch$ in last column ($L$), has the same rank(order) as the $i^{th}$ occurrence of $ch$ in first column ($F$). As can be clearly seen, for given character $ch=’G’$, the ranks of $ch$, $(G_1,G_2,G_0)$ in $L$ have the same order as $(G_1,G_2,G_0)$ in the first column $F$.

<table>
<thead>
<tr>
<th>$BWM$ &amp; $L-F$ mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_0G_0G_1A_1G_2C$</td>
</tr>
<tr>
<td>$A_1G_2C$A_0G_0G_1</td>
</tr>
<tr>
<td>$A_0G_0G_1A_1G_2C$</td>
</tr>
<tr>
<td>C$A_0G_0G_1A_1G_2$</td>
</tr>
<tr>
<td>$G_1A_1G_2C$A_0G_0</td>
</tr>
<tr>
<td>$G_2C$A_0G_0A_1G_2</td>
</tr>
<tr>
<td>$G_0G_1A_1G_2C$A_0</td>
</tr>
</tbody>
</table>

Given $BWM$ for the string $X$, assume $BWM'$ as matrix obtained by rotating all rows of $BWM$ one position to the right. The reasoning obtained from the first observation defined
above concludes that, the first column of BW M' equals the last column of BW M.

Given any \( ch \), we can observe that \( ch \)'s rank in first column \( F \) of BW M appears in same order as in first column, say \( F' \) of BW M'. Major reason behind this is the fact that, sort is executed in both BW M and BW M' rows in the same manner. In other words, as rows of BW M is sorted starting from the first position, rows of BW M' are attached regard to their first positions and sorted starting from their second positions. Combining this reasoning and the fact that, first column of BW M' equals the last column of BW M, L-F mapping holds[5].

One of the major advantages of L-F mapping for IBWT is its ability to deliver the reference string \( X \) by backtracking \( B \) only. This is possible due to the previously defined fact that, the first row in the BW M starts with a ‘$’ sign and the last character in the first row is possible to obtain directly from the \( B \), which will clearly indicate that the last character in \( X \) is the corresponding character. Furthermore, reconstructing the first column by simply sorting \( B \) lexicographically and obtaining first and last elements in each row of BW M will reveal some information[5]:

- It will reveal all pairs of successive characters in \( X \), where pairs are obtained cyclically, so that the last and first character from each row form a pair. Sorting obtained pairs results into construction of the first and second columns, therefore continuing in this manner produces the complete reversion of the \( X \).

This process can be clearly followed from Table 2.4, where input for transformation process of the string is simply \( B \). As a result of sorting each character in \( B \), the first column in reverse matrix BW M', namely \( B' \) can be obtained. As a next step, \( B \) is prepended in the matrix and sorting accordingly results with first two column of the BW M'. Following similar steps in a range \( i \in [0..n] \) where \( n \) is the length of the \( B \), achieves decoding of BW M' completely, hereby selecting the row which ends with separator ‘$’ (in Table 2.4) outputs the reference string \( X \).

<table>
<thead>
<tr>
<th>( B )</th>
<th>Sorted (( B' ))</th>
<th>( B + B' ) (seq)</th>
<th>Sorted(seq) (seq)'</th>
<th>( B+(seq)' )</th>
<th>BW M'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 C</td>
<td>$</td>
<td>C$</td>
<td>$A_2$</td>
<td>$A_2$</td>
<td>$A_2G_3G_1A_2G_2C'$</td>
</tr>
<tr>
<td>1 G</td>
<td>A</td>
<td>G_1A_1</td>
<td>A_1G_2</td>
<td>$A_1A_2$</td>
<td>$A_1G_2C$A_2G_3G_1</td>
</tr>
<tr>
<td>2 $</td>
<td>A_2</td>
<td>$A_2G_3$</td>
<td>A_2G_3</td>
<td>$A_2G_3A_2G_3C$</td>
<td></td>
</tr>
<tr>
<td>3 G</td>
<td>C</td>
<td>G_2C</td>
<td>C$</td>
<td>G_2 C$</td>
<td>C$A_2G_3G_1A_2G_2</td>
</tr>
<tr>
<td>4 G</td>
<td>G_1</td>
<td>G_1G_1</td>
<td>G_1A_1</td>
<td>G_1G_1A_1</td>
<td>G_1A_1G_2C$A_2G_3G_1</td>
</tr>
<tr>
<td>5 A</td>
<td>G_2</td>
<td>A_1G_2</td>
<td>G_2C</td>
<td>A_1G_2C</td>
<td>G_2C$A_2G_3G_1A_1</td>
</tr>
<tr>
<td>6 A</td>
<td>G_3</td>
<td>A_2G_3</td>
<td>G_3G_1</td>
<td>A_2G_3G_1</td>
<td>G_3G_1A_1G_2C$A_2</td>
</tr>
</tbody>
</table>

### Table 2.4: Reverse BWT - Decoding

#### 2.3.3 BWT Backtrack

**Exact Backward search**

The motivation behind exact backward search is the same as in original exact string matching problem from Definition 2.2.1. BWT exact backward search uses above reasoning for problem definition, as given search query \( W \), algorithm finds starting positions for all occurrences of exact consecutive segment in reference string \( X \). Firstly, in pre-processing step some auxiliary array-based data structures should be set up:

- Let \( B \) be the BWT transform of \( X \).
- Let \( \sum \) be a constant-size alphabet consists of characters \( ch \in \{A,C,G,T\} \)
- Let \( C(ch) \) be the number of elements in original string \( X \) that lexicographically smaller than \( ch \).
• Let \( O(ch, index) \) be the number of occurrences of character \( ch \) in transform \( B[0 : index] \)
  hence its utilized to implement L-F mapping.

Exact Backward search shown below[18]:

Algorithm 3 Exact Matching: Backward Search

1: procedure BACKWARD SEARCH(INPUT: \( W, i, k, l \) OUTPUT: \( I \))
  \( \triangleright \) Pre: \( W \) : search query, \( i \) : index in \( W \), \([k, l]\) : SA interval corresponding to \( W[i:] \) where \( 0 \leq k \leq l \leq n \), for \( n = \text{len}(X) \)
  \( \triangleright \) Post: \( I \) : SA interval corresponding to the occurrences of \( W \)

2: \( I \leftarrow \text{set()} \) \( \triangleright \) initialization to empty set
3: if \( i < 0 \) return \([k, l]\)
4: for each \( ch \in \{A, C, G, T\} \) do
5: \( k_p \leftarrow C(ch) + O(ch, k - 1) + 1 \)
6: \( l_p \leftarrow C(ch) + O(ch, l) \)
7: if \( k_p \leq l_p \):
8: \( \text{if } ch = W[i] \text{ then:} \)
9: \( I \leftarrow \text{Backward Search}(W, i - 1, k_p, l_p) \)
10: end for
11: return \( I \)

As it has already proven by authors of [12], Backward Search Algorithm exploits the relation between SA and BWM. In order to understand motivation behind work principle of the algorithm, it would be appropriate to recall the specific properties of the suffix array SA:

1. observing that all suffixes of the string \( X \), occur as prefixed intervals in corresponding rows of the \( BWM \);
2. this interval has a starting point \( k \) and an ending point \( l \), where \( k \) represents order(position) of such an interval among lexicographically sorted sequence of suffixes.

In order to proceed to next step of analysis, following precondition needs to be considered:

All occurrences of the search term \( W[i:] \) occur at suffix positions \( SA[k], SA[k + 1], \ldots SA[l] \) in \( X \). In particular, initial values for this interval are \( k = 0 \) and \( l = len(W) - 1 \), representing start and end positions in suffix array, respectively. As suffix at \( i^{th} \) position is empty in the beginning, \( i \) is the last character in \( W \), namely \( len(W) \). Each next step in search algorithm will follow the procedure accordingly:

Given suffix interval \([k_p, l_p]\) in SA at the \( i^{th} \) phase of the search algorithm, \( k_p \) represents the first row within \( BWM \) prefixed by this interval, while \( l_p \) points to the last row within \( BWM \) prefixed by same interval. Moreover, an interval \([k_p, l_p]\) constitutes a set of positions of all occurrences of \( W \) in \( X \) and identified as SA interval.

If \( k_p > l_p \), then we conclude that \( W[i:] \) does not occur within \( X \) and therefore, \( W \) does not also. Next, algorithm iteratively approximates the \([k, l]\) interval in order to count occurrences of \( W \), namely calculating \( k_p \) and \( l_p \) starting from the end of \( W \). On each search iteration, for character \( ch \) in \( W \), algorithm expected to obtain an equal or narrower \([k_p, l_p]\) interval for the larger substring. Further, if the condition \( (k_p \leq l_p) \) is satisfied, \([k_p, l_p]\) interval can be defined with these specific formulas:

\[
k_p = C(ch) + O(ch, k - 1) + 1 \quad (2.2)
\]
\[
l_p = C(ch) + O(ch, l) \quad (2.3)
\]

The crucial point is to understand how \( C \) and \( O \) functions work and how it is related to calculation of interval \([k_p, l_p]\). While analyzing equations defined in (2.2) and (2.3), let us recall the
definitions for $C$ and $O$ functions:

The motivation behind Equation (2.2) is, $C(ch)$ points to the first row in the BWM with character $ch$ and $O(ch, k - 1)$ counts the number of rows starting with $ch$ that we need to skip during the search process. Major point to prove correctness of formulas, that is to prove the fact that they have been utilized to implement L-F mapping. Furthermore, any entry in last column of BWM, namely in $B$ that has a $ch$ that comes before row $k$, corresponds to a string starting with $ch$ that comes lexicographically before prefix in row $k$, due to the LF-mapping defined in Section 2.3.

Similar argumentation can be applied to Equation (2.3), in which $l$ stores the position of the last row in matrix BWM prefixed by $W[i :]$. If $W[i - 1 :]$ occurs in reference string $X$, then given $l_p$ formula updates the value of $l$, thus pointing to the last row prefixed by $W[i - 1 :]$. This can be proven from, same reasoning of L-F mapping as discussed for Equation (2.2). As $l$ is equal to the number of occurrences of character $ch$ in $B$, plus the rank of the last row, which follows position $l_p$ among all rows in BWM ending with character $ch$. This rank is obtained as $O(ch, l)$ as it represents the number of occurrences of character $ch$ in prefix of transform $B[i][12]$.

Above reasonings finalize the correctness of the Equation 2.2 and 2.3. Obtained portion of the SA, namely $W[i :]$ will be bounded by $k_p$ and $l_p$ which contain all suffixes prefixed by $W[i :]$.

In order to understand process behind function $O$, given $ch = 'G'$, observe that second, fourth and fifth rows in BWM end with $G$ as can be clearly seen from Table(2.5), while $O$ calculates occurrence of character $G$ in prefix $B[0, 1], B[0, 3]$ and $B[0, 4]$ respectively and sums up. Additionally, recall that function $C(ch)$ represents the number of characters that are lexicographically smaller than character $ch$ in $X$.

Table 2.5: Look for suffix ‘$G$’ of $W$

<table>
<thead>
<tr>
<th>$W$</th>
<th>$X$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 $AGGAGC$</td>
<td>0 $A_0G_0G_1A_1G_2C$</td>
</tr>
<tr>
<td>1 AGCS$AG$</td>
<td>1 $A_1G_2C$A$A_0G_0G_1$</td>
</tr>
<tr>
<td>2 AGGAGCS</td>
<td>2 $A_0G_0G_1A_1G_2C$</td>
</tr>
<tr>
<td>3 C$AGGAG$</td>
<td>3 C$A_0G_0G_1A_1G_2$</td>
</tr>
<tr>
<td>4 GAC$AG$</td>
<td>4 $G_1A_1G_2C$A$A_0G_0$</td>
</tr>
<tr>
<td>5 GCS$AG$</td>
<td>5 $G_2C$A$A_0G_0G_1A_1$</td>
</tr>
<tr>
<td>6 GGC$AG$</td>
<td>6 $G_0G_1A_1G_2C$</td>
</tr>
</tbody>
</table>

Note: Green - represents final correct suffixes obtained, Red - represents incompatible suffixes that are ignored, Blue - denotes correct suffix identification.

Say we look for occurrences of $W = AGG$ within $X = AGGAGC$. As BWM is sorted, rows containing $W$ as a prefix will be consecutive in BWM. We start first to look for the rows in BWM starting with shortest proper suffix of $W$, being ‘$G$’, in this case. It is important to notice that, even though complete BWM is constructed, search algorithm makes use of first column $F$, last column $B$ and rank information in BWM only. Rank utilized to point correct SA interval in BWM. It has obtained by $C(G)$ function which shows number of characters less than ‘$G$’ occurred within $X$. Therefore, function $C$ eliminates incompatible rows which start with character that differs from ‘$G$’ as represented in Table 2.5.

As a next step, let us analyze the computation of the $[k_p, l_p]$:

- As in formula 2.2, $k_p$ measured by adding $O$ - number of occurrences of character ‘$G$’ until the row ends with $B[k - 1]$, in this case until $B[0]$, $C$ - number of characters lexicographically smaller than ‘$G$’ in $X$ and one unit(for $\$”) resulting into $k_p = 0 + 3 + 1 = 4$.
- As in formula 2.3, $l_p$ measured by adding $O$ - number of occurrences of character ‘$G$’ until the row ends with $B[l]$, in this case until $B[6]$, $C$ - number of characters lexicographically smaller than ‘$G$’ in $X$ resulting into $l_p = 3 + 3 = 6$. 

Privacy-Preserving DNA Sequence Alignment 15
It is important to observe that, number of characters less than ‘G’ plus one(for $), resulting into 4, exactly tells to eliminate 4 rows starting with different characters in $BWM$ as it is sorted. Then, function $O$ by representing number of occurrences of ‘G’ before particular row tells how many rows, in this case 3, still start with suffix ‘G’. Therefore, $[k_p, l_p]$ becomes $[4, 6]$.

Next we look for all rows beginning with next longest suffix of $W$, being ‘GG’. Applying same procedure as above we get, update on $[k_p, l_p]$ as follows:

- $k_p = O(‘G’, 4 - 1) + C(‘G’) + 1 = 2 + 3 + 1 = 6$
- $l_p = O(‘G’, 6) + C(‘G’) = 3 + 3 = 6$

Therefore, new interval becomes the range $[6]$ in $BWM$. Number of characters less than ‘G’ plus one(for $), resulting into 4, exactly tells to eliminate 4 rows starting with different characters in $BWM$. Then, function $O$ eliminates incompatible rows, namely every character $G$ that has character after it, which is different than what we look for next in $W$, in this case being rows $BWM[1]$ and $BWM[3]$ by suffix $A_1$ and $C$ resulting in $G_1A_1$ and $G_2C$ respectively, in Table 2.6. Row $BWM[4]$ indicates that there is one instance where ‘G’ preceded by ‘G’.

Finally, we must find all rows beginning with final suffix AGG. New $[k_p, l_p]$ computed as follows:

- $k_p = O(‘A’, 6 - 1) + C(‘A’) + 1 = 1 + 0 + 1 = 2$
- $l_p = O(‘A’, 6) + C(‘A’) = 2 + 0 = 2$

New interval becomes the range $[2]$ in $BWM$. Number of characters less than ‘G’ - function $C$ plus one (for $), resulting into 1, tells to eliminate one row starting with different characters in $BWM$. Then, function $O$ eliminates incompatible rows, namely every character $G$ that has character after it, which differs from what we look for next in $W$, in this case being row $BWM[1]$ which has suffix $A_1G_2$ resulting in $A_1G_2C$ in Table 2.7. Finally, remaining $BWM[2]$ shows the start position of the ‘AGG’ in $X$.

To conclude, motivation behind algorithm is simply applying $L-F$ mapping iteratively in order to find the range of rows prefixed by all possible suffixes of $W$ until the range becomes empty, that is we run out of possible suffixes.

**Inexact Backward search**

This search technique concentrates on bounded traversal/ backtracking of the search query $W$ within reference string $X$. It is possible due to recursive algorithm that searches for suffix array intervals of substrings of $X$ which corresponds to the $W$ with bounded number of mismatches or differences, $z$.

Algorithm uses backward search to pattern distinct substrings from $X$. In particular, concept behind inexact search is similar to the exact search which has been defined previously in this section, except the fact that upper bound for the number of differences introduced, and algorithm tries to find substrings with maximum edit distance. An inexact search is equivalent to search for the $SA$ interval of substrings of the $X$ that match $W$, allowing $z$ mismatches. Inexact Search Algorithm defined below[18]:

**Algorithm 4 Inexact Search:** Bounded Traversal/Backtracking

1: **procedure** INEXACT SEARCH(INPUT: $X, W, z, SA, B$, OUTPUT: $I$)
   
   ▶ **Pre:** $X$: reference string, $W$: search term, $z$: upper mismatch bound, $SA$: suffix array for $X$, $B$: $BWT$ transform of $X$
   
   ▶ **Post:** $I$: set of all $SA$ intervals corresponding to $W'$, such that $X = YW'Z$, $ED(W, W') \leq z$

2: $I \leftarrow I \cup \text{Inexact Recurrence}(X, W, \text{len}(W) - 1, z, 0, \text{len}(X) - 1, SA, B)$

3: **return** $I$

Privacy-Preserving DNA Sequence Alignment
for the sequence of edit operations, namely for any \( W \) positions for \( i \). Is zero in Exact Search Algorithm. That correspond to say that, exact match only finds start
A
as no insertion, deletion or substitutions are allowed in order to transform string
B
operational cost applies, minimal number of edit operations, namely edit distance
I
for edit operations. Let us define these operations for index
Z
zero. However, inexact search bounds number of mismatches to
X
have been implemented in Algorithm 5:

**Algorithm 5 Inexact Search Recurrence**

1: procedure Inexact Recurrence \( \text{INPUT: } X, W, i, z, k, l, SA, B, \text{ OUTPUT: } I \)
   \( \triangleright \) Pre: \( X \): reference string, \( W \): search term, \( i \): index in \( W \), \( z \): upper mismatch bound, \( [k, l] \): SA interval where \( 0 \leq k \leq l \leq n \), \( SA \): suffix array for \( X \), \( B \): BWT transform of \( X \)
   \( \triangleright \) Post: Let \( R \) be the shortest string that corresponds to \( SA \) interval of \( [k, l] \) (see Def 2.3.1), \( I \): set of all \( SA \) intervals corresponding to \( W' \), such that \( X = YW'ZR \), \( ED(W[:i], W') \leq z \)
2: \( I \leftarrow \text{set()} \)
3: if \( z < 0 \) then \( \text{return } \emptyset \) \( \triangleright \) no occurrences found
4: if \( i < 0 \) then \( \text{return } ([k, l]) \) \( \triangleright \) returning interval of the actual match
5: \( I \leftarrow \text{Inexact Recurrence}(X, W, i - 1, z - 1, k, l, SA, B) \) \( \triangleright \) Insertion
6: for each \( ch \in \{A, C, G, T\} \) do
7: \( k_p \leftarrow C(ch) + O(ch, k - 1) + 1 \)
8: \( l_p \leftarrow C(ch) + O(ch, l) \)
9: if \( k_p \leq l_p \):
10: \( I \leftarrow I \cup \text{Inexact Recurrence}(X, W, i - 1, k_p, l_p, SA, B) \) \( \triangleright \) Deletion
11: if \( ch = W[i] \) then
12: \( I \leftarrow I \cup \text{Inexact Recurrence}(X, W, i - 1, z, k_p, l_p, SA, B) \) \( \triangleright \) Match
13: else
14: \( I \leftarrow I \cup \text{Inexact Recurrence}(X, W, i - 1, z - 1, k_p, l_p, SA, B) \) \( \triangleright \) Mismatch
15: \( \text{return } I \)

In order to understand the motivation behind BWT Inexact Search Algorithm, recall the definition (Equation 2.2.2), namely a variant of approximate string matching problem. It is important to observe that, algorithm finds all occurrences of \( W \) within \( X \). Therefore, given \( W \), string \( X \) can be denoted as series of substrings as \( YW'ZR \), inexact search finds all starting positions for all \( W' \) that meets requirement \( ED(W', W[:i]) \leq z \).

Recall from Section 2.2 that, edit distance between strings \( A \) and \( B \) is defined as a minimal cost for the sequence of edit operations, namely for any edit script that transforms string \( A \) to string \( B \) and otherwise.

As discussed earlier in this section, edit distance between \( W \) and possible substrings of the \( X \) is zero in Exact Search Algorithm. That correspond to say that, exact match only finds start positions for \( W \), if it exactly exists in \( X \). Edit script for exact search is empty, when \( z \) is zero, as no insertion, deletion or substitutions are allowed in order to transform string \( A \) to string \( B \). Recall that, in Levenshtein distance concept, cost for each allowed edit operations, namely for insertion, deletion and substitution is 1. As for exact search edit script is empty and therefore, no operational cost applies, minimal number of edit operations, namely edit distance is calculated as zero. However, inexact search bounds number of mismatches to \( z \), and therefore allows possibility for edit operations. Let us define these operations for index \( i \) with mismatch limit \( z \), as they also have been implemented in Algorithm 5:

- Insertion - pointer on \( W \) will be moved as some elements from \( W \) will not be considered(Line 5).
- Deletion - pointer on \( X \) will be moved as some elements from \( X \) will not be considered(Line 10).
- Match - occurs when pointer from search string \( W \) can found queried character \( ch \) in \( X \)(Line 12).
- Mismatch - or substitution operation, occurs when pointer from \( W \) can not found queried character \( ch \) in \( X \)(Line 14).

In particular, work principle of the Inexact Search algorithm based on following invariant:

In each recursive call of algorithm for particular character from set \( \{A, C, G, T\} \), either when index \( i \) of \( W \) is decreased or lower bound for mismatches, namely \( z \) is decreased, or both are decreased,
and whenever they reach zero, algorithm terminates. For each character from set \{A,C,G,T\}, Algorithm 5 executes one recursive call with values \((i-1, z-1)\), \((i, z-1)\) and \((i-1, z)\).

Penalty is achieved by subtracting one from upper bound \(z\), namely using \(z-1\) as new bound in each new iteration of the Algorithm 5. In particular, due to understand pre-condition of the Inexact Search, it is crucial to define \(R\):

**Definition 2.3.1. Shortest string** \(R\) Given reference string \(X\), a search query \(W\), \(SA\) interval \([k, l]\) and a matrix \(BWM\), \(R\) corresponds to the shortest string if:

- All rows in \([k, l]\) distance in \(BWM\) start with string \(R\)
- No other rows in \(BWM\) start with \(R\).

Pre-condition of the algorithm states that, given shortest string \(R\) that corresponds to \(SA\) interval of \([k, l]\), reference string \(X = Y W' R Z\), Inexact Search Algorithm returns the set of all \(SA\) intervals that correspond to \(R\).

However, pre or post conditions in algorithm do not give information about total number of \(z\) or number of mismatches that already have been used by algorithm. Instead, idea is to get some \(SA\) interval as an argument which corresponds to some suffix that may not match completely to \(W\) exactly in the end. Iterative procedure can be explained as follows:

In particular, in all rounds of iteration, \(R\) should be the shortest string such that suffix array \([k_p, l_p]\) starts with, and no other \(SA[i]\) in whole suffix array. This corresponds to the invariant that, \(R\) should be the shortest string that all rows in \([k_p, l_p]\) interval start with and no other rows start with \(R\). Initially \([k, l]\) interval consists whole \(X\), therefore \(R\) becomes empty as an empty string corresponds to the full suffix array where \(X = Y W' Z\). Initial case also satisfies condition that no rows outside of the \([k, l]\) interval start with empty string as there is no other rows than \([k, l]\) which cover all rows.

Inexact search is called for each edit operation (insertion, deletion, match, mismatch), thus in each call of inexact search, corresponding interval \([k_p, l_p]\) should be computed again. Furthermore, this procedure should be done for each character from \{A,C,G,T\}, that is updating \([k, l]\) interval with \([k_p, l_p]\) values, in other words computation of \(SA\) interval again. This is also proven by induction, as new \(k_p\) and \(l_p\) is calculated for each character \(c_h\) by computing \(C\) and \(O\) functions. For separate \(k_p\) and \(l_p\) in each call, we can point to separate \(R\) in each segment, same string \(R\) can be observed as prefix in all \(BWM\) rows. In exact match concept suffix of \(X\) comes exactly after one particular interval, however in inexact search, many \(k_p\) and \(l_p\)’s are computed. In order to understand the relation between string \(R\) and \([k, l]\) interval better, we can give concrete example with inexact search:

Assume inexact search for given \(W = AGG\), \(X = AGGAGC\) and \(BWM\) as depicted in Table 2.7. In initial step, string \(R\) is empty and \([k, l]\) corresponds to the all rows, namely interval \([0, 6]\).

Invariant is \(R\) should be the shortest string such that all rows in \([k, l]\) interval start with \(R\) and no other rows start with string \(R\) given \(X = Y W' R Z\).

- We start inexact search with \(i = 2\), and looking for character ‘G’. \([k, l]\) becomes interval \([4, 6]\) and \(R = 'G'\). Invariant holds, as \(R\) is the shortest string that all rows in interval \([4, 6]\) start with ‘G’ and all other rows start with some other suffixes.
- Next, we proceed on \(i = 1\), looking for the second character ‘G’ from backward in \(W\). Similarly, \([k, l]\) becomes interval \([6]\) and \(R = 'GG'\). Invariant again holds, as \(GG\) is the shortest string that row in interval \([6]\) starts with and all other rows start with some other suffixes different than \(R\).
- Finally, \(i = 0\) looking for ‘AGG’. \([k, l]\) becomes interval \([2]\) and \(R = 'AGG'\). Invariant holds again trivially and there is no other row that starts with \(R\).

It is important to observe that, on halfway of the inexact search, there exists some part of \(W\) that has been found in \(X\) with possibly some insertions and deletions, while there exists also some part of
$X$ that already has been matched. Furthermore, pre-condition of algorithm ensures that we already found some $R$. Post-condition states that, algorithm has found all $W'$ such that, $X = YW'RZ$.

As an example:

- **Pre**: $X = \text{ACGTACCC}$, $W = \text{TTA}$, $i = 1$, $[k, l] = [0, 7]$
  
  That is, algorithm looks for the second character in $W$ as a first step. $W[:i] = \text{TT}$ is the part until second character, $Y = \text{AC}$ is part of the $X$ coming before $W[:i]$.

- $W' = \text{GT}$ is the part of the $X$ that matches $W[:i]$ with $ED(\text{TT, GT}) = 1$, $R = \text{A}$ is the part of $W$ that we have already matched, and finally $Z$ is the part that whatever comes after in $X$, that is \text{CC}'.

- **Post**: We have found all $W' = \text{GT}$, such that $X = YW'RZ$ where $[k_p, l_p] = [2, 3]$

These all contribute to the $X = \text{AC- GT - A - CC}$.

In order to prove pre and post-conditions of Inexact Search Algorithm correct, we ensure that they are valid in all calls of the algorithm (i.e. insertion, deletion, match and mismatch). Given some string $R$, as $[k, l]$ interval does not change for insertion call, although prefix is extended, the same $R$ still exists. Therefore, it corresponds to say that pre-condition of algorithm is still valid for insertion where only update of $i$ and $z$ occurs. On the other hand, it affects the post-condition because as algorithm detects mismatch, it sets $z = 1$ and with affect of insertion with other calls algorithm computes new values for $k_p$ and $l_p$.

**Optimization:** In order to succeed inexact search for genomic sequences, we have done slight modifications to the original search algorithm. ‘Calculate $D$’ function is an optimization to the original BWT algorithm [18] which employed to return output faster by eliminating some fail cases. Motivation behind the function is to simply specify stop conditions by setting lower bound to the number of differences in $W[0,i]$. However, particular function was not compatible with multi-party computation, as stopping execution immediately due to some condition can break obliviousness. The number of recursions turned by search algorithm should stay the same regardless the number of mismatches.

### 2.3.4 Long-read Smith-Waterman Alignment (BWA-SW)

In recent years, a tremendous increase in development of the programs for aligning short DNA sequencing reads to a reference genome has been observed. Several next generation sequencing technologies (NGS) existing these days are essentially efficient for short read DNA sequences while not applicable or any effective for longer (> 200bp) sequences [19]. The major reason behind this problem is the fact that, algorithms (e.g. BWT transform [18]) that have been used in those programs are specifically designed for short DNA sequences with low error rate. Therefore, heavy computations of longer sequences result into significant delay and performance issues in these platforms.

However, BWA-SW has utilized FM-index data structure introduced by authors of [12] in order to find all local matches of the query in original sequence. In particular, new version of FM-index excluded the step of compressing transform $B$, function $O$ and some part of the suffix array [19]. Details about the construction of these data structures in BWA-SW are the same as explained in Section 2.3.3.

One of the major advantages of BWA-SW is, only DNA alignments which largely non-overlapping on query sequence are being reported, instead of investigating all significant local alignments. In particular, BWA-SW identifies and discards low-scoring matches while applying dynamic programming techniques around good matches only. Therefore, reduce in usage of dynamic programming scale results into faster sequence alignment [19]. However, accuracy of the algorithm is not high in comparable speed checked against the other long-read sequence alignment algorithms [17].
2.3.5 Long-read Maximal Exact Match Alignment (BWA-MEM)

BWA-MEM is one of the most powerful longer read NGS algorithms. Major advantages of the algorithm are robust error sequencing and wide scope of applicability to analyze large scale NGS data (sequence reads from 70 bp to few megabases).

In order to define motivation behind BWA-MEM, its important to mention that it essentially looks for longest exact matches that cover position at each particular search query position. It decides automatically between local and global alignment (see Section 2.2.1). Furthermore, when search query and reference string are not similar, global alignment of sequences can result into excessive mismatches and gaps. In order to tackle this problem, BWA-MEM automatically chooses local alignment and therefore, efficiently aligns sequence of longer reads at high accuracy[17].

2.4 Secure Multi-Party Computation

Secure Multi-Party Computation(MPC) considers a scenario where multiple parties want to execute distributed computational tasks of some function securely.

2.4.1 Security

As a mental experiment, security can be analyzed in terms of real versus ideal world. Given a number of entities $P_1 \ldots P_n$ with private inputs denoted as $X = x_1 \ldots x_n$, in ideal framework there is an external trusted party $I$ that helps to carry out some joint computation $Y = f(x_1 \ldots x_n)$. In this model, $I$ is not corruptible, and therefore it receives private inputs from each entity and computes prescribed output $Y$ which then revealed to all parties. As only action required by entities is to send their private inputs to the $I$, information could be exposed by an adversary that has chosen input values for dis-honest entities[6].

Moreover, basic properties for $MPC$, namely privacy and correctness can be proven as follows:

- Each entity receives only prescribed output and can learn nothing more, thus privacy holds;
- Trusted party $I$ cannot be dishonest and prescribed output is always correct, therefore correctness holds.

![Figure 2.1: Ideal World: MPC, 2-party example](image)

On the contrary, in real world example it is not possible assume unique trusted entity which is also, incorruptible. Instead, entities run protocol only among themselves without any help from any external entity, where at the end of computation, all parties learn prescribed output $Y$. 

---

20 Privacy-Preserving DNA Sequence Alignment
While considering real world example, where no trusted entity exists, MPC achieves that no adversary can carry out more harm in attack to real world computation compared to some attack to the ideal world computation. However, as successful attacks are not possible in the ideal world framework, we can therefore, conclude that successful adversarial attacks on MPC are also, not possible\[20\].

2.4.2 Motivation

Assuming malicious behavior by an external entity or even by adversarial participants, MPC avoids reveal of any important information in secure manner by important properties:

- **Privacy**: Parties should learn no additional information beyond what is absolutely necessary, namely output, even if they maliciously collude to extract more. Only information that can be conducted about other entities’ input is what can be extracted from prescribed output.

- **Correctness**: Each honest entity $P_i$ taking part in joint computation should receive it’s correct output $y_i \in Y$ such that $(y_1 \ldots y_n) = f(x_1 \ldots x_n)$ and no entity can alter this. After all honest entities obtain their outputs, computation terminates.

Above mentioned set of requirements ensure the base security of any protocol. However, as different applications of protocols have different security requirements above defined properties are general enough to ensure security in majority of the computations. In case of this thesis, set of entities who want to run protocol for function $f$ is defined as $\mathcal{P} = (P_1, \ldots, P_n)$. An adversary being active or passive, and static or adaptive, is an entity that corrupts some subset of defined parties, say $\mathcal{P}' \subset \mathcal{P}$.

- **Passive adversary** - Even though it follows the protocol as prescribed, it eavesdrops communication between entities and saves all information exchanged\[8\].

- **Active adversary** - It may also deviate from protocol and intercept, change or reroute the exchanged messages by taking full control over subset of corrupted entities\[8\].

Subset of corrupted entities are fixed in static adversary model, whereas this subset may change for an adaptive adversary model. In framework we use, honest-majority property has been considered, which is if protocol run between $n$ participants, less than $n/2$ of them should be corrupted. In particular, proposed approach in this thesis protects against passive, static adversaries with honest majority.

Privacy-preserving genomic data processing considers the problem of running data processing algorithms on highly confidential data that is not supposed to be exposed even to the party running the algorithm. MPC has many applications in data processing such as, Yaos millionaires problem \[3\] which is significant due to understand the concept behind secure computation.

One of the most sensitive application fields for MPC is the privacy-preserving database queries in health care, whereas this thesis also concentrates on this particular field. Arising problem can be defined by the scenario where multiple hospitals wish to jointly share patients’ medical data obtained for medical research, without exposing any additional information about the patient other
CHAPTER 2. PRELIMINARIES

than necessary. Expose of the highly confidential patient information can also be restricted by privacy policies of hospitals. Ability to conduct necessary information for medical research purposes without need of combining and revealing all confidential data can solve previously defined problem by providing only output of the data processing algorithm. Defined data processing ability would benefit research domain significantly, as hospitals may have greater access to a larger amount of data collected from other hospitals in secure manner.

2.4.3 Secret Sharing

In this thesis, we use TUeVIFF framework\(^5\).

Owning a secret dealer distributes a share of secret among \(n\) participants in the scheme. The motivation behind splitting secret to several pieces is the secret can be reconstructed once sufficient number of shares are available. In case when an insufficient number of shares are available, it is not possible to determine secret or any part of it. The first secret sharing schemes constructed by Shamir\(^2\) and Blakley\(^13\), independently. The secret share \([s]\) from entity \(P_i\) can be denoted as \([s_i]\). Given dealer \(D\) and participants \(P_1, \ldots, P_n\) secret sharing scheme consists of two protocols:

- **Distribution**: Dealer \(D\) shares a secret \([s]\) with \(n\) participants where each participant \(P_i\) obtains share \([s_i]\), given \(P_i \in \{P_1, \ldots, P_n\}\).

- **Reconstruction**: A secret \([s]\) is determined once sufficient number of shares \([s_i]\) are available, where \(i \in A\), for some qualified coalition of participants that allowed to reconstruct \([s]\) together, that is \(A \subseteq \mathcal{P}\).

**Linearity**: A secret sharing scheme considered linear, if the secret \([s]\) is considered as an element of a finite field, and if each share \([s_i]\) is obtained by applying a linear mapping to the \([s] and some independent random field element.

As a next step, we introduce Shamir secret sharing which is linear and protects against passive attacks:

*Shamir secret sharing* is \((T, n)\) secret sharing scheme for \(n\) number of participants for some set \(A\), whose size is bigger than given threshold \(T\). In other words, participants consist a qualified set which can be defined as \(A \subseteq \{P_1, \ldots, P_n\}\) where \(|A| \geq T + 1\). Furthermore, participants from qualified set \(A\) can reconstruct dealer’s secret \([s] \in \mathbb{F}_q\) \((q > n\) is size of field \(\mathbb{F})\), while a set of \(T\) or less can not. Efficient way to compute \([s]\) from shares of a qualified set \(A\) is possible due to the Lagrange interpolation\(^6\).

In this thesis, we use VIFF framework which is based on Shamir secret sharing scheme.

2.4.4 VIFF

The Virtual Ideal Functionality Framework, VIFF is a software library which provides a platform to implement general *MPC* protocols, in which a number of parties (three or more at the moment) execute a cryptographic protocol to do some joint computation. *MPC* protocols can be structured of basic primitives, such as addition, multiplication, comparison of secret shared values or new primitives that can be implemented. VIFF is written in Python and aims to be used by real-world participants. VIFF is asynchronous by Twisted framework, and provides possibility to players proceed whenever possible.

Programming asynchronously is a significant factor to have simpler implementation. If otherwise, say *MPC* protocol to be implemented synchronously, participant has to make sure that all messages have arrived within current round, then waiting time should be calculated properly based on the network usage. However, this makes software completely dependent from network usage.

\(^{5}\)TUeVIFF, available in: http://www.win.tue.nl/~berry/TUeVIFF/

which can cause insecurity, extra errors, etc.

VIFF works with Deferred objects, which achieve computation of the values during the procedure, as some computations (e.g., multiplication, etc.) require communication between whole participants. Deferred result is the value that eventually will be computed during execution, and method to reach that value is adding callbacks to it. Callbacks can be considered as function pointers and each deferred result have a list of these pointers. Furthermore, computing \([a] \ast [b]\) will result in Deferred object, which will not directly receive the value, instead it can be yielded which is simply waiting for the value during execution of function being computed at background, then once value received, function is executed and result can be opened. Deferred class is available due to the Twisted network library that VIFF uses[14].

**Security Assumptions**

As any other cryptographic system, VIFF is only secure when certain assumptions are fulfilled. The adversary can only corrupt as long as certain threshold \((T < n/2)\) of the total number of players \(n\) is not exceeded. In other words, in general protocol needs at least three players to tolerate one corruption for ensuring security against passive adversary. This corresponds to say that honest-majority is required between players in particular construction.

### 2.4.5 Operations

In order to understand high level interface provided by VIFF, we will describe some secure distributed computations which have been used later in this thesis. A global view of the computations can be helpful to understand the concept of ideal functionality by analyzing semantics of each operation. From many different operations available in Shamir secret sharing scheme protocols, following primitives will be used within this thesis:

- **Input**: In order to store data, each participant \(P_i\) executes \([v] \leftarrow \text{input}(i, ZP, v)\) in which \(v\) represents integer from finite field \(ZP\).

- **Open**: In order to reveal their value to a particular participant, all entities execute \(y \leftarrow \text{open}([y])\) which outputs the value of \(y\) for all \(P\). Particular open method is responsible for sending secret shares to prescribed receivers or to all participants by default. It results into exactly one exchange of shares between participants.

- **Addition**: Assume each \(P_i\) has two shared secrets as \([s_1]\) and \([s_2]\), then parties can obtain \([s_1 + s_2]\) as follows: each participant \(P_i\) only has to locally compute \(s_1 + s_2\). Therefore, this computation can be done without communication among \(n\) participants. Further, \([s_1 + s_2] = s_1 + s_2\). The cost of one addition \([s_1] + [s_2]\) considered negligibly small compared to the cost of multiplication.

- **Multiplication**: This procedure is bit more complex than addition and requires communication among \(n\) entities. To obtain \([s_1 \ast s_2] = [s_1] \ast [s_2]\), each participant \(P_i\) performs multiplication protocol, that is multiplying part of the shares result into whole multiplication.

- **Bitwise Comparison**:
  - **Less-Than-Equal (Greater-Than-Equal)**: Given two shared secrets \([s_1]\) and \([s_2]\), each participant \(P_i\) can obtain if \([s_1 \leq s_2]\) or \([s_1 \geq s_2]\) without revealing \(s_1 \leq s_2\) or \(s_1 \geq s_2\) itself. Operation yields a secret bit value, either returning 0 representing false or returning 1 representing true. Operation is more expensive than secure multiplication.
  - **Equality**: Given two shared secrets \([s_1]\) and \([s_2]\), each participant \(P_i\) can obtain if \([s_1 = s_2]\) by yielding a secret bit value, either returns 0 representing false or returns 1 representing true. Operation is more expensive than secure multiplication.
  - **Sgn**: Given two shared secrets \([s_1]\) and \([s_2]\), signum function extracts sign of the secret shared values, and returns secret number, \(-1\) when the \([s_1]\) less than the \([s_2]\), \(0\) when both secret shares are equal, and a positive number, \(1\) otherwise.
• **Equal-zero-public**: Operation is used for securely finding if secret share $x$ has the value zero, without revealing information about it, notated as $r \leftarrow [x == 0]$.

• **Lsb**: Function returns the least-significant bit $b$ of secret $[s]$ in secret-shared form, i.e. $b \leftarrow \text{lsb}([s])$. While analyzing lsb function, we observe that each call to the function needs 2 exchanges of shares (one exchange to compute least-significant bit $b$, and other one to open it’s value).

• **Dot product**: is obtained by taking entry wise product of given two secret bit vectors $[x]$ and $[y]$ of length $n$, that is, $[x_1] \cdot [y_1] + \ldots + [x_n] \cdot [y_n]$.

• **Secret Indexing**: Given public or private array $A$, it can be accessed by secret indexing which consists of converting number to the unit vector and applying inner product. Array $A$ secret indexed at $i$ can be represented as $[A_i]$ or $A_{[i]}$, if it is public. Currently, there are two different types of the secret indexing that can be used in different settings: one that works only in large field and another being less efficient, yet working with also small field of elements. Implementation of both secret indexing functions can be found in Appendix A.
2.4.6 Pinocchio-Based zk-SNARKs and Secure Adaptive Function Evaluation

 Succint Non-Interactive ARGument of Knowledge, namely SNARK can be defined as computationally sound proof which makes impossible to prove any false statement. This property is very important in security sense, for example, if prover wants to prove correctness of the certain operation to a verifier, whole process is safe even if verifier does not trust prover itself. Furthermore, malicious prover cannot trick verifier to believe that procedure is executed correctly, while it is not in fact.

Cryptographic structure of zero-knowledge proof systems ensures perfect secrecy even against an adversarial verifier with infinite computational power. Practical Pinocchio zero-knowledge SNARK allows to execute verifiable computations potentially in more efficient way than normal computation procedure. It is possible due to computation proofs which refer to the committed data, in particular two property of them: proof should be adaptive, that is being able to execute different computations on the same commitment; and zero-knowledge, that is neither commitment nor proof should reveal any information about the committed data[28].

As initialization step, let us consider the scenario that Data Provider owns dataset(data can be both private or public) while Client has computational input. The motivation behind complete verification procedure is, Client wants to ensure the correctness of the particular operation for all participants in multi-party scheme. In general, Secure Adaptive Function Evaluation procedure illustrated in Figure 2.3 can be explained by following steps:

1. Trusted party sends generated keys within respective application domains, in particular, verification key to Client, commitment key to the Data provider and evaluation key to all parties. Normally, key material generated by trusted party itself.

2. Data provider sends secret shares of the dataset(ds) to all parties. It also sends commitment to this data(ds) to the Client. Commitment to the data from Data provider can be defined as commit(ds) encryption of particular data. Major reason behind this is a need for information from data provider when Client wants to verify the correctness of the computation in the end.

3. Client sends secret shares of the private data(in) to all parties.
4. All involved parties perform the operation to be evaluated in MPC context.

5. Parties send back the output, representing whether particular operation is succeeded or not, and proof if particular output is correct. In other words, if operation is not succeeded there is no need to prove it. Even if Client does not trust involved parties, obtained zero-knowledge proof ensures correctness of the performed operation.

6. After performing proof, Client verifies proof with $commit(ds)$. Its cryptographic construction helps to succeed verification procedure and returns whether obtained result from particular operation is correct or not.
Chapter 3

Privacy-Preserving Genomic Data Processing

3.1 Related Work

Recent progress in study of human genome has introduced new privacy concerns while resulting into revolution in biomedical science. The problem of oblivious genomic data processing within MPC context could be considered relatively new research domain. Before explaining our novel approach, it would be appropriate to precisely analyze related work that has been conducted in the literature for privacy and security of the sensitive genomic records.

The proposed solutions for privacy-preserving genomic data processing can be separated into two subcategory as follows[10]:

- Private query on a private genomic data (for example, inspection for DNA mutation regarding the specific illness with a secret parameters on a private patient-specific genomic data)
- Private query on a public genomic data (for example, searching with secret parameters on a public health-care data obtained from human genome libraries)

In first case, it is assumed that, private patient-specific genomic data can reveal additional information about the data owner such as date of birth, address, physical attributes (height, weight, blood type, etc.) apart from sensitive genomic data. Several approaches have been proposed in order to tackle first problem defined above.

Prior to investigate proposed solutions for particular problem, the work of Goodrich[15] that has introduced an attack model for specific scenario (i.e. private query on private genomic data) could be analyzed. Author’s approach showed that, privacy-preserving protocols also can reveal extra information about sensitive genomic data. Introduced attack method succeeds by repetitively querying victim participant until reaching significant information from protocol output. In fact, possibility of this attack to succeed is high regarding the fact that, actual genomic data used in realistic cases can be revealed by countable number of queries. It is important to mention that, this particular attack method is also applicable in our proposed privacy-preserving approach.

First approach for oblivious genomic data processing has been introduced by Atallah et al.[1]. Original edit distance protocol has been utilized and new protocol in which neither of parties reveal any information about their private input to other parties has been developed. It is possible to identify particular similarities between this work and our approach in order to compute edit distance obliviously such as usage of the dynamic programming recurrence relation. On the contrary, our privacy-preserving protocol is applied to the specific sequence alignment algorithm (i.e. BWT[18]). One of the major drawbacks in their approach[1] is the performance inefficiency that has been resulted from security requirements in multi-party computations as conducted by authors of [16]. This problem also has been observed in our approach (see Section 3.4.2).
In their work, authors of [7] introduce a new strategy in order to preserve privacy of the Smith-Waterman DNA sequence comparison algorithm. This approach identifies similarities between compared sequences with considerable reduction in amount of false positives while performing in reasonable efficiency compared to other sequence comparison protocols. The important point to obtain in this work is the selection of the algorithm (Smith-Waterman) for sequence similarities, which is also the reference algorithm in one of the utilized BWT algorithms (i.e. BWA-SW, see Section 2.3.4). However, introduced method reveals information during a computation procedure and therefore, cannot achieve strong privacy. On the contrary, our approach achieves full privacy during both computation and protocol verification process.

In addition, Jha et al. [16] proposed privacy-preserving technique that achieves secure genomic data processing and edit distance computation on sequences. In order to do so, authors have developed three different protocols. First protocol uses Yao’s garbled circuit [3] by exploiting its internal structure in order to achieve secure circuit evaluation. Second protocol applies the method of secure computation with shares [22, Chapter 7] where participants of circuit can combine their randomly chosen shares resulting into construction of the actual bit value of the output. Finally, third protocol is developed by merging first two protocols in order to handle scalability issues and improve efficiency of the computation. However, application of this method [16] within multi-party context could be considered problematic, due to the fact that solution is not able to deal with large-scale computations.

It would be interesting to mention a scheme proposed by Kerschbaum [11] in which efficient solution to detect and mitigate attack by Goodrich [15] has been introduced. Particular method preserves the privacy of both parties while detecting similarities in genomic inputs due to the combination of two cryptographic primitives - fuzzy commitments and secure computations of edit distance. In addition, authors also contribute zero-knowledge proof which prevents client detection and ensures that same input by both parties has been used [10]. This contribution is similar to the novel approach that we have introduced in this thesis, in order to verify the correctness of the oblivious genomic data processing. On the contrary, we have used secure computations on edit distance and succinct argument of the zero-knowledge proof [28] for verification of our proposed approach.

For the second case, namely to solve the problem of private query on a public database, up to our best knowledge, there is no specifically proposed solution. However, this particular problem can be considered as part of the above defined problem, namely “Private query on private genomic database”, hence above discussed and other proposed solutions can also be directly applied to it [10].

### 3.2 BWT short-read alignment with multi-party computation

As computations on a string characters are not allowed in VIFF framework, input strings should be represented as arrays of numbers modulo some prime $P$ in order to be used in BWT algorithm. In this thesis, DNA sequences consist of characters $A, C, G, T$ which have been encoded with $1, 2, 3, 4$, respectively. Additionally, $0$ represents special character ‘$’'. This special character is not present anywhere within the sequence, but appears only at the end.

On the other hand, while obtaining results of the algorithm implemented in VIFF framework, string representation still can be required after determining output. This problem has been tackled by implementing separate functions which convert character from set of \{A, C, G, T\} to the number and otherwise, using above encoding.

#### 3.2.1 Private Inexact Search

An inexact search is equivalent to search for SA interval of substrings of the reference string $X$ that matches the search query $W$, allowing $z$ mismatches or differences. Private inexact search algorithm looks for $W$ as a private data, in the public $X$.

Private inexact search algorithm has defined below in Algorithm 6 and Algorithm 7:
Algorithm 6 Private Inexact Search: Bounded Traversal/Backtracking

1: procedure Private Inexact Search(input: X, [W], z, SA, B, output: I)
   
   
   ▷ Post: I: set of all SA intervals corresponding to W, such that, ∃Y, Z : X = YWZ, ED(W; W') ≤ z
   
2:   I ← Private Inexact Recurrence(X, [W], len(W) − 1, z, 0, len(X) − 1, SA, B)
3:   return I

Algorithm 7 Private Inexact Search: Bounded Traversal/Backtracking Recurrence

1: procedure Private Inexact Recurrence(input: X, [W], i, z, k, l, SA, B, output: I)
   
   ▷ Pre: X: reference string, [W]: search term, i: index in W, z: upper mismatch bound, [k, l]: SA interval where 0 ≤ k ≤ l ≤ n, SA: suffix array for X, B: BWT transform of X
   
   ▷ Post: let R be the shortest string that corresponds to SA interval [k, l] (see Def. 2.3.1), I: set of all SA intervals corresponding to W' such that, ∃Y, Z : X = YW'Z, ED(W; i, W') ≤ z
   
2:   I ← [0], ..., [0]
3:   if z < 0 then return I
4:   if i < 0 then
5:     for each j ∈ {k, ..., l} do
6:       [Ij] ← [1]
7:     return I
8:   I ← Private Inexact Recurrence([W], i, z − 1, k, l, SA, B) ▷ no occurrences found
9:   for each ch ∈ {A, C, G, T} do
10:      kp ← C(ch) + O(ch, k − 1) + 1
11:      lp ← C(ch) + O(ch, l)
12:     if kp ≤ lp:
13:       I ← Private Inexact Recurrence([W], i, z − 1, kp, lp, SA, B) ▷ Deletion
14:     [I] = logicor([I], [I])
15:     [equality] = (ch == W)
16:     I ← Private Inexact Recurrence([W], i, z − 1, kp, lp, SA, B) ▷ Match
17:     I ← Private Inexact Recurrence([W], i, z − 1, kp, lp, SA, B) ▷ Mismatch
18:     for each j ∈ {0, ..., n} do
19:       Ij = (1 − [equality]) * Ij + [equality] * Ij,
20:     I ← logicor([I], [I])
21: return I

Let us analyze the main differences in input and output, which makes private inexact search algorithm more secure than original version of the algorithm. In this version, only W is private input while all others remain public. Yet another point is the public set that being returned in each recursion of the original Algorithm 5, converted into secret shared bit vector in private inexact search (Algorithm 7). Union of these bit vectors obtained by simply taking OR(OR) element-wise. Handling overlap between two bit vectors, I1 and I2 can be succeeded as follows:

1. As an initial idea, we can think of simply adding I1 and I2 up entry wise. However, in overlap positions sum of entries will result in 2, which is not a bit vector anymore. Later when result is opened, these numbers will leak some information.

2. Instead, we can take entry-wise logical OR of the bit vectors I1 and I2. In order to ensure the obliviousness of this procedure, \(a + b - a \cdot b\) formula is considered for each two entry from bit vectors where a, b are shared secret entries. Operation costs only 1 multiplication for each entry while addition is for free. Therefore, new set as a union of two sets has been computed (e.g. Algorithm 7, line 14)

This is possible by simple logicor function implemented in Algorithm 8:

Privacy-Preserving DNA Sequence Alignment 29
Algorithm 8 LogicOR function

1: procedure LogicOR(INPUT: \([list1], [list2]\), OUTPUT: \([res]\))
   ▷ Pre: \([list1], [list2]\): vector
   ▷ Post: \(res\) : union of the two input lists: \(res\)
2: \(res \leftarrow [0], \ldots, [0]\)
3: for each \(i \in \{0, \ldots, \text{len}(list1)\}\) do
4:   \([res], \leftarrow [list1], + [list2], = [list1], \ast [list2],\)
   ▷ formula: \(a+b-a*b\)
5: end for
6: return \([res]\)
   ▷ union of two lists returned

In Line 11-14 of the original Inexact Search (Algorithm 5), based on the decision whether character \(ch\) from \(W\) is equal to the one in \(X\) or otherwise, we execute either match or mismatch recursion call. However, in Private Inexact Search this distinction is not allowed. In order to not reveal any information about the decision, namely if \(ch\) is equivalent to the queried character in \(W\), namely to substring \(W[i: j]\), we need to execute both recursions and continue with the rest of function.

VIFF sgn comparison function defined in Section 2.4.5, can be used in order to obtain one bit value for decision in Algorithm 7 (Line 15), namely 1 if character \(ch\) from \(X\) is equivalent to the current character \(W[i]\), and 0 otherwise. However, after detailed analysis of different comparison methods in VIFF as a case study (see Appendix B), we have introduced interpolation based comparison that might increase efficiency at least approximately by 5 times. The motivation behind comparison mainly depends on the fact that compared strings can only consist of 5 different characters, namely \(\{\$, A, C, G, T\}\). Therefore, we have introduced new \(\text{compare}_{-}eq\) comparison function:

Given all possible inputs, in order to ensure an oblivious equality comparison we define function that maintains one formula for all possible results of the comparison. This is possible due to the Lagrange interpolation formula\(^{1}\) as range for the possible inputs is small and therefore, allows efficient comparison by interpolation. First, all possible inputs \(el1, el2 \in \{\$, A, C, G, T\}\) encoded to the numbers \(el1, el2 \in \{0, 1, 2, 3, 4\}\) as TUEVIFF works only with numbers. Then, all possible input comparison decisions \(x\) and corresponding outputs \(res\) can be defined as:

\[
res = \begin{cases} 
0, & \text{if } -4 \leq x \leq -1, \text{or if } 1 \leq x \leq 4 \\
1, & \text{if } x = 0
\end{cases}
\] (3.1)

Given equation 3.1, we have used Wolfram-Alpha\(^{2}\) to generate the unique interpolation formula. Let us define \(\text{compare}_{-}eq\) function in order to succeed on character-wise oblivious comparison for equality:

Algorithm 9 Character-wise oblivious DNA comparison for equality

1: procedure compare_eq(INPUT: \([el1], [el2]\), OUTPUT: \([res]\))
   ▷ Pre: \([el1], [el2] \in \{0, 1, 2, 3, 4\}\)
   ▷ Post: \([res]\): 1 if \((el1 = el2)\), 0 otherwise
2: \([x], \leftarrow ([el1] - [el2]) \ast ([el1] - [el2])\)
3: \([res], \leftarrow ([x] - 1) \ast ([x] - 4) \ast ([x] - 9) \ast ([x] - 16)/576\)
   ▷ Interpolation formula for all possible inputs
4: return \([res]\)

Private Inexact search considers both input and output of the \(C\) and \(O\) functions as public, and therefore functions are computed normally as described in earlier sections.


3.2.2 Fully Privacy-Preserving Inexact Search

Fully privacy-preserving inexact search assumes both search query \( W \) and reference string \( X \) to be a private data. Prior to investigating the proposed approach in more detail, oblivious inverse transformation algorithm has been implemented [see Algorithm 10]. This case study has been helpful to understand secret indexing and masking techniques in order to be applied in \( BWT \) as a next step.

Algorithm 10 Oblivious IBWT

1: procedure privateIBWT\( (\text{INPUT: } [B], \text{OUTPUT: } [X]) \)
2: \( \triangleright \text{Pre: } [B]: \text{last column of the BWM, } [SA]: \text{suffix array} \)
3: \( \triangleright \text{Post: } [X]: \text{reference string without ‘$’} \)
4: \( [X'] \leftarrow [0, \ldots, 0] \)
5: for each index \( i \in \{0, \ldots, \text{len}(SA)\} \) do
6: \( [\text{target} \_ \text{index}] = [SA[i]] \)
7: \( [X'[\text{target} \_ \text{index}]] \leftarrow [B[i]] \)
8: end for
9: \( [X] \leftarrow [X[1], \ldots, X[\text{len}(X') - 1]] \quad \triangleright \text{eliminate ‘$’} \)
10: return \( [X] \)

Then, \( BWT \) have been implemented in privacy-preserving model, as can be seen from below implementation:

Algorithm 11 Private BWT

1: procedure privateBWT\( (\text{INPUT: } [X], \text{OUTPUT: } [B],[SA]) \)
2: \( \triangleright \text{Pre: } [X]: \text{vector including also ‘$’} \)
3: \( \triangleright \text{Post: } [B]: \text{transform, last column of the BWM, } [SA]: \text{suffix array} \)
4: for each \( i \in \{0, \ldots, \text{len}(X)\} \) do
5: \( [\text{tab}]_i = [X[i + 1, \text{len}(X)]] \parallel [X[0, i + 1]] \parallel [i] \) \( \quad \triangleright \text{create table of cyclic rotations} \)
6: end for
7: \( [s \_ \text{tab}] \leftarrow \text{lexicographic sort of } [\text{tab}] \text{ with } \text{comparelst} \) \( \quad \triangleright \text{sorted table} \)
8: get transform
9: \( B \leftarrow [s \_ \text{tab}]_0, \text{len}(X)-2; \ldots; [s \_ \text{tab}]_{\text{len}(X)-1}, \text{len}(X)-2 \)
10: get suffix array
11: \( SA \leftarrow [s \_ \text{tab}]_0, \text{len}(X)-1; \ldots; [s \_ \text{tab}]_{\text{len}(X)-1}, \text{len}(X)-1 \)
12: return \( [B],[SA] \)

There are two major differences between simple and privacy-preserving model of \( BWT \):

- Private SA computation: Now in private \( BWT \), although indexes of rows are public, during execution of cyclic rotations, ordering indexes in \( BWM \) results into private data, namely suffix array which then secured by secret indexing function.

- Secure comparison-based sort: Variant of the bitonic sort which exists in original VIFF documentation has been applied. The main change is the adaptation of the particular function to the matrix sorting, instead of sorting lists as in original version. Then, as the input to the function, namely characters are known to be from set \( \sum = \{A,C,G,T\} \), to make comparison more efficient interpolation has been used. As the elements are secret shares, by giving all possible comparisons as input for selected lists, collecting all possible outputs in one single formula maintains the concept of the interpolation. The possibilities for input and output can be defined in following form:

Let us define \( \text{comparelst} \) function in order to succeed on interpolation based oblivious comparison, as original bitonic sort already can be reached by TUEVIFF documentation:
Algorithm 12 Oblivious DNA comparison: sequence

1: procedure comparelst(INput: \([lst1],[lst2]\), OUTPUT: \([res]\))

\[\textbf{Pre:} \ [lst1],[lst2] : \text{sequence including 'S'}\]
\[\textbf{Post:} \ [res] : 1 \text{ if } (lst1 < lst2) \text{ lexicographically, } 0 \text{ otherwise}\]
2: \([x] \leftarrow [lst1][0] - [lst2][0] \]
\[\text{In case of equality}\]
3: \([r] \leftarrow \text{comparelst}(\([lst1][1]],[lst2][1])\) \]
\[\text{Interpolation formula for all possible inputs}\]
4: \([res] \leftarrow (70 + [r] \ast ([x] \ast [x] - 1) \ast ([x] \ast [x] - 4) \ast ([x] \ast [x] - 9) \ast ([x] \ast [x] - 16) - [x] \ast (1066 + 5 \ast [x] \ast (205 + [x] \ast (66 + 7 \ast [x]))) \rvert 40320)\]
5: \(\text{return } [res]\)

It is important to observe that, even if decision has already been made due to first character comparison between \(B\) \(W\) \(M\) rows, sort must be executed for entire row in order to ensure security further. If comparison of the elements at Line 2 of Algorithm 12 result in equality, result is ignored and algorithm recursively runs again, now considering remaining elements of both lists as input parameters, that’s having recursion (defined as \(r\) in algorithm). This happens precisely, if second list(\(lst2\) character which comparison will be held on is equal to the one from first list (\(lst1\)). This recursion can be thought of, handling equality of the characters by cutting first characters from each list, and continue with execution of sorting on remaining parts, as they can be considered as new lists. In order to effectively handle this problem, we need to fill in different variables as input of the comparison function, namely \(x\) obtains recursion of the function. As the range for the possible inputs is small, efficient comparison by one general interpolation formula can be succeeded. This is possible due to the concept of polynomial interpolation evaluation, in particular, Lagrange interpolation formula. All possible input comparison decisions(\(x\)) and corresponding outputs(\(res\)) can be defined below (recursion defined as \(r\)):

\[
res = \begin{cases} 
0, & \text{if } -4 \leq x \leq -1 \\
r, & \text{if } x = 0 \\
1, & \text{if } 1 \leq x \leq 4 
\end{cases} \quad (3.2)
\]

Given equation 3.2, we used Wolfram-Alpha to generate the unique interpolation formula represented in Algorithm 12 (Line 4).

Previously, private inexact search(Algorithm 7) considers \(C\) and \(O\) as public and calculates them as a function each time, before starting execution of recursions. However, in fully privacy-preserving inexact search, algorithm uses secret shares as input to these functions which result in secret shared \(O\) and \(C\) functions. Therefore, functions \(C\) and \(O\) should be pre-computed for all possible arguments and passed as parameters. Further, they should be defined as dictionaries, instead of calculating them as function each time before executing recursions. Let us analyze how this problem is solved:

- Recall that \(C(ch)\) function uses character comparison to compute lexicographically smaller elements for each secret \(ch\) in private reference string \(X\). Element-wise comparison (\(compare\_el\)) is defined specifically for \(C\) function, as it gets only one secret shared element \(ch \in \{S, A, C, G, T\}\) as an input value. Given all possible inputs, in order to achieve oblivious compare we define function such that maintains one formula for all possible results of the comparison. This is possible due to the concept of polynomial interpolation evaluation, in particular Lagrange interpolation formula. As range for possible inputs is very small, efficient comparison by interpolation can be succeeded. First, all possible inputs \(el1, el2 \in \{S, A, C, G, T\}\) correspond to the numbers \(el1, el2 \in \{0, 1, 2, 3, 4\}\) as TUeVFF works only with numbers. Then, all possible input comparison decisions(\(x\)) and corresponding outputs(\(res\)) can be defined as:

\[
res = \begin{cases} 
1, & \text{if } -4 \leq x \leq -1 \\
0, & \text{if } x = 0 
\end{cases} \quad (3.3)
\]
Given equation 3.3, we have used Wolfram-Alpha to generate the unique interpolation formula. Let us define \( \text{compare}_{-el} \) function (Algorithm 13) in order to succeed on character-wise oblivious comparison:

**Algorithm 13 Character-wise oblivious DNA comparison**

1: procedure \( \text{compare}_{-el} \)(INPUT: \([el1],[el2]\), OUTPUT: \([res]\))

\(\triangleright\) Pre: \([el1],[el2]\) \(\in\) \(\{0,1,2,3,4\}\): representing \(\{S,A,C,G,T\}\)

\(\triangleright\) Post: \([res]\): 1 if \((el1 < el2)\), 0 otherwise

2: \([x] \leftarrow [el1] - [el2] \)

3: \([res] \leftarrow (x-4) \ast (x-3) \ast (x-2) \ast (x-1) \ast (0-x \ast (1066 + 5 \ast x \ast (205 + x \ast (66 + 7 \ast x)))) / 40320 \)

\(\triangleright\) Interpolation formula for all possible inputs

4: return \([res]\)

In order to understand motivation behind \(C\) function, let us analyze Lines 2-5 of Algorithm 14. After initialization of \(C\) as dictionary in Line 7, algorithm updates the values of \(C\) function corresponding to keys from character set \(\{S,A,C,G,T\}\). In order to tackle this problem, in Line 5 value of \(C\) is updated with comparison result of the \(\text{compare}_{-el}\) function. This finalizes computation of the alphabetically smaller elements than \(ch\) in secret shared \(X\).

- Comparison (\(\text{compare}_{-eq}\)) is used for \(O\) function, as it gets secret shared element \(ch \in \{S,A,C,G,T\}\) and start point of the SA interval, namely \(k\) as input values. Recall that, \(O\) function simply computes the number of occurrences of input character \(ch\), in BWT transform, in particular with given index \(k\), namely \([B[0:k]]\).

In order to understand motivation behind \(O\) function, let us analyze Lines 6-11 of Algorithm 14. After initialization of \(O\) as dictionary in Line 7, algorithm sets the count and increments it with comparison result of the \(\text{compare}_{-el}\) function. In particular, secret variable \(count\) computes number of occurrences of \(ch\) in secret shared transform \(B\) and updates value of the function \(O\) due to the corresponding key accordingly in line 11.

Let us also define full private inexact search algorithm with Algorithm 14 and Algorithm 15:

**Algorithm 14 Full Privacy-Preserving Inexact Search: Bounded Traversal/Backtracking**

1: procedure \(\text{INEXACT SEARCH}\)(INPUT: \([X],[W],z,[SA],[B]\), OUTPUT: \([I]\))

\(\triangleright\) Pre: \([X]\): reference string, \([W]\): search term, \(z\): upper mismatch bound, \([SA]\): suffix array for \(X\), \([B]\): BWT transform of \(X\)

\(\triangleright\) Post: \([I]\): set of all SA intervals corresponding to \(W\), such that \(X = YWZ\), \(ED(W,W') \leq z\)

\(\triangleright\) C function as dictionary

2: for each \(ch \in \{1,2,3,4\}\) do: \([C_{ch}] \leftarrow [0]\)

3: for each \(char \in \{X_0, \ldots, X_{|X|-1}\}\) do

4: for each \(ch \in \{1,2,3,4\}\) do

5: \([C_{ch}] \leftarrow [C_{ch}] + \text{compare}_{-el}(ch, [char])\)

6: for each \(ch \in \{1,2,3,4\}\) do

\(\triangleright\) O function as dictionary, keys length of \(B\)

7: \([O_{ch}] \leftarrow [0], \ldots, [0]\)

8: \(count = 0\)

9: for each \(j \in \{0, \ldots, \text{len}(B)\}\) do

10: \([count] \leftarrow [count] + \text{compare}_{-eq}([B][j], [ch])\)

11: \([O_{ch}] \leftarrow [count]\)

\(\triangleright\) store value for \(ch\)

12: \([I] \leftarrow \text{Full Inexact Recurrence}([X],[W], \text{len}(W)-1,z,[0],[\text{len}(X)]-1,[SA],[C],[O])\)

13: return \([I]\)

In order to investigate above defined algorithm in more detail let us highlight several breakpoints:
Algorithm 15 Full Privacy-Preserving Inexact Search Recurrence

1: procedure Full Inexact Recurrence (INPUT: $[X], [W], i, z, [k], [l], [SA], [C], [O]$,
2: OUTPUT: $[I]$)
4: functions by Algorithm 14
5: $\triangleright$ Post: $: if k < l$, let $R$ be the shortest string that corresponds to $SA$ interval of $[[k,l]]$
6: (see Def. 2.3.1), $[I]$: set of all $SA$ intervals corresponding to $W'$ such that $X = YW' RZ$, $ED(W[i], W') \leq z$
7: $[I] \leftarrow \{0, \ldots, [0]\}$
8: if $z < 0$ then return $[I]$ $\triangleright$ no occurrences found
9: if $i < 0$ then:
10: for each $j \in \{0, \ldots, [n]\}$ do
11: $[cmp1] \leftarrow \{ j \geq [k] \}$
12: $[cmp2] \leftarrow \{ j < [l] \}$
13: $[I]_j = [1] \ast ([cmp1] \ast [cmp2])$ $\triangleright$ store binary decision 1
14: $\triangleright$ returning interval of the actual match
15: return $[I]$
16: $\triangleright$ Insertion
17: $[I] \leftarrow$ Full Inexact Recurrence($[W], i-1, z-1, [k], [l], [SA], [C], [O]$)
18: for each $ch \in \{A, C, G, T\}$ do
19: $[res1] = [O_{\text{ch}}][k] - 1$
20: $[res2] = [O_{\text{ch}}][l] - 1$
21: $[k_p] \leftarrow [C_{\text{ch}}] + 1 + ([res1] \text{ if } i = \text{len}(W) - 1 \text{ else } 0)$
22: $[l_p] \leftarrow [C_{\text{ch}}] + [res2]$
23: $[cmp3] = \{ [k_p] \leq [l_p] \}$ $\triangleright$ store binary decision 3 for $k_p$ and $l_p$
24: $\triangleright$ Deletion
25: $[I_1] \leftarrow$ Full Inexact Recurrence($[W], i-1, z-1, [k_p], [l_p], [SA], [C], [O]$)
26: $[equality] = (ch == [W][i])$
27: $\triangleright$ Match
28: $[I_2] \leftarrow$ Full Inexact Recurrence($[W], i-1, z-1, [k_p], [l_p], [SA], [C], [O]$)
29: $\triangleright$ Mismatch
30: $[I_3] \leftarrow$ Full Inexact Recurrence($[W], i-1, z-1, [k_p], [l_p], [SA], [C], [O]$)
31: for each $j \in \{0, \ldots, [n]\}$ do
32: $[I_4]_j = (1 - [equality]) \ast [I_3]+ [equality] \ast [I_2]_j$
33: $[I_5] \leftarrow \text{logior}([I_1], [I_4])$
34: for each $j \in \{0, \ldots, [n]\}$ do
35: $[I_6]_j = [cmp3] \ast [I_5]_j$
36: $[I] \leftarrow \text{logior}([I_1], [I_6])$
37: return $[I]$
While analyzing stop conditions of the algorithm, it can be clearly observed that, in order to point all SA interval, namely \([k, l]\) as a final result, all character comparisons should be made. In previous versions of inexact search algorithm, this interval can be determined by succeeding two comparisons in public string \(X\) (e.g. Algorithm 7, Line 5-6). However, as in new full private version \(X\) is also private, SA interval, \(k\) and \(l\) should be the secret shares, therefore they cannot be opened during the function execution. Therefore, comparison should be held on all elements of \(X\) in order to keep obliviousness. That is, checking every entry one by one, and to put all 1’s in \([k, l]\) distance in bit vector (Algorithm 15, Line 6-8).

In Line 12-13, pre-computation step for \(k_p\) and \(l_p\) formulas, namely computation of \(O\) function for both \(k\) and \(l\) succeeds. As \(k_p\) and \(l_p\) values become secret shares during the execution of the algorithm, comparison on these values is stored as binary decision, at \(cmp3\) (see line 16, Algorithm 15). Later in algorithm, we apply this binary decision value to all elements of a result set, thus it does not reveal any information about private values (see line 25).

### 3.3 Correctness Proof and Adaptive Function Evaluation

Goal of the proof function is to evaluate the correctness of the private inexact search (Algorithm 7) and full private inexact search algorithm (Algorithm 14) results. In order to tackle this problem, extra information about the computation is needed. Prior to investigating the problem in detail, number of edit operations, namely match, mismatch, insertion and deletions have been computed. These operations represented with numbers 0, 1, 2, 3, respectively in order to be stored in \(edit\ script\). Additionally, we used 4 for \(dummy\) value, which is also stored in \(edit\ script\). In other words, \(edit\ script\) contains full sequence of operations including also match and mismatches rather than changes on reference string. As an example, \(edit\ script = [0, 1, 4, 0, 2, 3, 4]\) can be explained as sequence of following edit operations: \([\text{match, mismatch, dummy, match, insertion, deletion, dummy}]\).

In a pre-processing phase of proof, other variables which are computed during execution of the private inexact search algorithm have been used. These variables are not represented explicitly in Algorithm 7 but defined separately as follows:

- **\(res\)**: represents the bit vector of the locations resulted from each recursion (insertion, deletion, match, mismatch) in Inexact Search Algorithm. Afterwards, it combined with other recursion results with \(logor\) function as defined in Algorithm 8.

- **\(pos\)**: corresponding to the index in \(X\), which shows from where to start executing the algorithm. From relation with suffix array and \(X\), \(pos\) will exactly show the position where \(W\) should be obtained in \(X\). As \(pos\) represents position of the \(match\) if it occurs, initial value for it in algorithm is \(k\), representing location of the first match. Estimated \(pos\) value from each recursion then combined with results from other recursions (insertion, deletion, match or mismatch) with \(b\)-trick. That is, combining bit result of the two computation in order to keep obliviousness of comparison result.

- **\(found\)**: corresponds to value 0 or 1 showing respectively, whether match ever occurred or not. In base cases of Inexact Search, namely when \(i < 0\) and \(z < 0\) (see Line 3-4, Algorithm 7) \(found\) has been initialized as 0 (match not found) and 1 (match found) respectively. Each time when match occurred, \(found\) result is tabulated together with corresponding recursion result. Estimated value from each recursion then combined with results from other recursions (insertion, deletion, match or mismatch) with \(logor\) (see Algorithm 8) method. Decision values obtained from last recursions, namely match and mismatch are combined with \(b\)-trick.

- **\(edit\ script\)**: corresponds to the ordered sequence of the edit operations. Length of \(edit\ script\) can be formulated as follows:

\[
\text{len}(edit\ script) = \text{len}(W) + \text{num}(deletions) \tag{3.4}
\]
of the all computations that took place and contains exact number of equations to be checked. In
Proof (See Algorithm 16) which works with both private and full private inexact search algorithms.

One major problem with edit script is that, it reveals information about the number of deletions occurred during the process (see Equation (3.4)). In order to tackle this problem, maximal length of the edit script as defined in Equation (3.6) is always returned. In particular, for each mismatch or insertion, respectively ‘1’+‘4’(dummy) or ‘2’(insertion)+’4’(dummy) have been added to edit script. Dummy character helps to keep balance of edit operations resulted from different recursions. In particular, the motivation behind Equation 3.4 is the fact that, only in deletion algorithm executes on the same character within W and termination of the algorithm succeeds only when it reaches the end of W. On the contrary, in insertion comparison stays in the same position within X and proceeds to the next character in W. As required, for each match or deletion, respectively ‘0’ or ‘3’ is added to edit script.

In addition to the Equation 3.4, edit script formula should be bounded by base cases of the inexact search algorithm, namely when $i < 0$ and $z < 0$ (see Line 3-4, Algorithm 7). Following initialization has been made for edit script before any recursion is executed:

\[
len(\text{edit script}) = i + z + 2 \quad (3.6)
\]

The trick (adding 2) covers the bound case in which both $i$ and $z$ can get value of ‘-1’. In other words, when $z < 0$, exact length of the algorithm result should be initialized with all 0’s. Hence, not being able to know exact length of the edit script cannot cause any problems, as due to this trick exact number of zeros for each character can be initialized. In particular, Equation 3.6 always initializes dummy character(4) while starting first recursion of the algorithm. The major reason behind this is, while executing proof function it will proceed to next edit operation, namely insertion, deletion, match or mismatch when having dummy in sequential order of edit script data.

In particular, if edit script and pos have been given, proof algorithm can check whether W occurs in X with same manner and not return anything as a result. In order to understand work principle of the proof function a counter example can be helpful:

- Given function proof($X$, $W$, pos, edit script) with following parameters: $X=\text{ACCACG}$, $W=\text{AGG}$, pos=3, edit script=[0,1,0]. Assumed that proof function does not return any errors.

- Correctness proof can be explained as follows: Firstly, we start from position 3 in X, namely character ‘A’ (see X=“\text{ACCACG}”). In search query, first character is also an ‘A’ (see $W=\text{AGG}$), comparison results in match, namely 0 which complies with edit script (see [0,1,0]). We then continue with second character in W, namely ‘G’, pos also increases by one as we want compare the next element in X, namely ‘C’. This comparison results in mismatch which also matches with edit script (see [0,1,0]). Continuing in this manner pos increases again, as last step we compare last character of W, that is ‘G’ with next character in X, namely ‘G’ (see “\text{ACCACG}”). Comparison results into the match that complies with edit script also (see [0,1,0]). This last step verifies the correctness of the proof function, by finding W in X by given pos and edit script.

However, if one of the possible error cases occurs, algorithm returns the information error occurred and therefore, W can not be detected within X. Let us define the implementation of the Correctness Proof (See Algorithm 16) which works with both private and full private inexact search algorithms.

We used Trinocchio[25] as a framework, in which proof is achieved by log file that keeps record of the all computations that took place and contains exact number of equations to be checked.
Algorithm 16 Proof function

1: procedure Proof(INP: [X], [W], [pos], [edit_script])
   ▷ Pre: X: reference string, W: search term, pos: index in X, edit_script: log of the edit operations,
   ▷ Post: In case that search query W can not be obtained within X given edit_script, call assert
   2: iserror ← 0 ▷ initially, no mistakes so far
   3: [i] ← 0 ▷ index in W
   4: for each \( j \in \{0, \ldots, \text{len}(\text{edit_script})\} \) do
   5: \([x] \leftarrow \text{edit_script}[j]\) ▷ Correspondence: 0-match, 1- mismatch, 2- insertion, 3- deletion
   6: \([\text{match}] \leftarrow ([x] - 1) * ([x] - 2) * ([x] - 3) * ([x] - 4) / 24\)
   7: \([\text{mismatch}] \leftarrow [x] * ([x] - 2) * ([x] - 3) * ([x] - 4) / (-6)\)
   8: \([\text{insertion}] \leftarrow [x] * ([x] - 1) * ([x] - 3) * ([x] - 4) / 4\)
   9: \([\text{deletion}] \leftarrow [x] * ([x] - 1) * ([x] - 2) * ([x] - 4) / (-6)\)
   10: \([\text{is-error}] = [\text{is-error}] + [\text{mismatch}] + [\text{insertion}] + [\text{deletion}]\)
   ▷
   11: assert \([\text{is-error}] - (z + 1) \neq 0\) ▷ error check
   12: \([s_x] \leftarrow [X][\text{pos}]\) ▷ secret indexed X for pos
   ▷
   13: assert \((i - \text{len}(W)) \neq 0\) ▷ error check
   14: \([s_w] \leftarrow [W][i]\) ▷ secret indexed W for i
   ▷
   15: \([\text{res}] = \text{compare_eq}(s_x, s_w)\) ▷ oblivious comparison for equality between W and X
   16: \([a] = [\text{match}] * (1 - [\text{res}])\)
   17: \([b] = [\text{mismatch}] * [\text{res}]\)
   18: \([\text{new-error}] \leftarrow [a] + [b] - [a] * [b]\) ▷ logicOR technique due to combine error cases
   19: assert \(([\text{new-error}] = 0)\)
   ▷
   20: \([\text{inc}_X] \leftarrow [\text{match}] + [\text{mismatch}] + [\text{deletion}]\) ▷ increase in index of X
   21: \([\text{inc}_W] \leftarrow [\text{match}] + [\text{mismatch}] + [\text{insertion}]\) ▷ increase in index of W
   22: \([\text{pos}] \leftarrow [\text{pos}] + [\text{inc}_X]\)
   23: \([i] \leftarrow [i] + [\text{inc}_W]\)
particular, log file is generated during the proof function. On the other hand, performing inexact
search generates computation results, i.e. edit script, pos which can be converted then to the verifi-
able computation shares together with W, X. Furthermore, every computation that relates to these
converted arguments will be part of log file which gives an instruction to check correctness of these
computations. In other words, when we verify particular function for each party in MPC scheme,
we should verify for every step of computation existing on the log file. There will be a cryptographic
zero-knowledge proof ensuring the correctness of each step in the log file. Due to this proof and
provided log file, all involved parties and others can check correctness of sequence of computations
in log file.

Arguments used in proof function are verifiable computation shares, instead of the secret shares.
The major difference is, for each computation performed for verifiable computation shares, there
should be a proof ensuring correctness of the particular computation by using Pinocchio Adaptive
Function Evaluation as defined in Section 2.4.6. In order to explain the major idea behind this
approach, let us analyze steps in Figure 2.3 adapted to the scenario in which performed operation
by entities is private inexact search. It is important to observe that, computed values during the
execution of different steps in proof function are unknown, whereas computation steps themselves
are known to any verifier/trusted party. Therefore, during function’s initial phase, evaluation and
verification key is generated based on these computation steps and used to build entire proof.

Based on inexact search scenario, in Figure 2.3, at Step 2 dataset(ds) sent by Data Provider as
secret shares is a reference string X. Accordingly, computational data(in) is a search string W which
sent by Client to all parties as secret shares at Step 3. Then, as a next phase all involved parties in
MPC scheme perform private inexact search operation (Step 4), and afterwards send corresponding
response to the Client including edit script and pos that either to be opened to the Client or kept
closed, representing whether search succeeded or not (Step 5). If search could not succeed, namely if
W cannot be found within X, no proof is given. However, if search succeeds, verification of the proof
is achieved by Client with respect to commitment to X, namely commit(X). This step finalizes
zero-knowledge proof function by obtaining exact correctness of the inexact search. To conclude, if
correctness of the performed operation has been proven, Client learns that it succeeded. Otherwise,
Client is not able to learn whether edit script has been consisted of wrong sequence of operations
or W is not obtained within X at all.

Next, let us explain how interpolation is used in order to compare match, mismatch, insertion,
and deletion values from edit script and store during execution of the proof function. Similar rea-
soning of Lagrange interpolation formula as in Equation 3.1 has been used. In particular, range
for possible inputs from edit script is {0,1,2,3,4} corresponding to the match, mismatch, insertion,
deletion and dummy respectively. Then, all possible input comparison decisions can be arranged
by polynomial interpolation formulas represented in line 6-9 in Algorithm 16.

In particular, proof function is required as a verification in a framework such as Pinocchio[28].
Therefore, proof function step-by-step checks if inexact search algorithm result can be found as its
expected, and calls an assert in case of any error. While executing proof function to verify work of
the private inexact search algorithm on DNA sequences, we demonstrated Algorithm 16 that covers
all possible error cases. In order to tackle this problem, it’s important to consider that if number
of insertion or deletions is greater than allowed mismatch bound, whole proof function should call
assert corresponding that there exists an error. As given particular edit script, e.g case where all
operations included in edit script are insertions, output of the proof function will always be nothing,
i.e. no errors, however these false cases should also be covered by algorithm. Following cases can
be handled by setting count(is_error) and increment it each time when an insertion, mismatch or
deletion occurs(Line 10, Algorithm 16).

At Line 12-14 of the Algorithm, secret indexing function(see Appendix A) has been applied to
the both private X and W, due to get value from post location of the X and i-th location of the
W, respectively. At Line 15, res assigned due to check equality of s_x and s_w which correspond to
secret indexed values from X and W, respectively. Accordingly, when the result of the compare_eq
function, namely \( res \) is zero, that means, \( s_x \) and \( s_w \) are not equal to each other and therefore, mismatch occurs. Continuing with algorithm, in Line 16 if the case \( \text{match} = 1 \) representing \( s_x \) and \( s_w \) are equal, and \( res = 0 \) occurs, it results into contradicting statements and therefore, corresponds to an error case. Same applies to Line 17 where, another error case can occur: \( \text{mismatch} = 1 \) and \( res = 1 \), representing \( s_x \) and \( s_w \) are not equal and equal, respectively at same time, this corresponds to say that, there is indeed an error. In order to keep obliviousness, both error cases are combined by \text{new_error} \) and if it’s value is different than zero whole function calls \text{assert} \), corresponds to say that there exists an error.

All in all, proof function does not return any value if there is no error, and therefore, it corresponds to state that \( W \) occurs in a manner of given \text{edit script} in \( X \). If proof function calls an \text{assert} \) in any stage of the execution, that means particular manner of \text{edit script} cannot return queried \( W \) within \( X \). In addition, following considerations have to be taken into account, during proof algorithm execution:

- Condition checking whether end of the search query (\( W \)) has been reached or not;
- Condition checking whether \( pos \) and \( i \) exceed limits, namely \( X \) and \( W \), respectively.

It is appropriate to mention that, due to the time constraints second consideration above is not implemented fully which can result into some exceptions in execution of the algorithm.

### 3.4 Discussion

#### 3.4.1 Optimizations

Let us first give a small summary of the major optimizations considered in this thesis for both private and full private inexact search algorithms. Implemented optimizations will include tabulation in order to avoid repetitive identical computation, utilized secret index function (see Appendix A) to be able to work with both small and large fields, pre-computed \( C \) and \( O \) functions, interpolation based comparison. Additionally, in theory we will discuss combining all \( SA \) intervals, namely output of the private or full private inexact search algorithm as another optimization.

- **Tabulation**: is the bottom-up dynamic programming technique that starts from bottom, and cumulates the values to the top. The motivation behind this concept is, basically to remove unnecessary calculation by computing the values needed for next step.

  Algorithm 11 often calls recursions with the same parameters and follow the same pattern. Therefore, at the end we can store the result in dictionary sequentially and access computed entries from the dictionary in any later stage of the algorithm. Overall, it is possible to conclude that the tabulation approach essentially reduces the time exposed to the execution.

  To succeed at tabulation in full private inexact search, in which some of the above considered parameters ([\( k, l \)]) are also private, in order to compute and access particular parameters from the table secret indexing should be used. Therefore, considering possible performance inefficiency risk, tabulation is not an optimization for full private inexact search algorithm.

- **Small & Large Field**: In general, VIFF framework performs faster if small field is used instead of the some large field. It is important to observe that, for example, in private inexact search algorithm for DNA sequences, search string \( W \) (consists of characters from \{A, C, G, T\}) and bit vector \( I \) (consists of only 1 or 0's), are only secret shared values (see Algorithm 7).

  Hence, it approves the fact that private inexact search algorithm can be executed over small field of size 5 as an optimization.

  For full private inexact search algorithm, we need slightly larger field in order to execute computations on private \( C, O \) functions and \( X, W, [k, l] \). As it can be clearly observed, the largest value that field can get is the size of the reference string, namely \( \text{len}(X) \). Therefore, during the implementation of the algorithm small field at least in size of \( X \) still can be used to achieve better efficiency. Additionally, we can use secret indexing both for computing and obtaining values of pre-computed private \( C \) and \( O \) functions. This process decreases efficiency...
of the algorithm in small field, hence makes optimization ineffective. Major reason behind this is a construction of the secret index function (can be reached at Appendix A) which performs more efficiently in large field.

- **Interpolation based comparison**: Due to the fact that, comparison of secret shared values is considered as an expensive operation in VIFF, we make use of the advantage that possible input characters for comparison are in small range of \{A, C, G, T\}. Hence, interpolation can achieve the most efficient comparison by assigning corresponding decision values for each possible type of inputs.

**Further Optimizations**

- **Output locations**: As it has been stated earlier, private inexact search outputs secret shared bit vectors as a result of each recursion during the execution of algorithm. In order to handle overlap between these bit vectors, instead of taking union of them, idea of adding them up element-wise can be considered. The problem with this idea is the information leakage while opening the result.

  However, adding up all interval values for output \(I\), then applying *equal-zero-public* function from VIFF solves this problem as follows: it checks if result is greater than zero without giving an exact amount. This approach is significantly more efficient compared to using *logicor* which needs communication of entities. Although this method keeps obliviousness, it can be used only if user wants to execute search algorithm and open the result directly after it. If \(I\) intended to be used afterwards, for example in proof function as a private input, in order to obtain union of secret bit vectors, *logicor* method is more useful and secure.

**3.4.2 Performance Analysis**

Prior to analyze both sequence comparison algorithms, namely private and full private inexact search in multi-party context, it’s important to observe the difference compared to the original versions in which VIFF runs with one-party (used only for testing purposes). As the sequence of the executed operations remain the same in both configurations (i.e., one party and multiple parties), linear connection between them has been expected. The main difference of MPC involved algorithm is, as we have multiple entities which execute algorithm, some messages between them need to be exchanged. Surprisingly, the relation between private (see Figure C.2) and MPC involved private (see Figure C.6) inexact search in same \(X\) length of 10 did not appear to be linear.

**Reference string: length and variance**

Performance evaluation of the sequence comparison algorithms, namely private and full private inexact search for reference string \(X\) with different lengths can give an idea about overall efficiency of the proposed multi-party approach. In order to understand this relation, firstly full private inexact search results in graph C.7 (where \(X\) is length of 10) and graph C.8 (where \(X\) is length of 20) can be compared. Due to the additional time used for secret indexing that have to be done for entire genomic sequence \(X\), inexact search in longer \(X\) takes more time to succeed.

From this perspective, another key point to consider is the number of different substrings that exist in \(X\). It is important to observe that, the number of recursive calls in inexact search algorithm corresponds to the size of the suffix tree that has been constructed during process. Variance in characters of \(X\) affects efficiency of private inexact search algorithm significantly. For example, if we analyze private inexact search in \(X\) length of 20 and consists of character \(A\)’s only, number of the different substrings will be few. As a result, algorithm will perform faster (see Figure C.4) compared to the case where \(X\) has same length but consists of random characters (see Figure C.3).

Furthermore, private inexact search algorithm tries to match substrings that actually occur in \(X\). In particular, Algorithm 7 states that if \(W[i:]\) already matched at some point during the algorithm, inexact search will look for some substring \(W[i-1:]\) only if it occurs within \(X\). This procedure is ensured by \((k_p \leq t_p)\) check which is important for obtaining correct SA interval in
Algorithm 7 (Line 12).

While reasoning about above consideration in full private inexact search, all substrings generated in each round of the suffix tree should be checked. Major reason behind this is a fact that, as both $X$ and $W$ are unknown in full private inexact search algorithm, substrings that were not fit with search string in previous rounds cannot be eliminated further. Algorithm 15 tries to match all substrings regardless of whether they occur in $X$ or not.

Moreover, computations on full private inexact search are held on one-party variant of the VIFF framework due to the performance inefficiency of the MPC involved one. Although Figures C.7 and C.8 do not give clear idea about the real multi-party context, performance results are still interesting in order to do comparison with private inexact search under the same configuration (see Figures C.2 and C.3, respectively). As might have been expected, increase in number of oblivious computations result into the performance loss in full private inexact search compared to the private inexact search.

Additionally, it’s appropriate to mention that increase in length of $X$ may result in more diversified substrings in real-life scenarios. Therefore, length of the genomic sequence is indirectly related with execution speed of the algorithm.

Upper mismatch bound and search query length

Two major breakpoints in inexact search algorithm are mismatch bound $z$ and length of the search query $W$, namely $\text{len}(W)$. Hereby, we represent efficiency analysis for both private and full private versions of inexact search assuming different $z$ and $W$’s:

Private inexact search algorithm has to be executed until end of the search string regardless the result, in other words whether any part of the $W$ has obtained (match) or not (mismatch). Thus, execution time of the algorithm highly dependent on only length of $W$, instead of its value.

As it can be observed from Algorithm 7, inexact search algorithm includes several branches. Furthermore, algorithm adds extra recursions (e.g. insertion, deletion) during execution unless mismatch limit is reached, i.e. $z = 0$. Another important conclusion is, we have expected computational complexity to be linear when $z = 0$ as it ensures that algorithm will not branch, namely not execute any edit operation and terminate instantly. As it can clearly seen from Figure C.3, expected linear relation for different search queries and fixed $X$ has been obtained essentially.

Any increase in length of $W$ and $z$ expected to affect number of recursions to be executed due to the branching. Hence, it also has expected to increase the time exposed to execution of the algorithm. This reasoning can be demonstrated in several graphs (e.g. Figure C.3, Figure C.8), as increase in $z$ and $W$ slow down both (private and full private) inexact search algorithms considerably.

One of the derived conclusions is, private inexact search time has been expected to be roughly exponential when $z > 0$. The motivation behind is the fact that Algorithm 7 is not expected to execute all recursions - edit operations such as insertion or deletions, and therefore, terminate instantly in case of reaching upper mismatch bound. However, at least half of the recursions still need to be executed during lifecycle of the algorithm. This is due to the branching in which algorithm decides to either execute match or mismatch. On the other hand, performance of the full private inexact search algorithm expected to be exponential, as it have to go through all recursions regardless of the mismatch bound or search query. It can be observed from both graphs representing private (e.g. Figure C.3) and full private (e.g. Figure C.8) inexact search that, obtained results are likely to meet expectations.

We have analyzed efficiency of private inexact search for reference string $X$ length of 10 and 20, in Figure C.2 and Figure C.3 respectively, while considering several $z$ and $W$’s of different length. However, it’s difficult to derive conclusion about how increase in $z$ and $\text{len}(W)$ exactly affects performance of the algorithms. There are still two main conclusions can be reached:
• if \( z \) is fixed, longer \( W \) is, algorithm takes more time to execute search. The reason is algorithm branches more, as number of substrings to be searched for increase accordingly.

• if \( \text{len}(W) \) is fixed, larger the bound \( z \) is, algorithm branches more for edit operations, resulting into the increase of time exposed to search.

While analyzing full private inexact search (Algorithm 14), observations from Figure C.7 and Figure C.8 have resulted into two main conclusions:

• if \( \text{len}(W) \) is fixed, increase in \( z \) results into more linear computational complexity while slowing down efficiency of the entire algorithm for same length of \( X \).

• if \( z \) is fixed, longer the search query \( W \) is, algorithm takes more time to execute search. The reason is, as \( X \) becomes private, algorithm looks for all possible substrings without eliminating the cases that cannot occur as mentioned earlier.

In addition, variance in characters of \( W \) had no relation with efficiency of the algorithm while testing procedure, as might have been expected. The reason is the fact that, both inexact search algorithms execute all recursions until whole \( W \) is searched for.

Small & Large Field Interpolation

As interpolation is the most efficient way to do comparison given fact that possible input range is small, we further have analyzed whether running VIFF over small or large field can affect the performance of the algorithm as micro-benchmark. Prior to investigate problem in more detail, we have used comparison by interpolation, particularly. As a consequence, interpolation has resulted into the most efficient private inexact search algorithm compared to the different comparison methods as can be determined from the graph B.1 in Appendix B. Another interesting point is, interpolation for large primes appeared to be faster than the extended one for small primes during the tests, where opposite has been expected. Details of the efficiency analysis can be reached in Appendix B.

However, its appropriate to state that, if all types of input are considered, algorithms cannot make use of interpolation approach as it might get slower and more difficult to construct. In general, if all different types of comparisons available in VIFF framework are considered, polynomial interpolation is the most efficient method for sequence comparison which can be proven by an efficiency analysis we conducted at Figure B.1.
Chapter 4

Conclusion

The study was set out to enable practical privacy-preserving analytics on shared Big Data, in particular on genomic data processing by exploring applicability of MPC techniques without the need to share the underlying data. Before we proceed with detailed summary about major findings and contributions of this study, it would be appropriate to recall our main research question that we have defined as follows:

Research Question:

1. Given a Virtual Ideal Functionality Framework (VIFF), by applying MPC techniques to particular genomic sequence comparison algorithm - BWT transform, we would like to develop secure distributed protocols which will help to achieve full privacy preserving sequence comparison within two construction:

   (a) Private query on private genomic data;
   (b) Private query on public genomic data.

2. Achieve verification and correctness proof of the proposed approach for all involved parties by zk-SNARK concept.

In order to explore first research question in more detail, we gave an efficient protocol for sequence comparison, in which neither entity reveals anything about underlying private data to other entity. We have discussed the applicability of MPC techniques in genomic data processing and indicated advantages that MPC may introduce from privacy-preserving perspective compared to the traditional sequence alignment methodologies. Our discussion of MPC research field has been enriched with explanation of working principle of framework (VIFF), in which search algorithms have been implemented. In order to demonstrate the applicability of our framework, dynamic programming approach has been applied. Despite the fact that, MPC does not scale well with Big Data needs and performs with decreasing efficiency in bigger data (i.e. longer genomic sequences), we still could show that it is possible to obtain oblivious genomic data processing in multi-party context for particular algorithms (e.g. $BWT[18]$, $IBWT[18]$).

One of the major contributions of this study was to highlight the importance of the protocol construction for performance analysis, in other words whether private query on private genomic data (case a) or private query on public genomic data (case b) has been processed. Hereby, two separate privacy-preserving frameworks have been implemented and compared from the functionality perspective. As might have been expected, in case a, sequence comparison appeared to be significantly slower compared to the case b (Section 3.4.2). The reason behind this conclusion is the branching in recursions of the BWT inexact search algorithm. As algorithm is not able to detect which parts of the private search query has already been matched with private reference data in case a, it simply checks for all possibilities to match. However, in case b, as reference data is public, algorithm therefore ignores cases which already have been excluded in earlier steps of search and reduces branching. Nevertheless, both implementations guarantee full privacy and enable joint data
analytics without the need of revealing data itself.

Prior to investigate second research problem accordingly, we have identified the concept of zero-knowledge proof as being crucial step in verification of the entire approach and ensuring correctness. In addition, the major contribution of this thesis is a proposed verification technique that facilitated the oblivious verification of the inexact string (i.e., genomic sequence) matches. Implemented technique consists of two major stages: proof function - log file extraction and correctness evaluation of the search algorithm. During the execution of inexact search, a so called log file is generated, which keeps record of all computations occurred in order to be used later in proof function. In other words, log file identifies exact number of the equations to be checked, namely insertions, deletions and substitutions required to convert one genomic sequence into the other. We continue with correctness evaluation stage of our oblivious verification, during which every step of the computation existing in log file is checked with help of zk-SNARK based on the cryptographic zero-knowledge proof. This procedure can be performed for every entity that wants to evaluate correctness of the each computation, resulting into verification of the entire algorithm.

To conclude, we have investigated and presented oblivious protocols for privacy-preserving sequence comparison technique (i.e., inexact search) obtained from particular algorithm (BWT transform). These protocols have been constructed separately for two different cases (case a and case b, explained above) and analyzed from both functionality and security perspective. Our approach constitutes a relatively new research domain and can be considered as a first step in privacy-preserving genomic computation research domain. We believe that developed protocols are reliable and secure, as their verification and correctness proof are also presented in this study.

Furthermore, MPC techniques that have been applied in this thesis, have applications beyond computational biology and in particular, DNA sequence alignment domain. These techniques can be used to obtain other efficient dynamic programming algorithms which preserve privacy of individual contributors over distributed datasets. As an application domain example, due to the MPC techniques any data mining algorithm can be performed on distributed data, while only revealing mining results to the other parties.

4.1 Limitations and Future Work

In this section, observed limitations of our implemented approach and possibilities for the future research within the oblivious sequence comparison study domain will be emphasized. Three major findings have been identified and presented below:

4.1.1 VIFF framework

First limitation of our oblivious sequence alignment method in multi-party context is related to the implemented framework that assumes the honest-majority model and passive adversary which make approach suitable for certain scenarios. Recall the fact from Section 2.4.1 that, subset of corrupted entities are fixed in static adversary model. In particular, VIFF framework have obligation that, if protocol run between n participants, less than n/2 of them should be corrupted.

A challenging task can be designing an efficient protocol in order to secure genomic data processing against malicious adversary also. It would be interesting to apply our approach in malicious adversary model in order to enable joint genomic data processing and evaluate applicability of the multi-party scheme in more realistic scenarios.

In addition, we have implemented several comparison techniques in VIFF framework in order to detect the fastest approach in BWT sequence alignment domain. These findings illustrated in Figure B.1 and played a crucial role for comparison method selection in our original implementation of the framework. After considering the analysis results within range of the various comparison methods,
we have decided to apply a particular method (i.e., interpolation) that increased performance significantly. Careful analysis of comparison methods for different algorithm phases and development of the new techniques can be an interesting research area to explore.

4.1.2 Non-realistic test data

Second important aspect that should be investigated is the fact that evaluation of our implementation presented in performance analysis (see Section 3.4.2) is not obtained from realistic genomic inputs (i.e., from real human genomes or DNA sequences). This limitation does not allow to track down a particular realistic evaluation of the algorithm in multi-party context. However, our experimental results discussed above, concluded that MPC techniques are tractable on genomic sequences of up to hundred characters in length.

In addition, another aspect of future work is related to the possibility of executing proposed approach in framework that involves more players rather than three. Although, performance of the inexact search algorithm in multi-party scheme has been evaluated for only three parties in this thesis, we believe that its possible to execute algorithm for more players. Further, future study can consider joint data analytics for larger set of entities which suits more to real-life scenarios, especially in health care sector where patient data needs to be distributed often among multiple entities (i.e., hospitals).

4.1.3 Privacy-Efficiency trade-off

Although, our approach ensures full privacy of DNA sequence alignment within BWT transform protocol and contributes several steps toward to efficiently handle heavy computational tasks, additional performance improvements are required as it’s still much slower than normal sequence alignment.

In fact, ensuring perfect cryptographic security costs a lot in computation and communication. Our approach generalizes an important step toward application of other dynamic programming algorithms both in health care and other sectors. The problem of oblivious sequence alignment within MPC context could be considered a relatively new research direction for secure computation discipline. Development of fast and secure techniques, in particular careful analysis of the other sequence alignment algorithms in order to succeed efficient, oblivious and collaborative genome processing is an important challenge for future researchers. We hope that our approach to facilitate MPC in practical application of genomic processing and corresponding performance improvements on that will enable many different privacy-preserving applications that use dynamic programming to be deployed.
Bibliography


Appendix A

VIFF: Secret Indexing

Secret indexing function operates on Shamir secret-shared values and produces unit vector representation of them. Comparison is succeeded by processing the bits of secret share \( x \) in reverse order. That is, in function \( si1 \) the bits of secret \( x \) are processed from most-significant to least-significant. This is possible by \( lsb() \) function, which defined earlier in Section 2.4.5.

Secret index function is constructed to handle the general case of a secret input \( x \). Precisely, if input \( x \) is a secret share, the function is executed using Shamir secret-shared values. The value of \( n \), namely length of the unit vector is always supposed to be publicly known by all parties. In general, function \( si1() \) performs an integer comparison (particularly, \( lsb \) function) and a scalar multiplication of a bit and a vector, apart from some simpler operations. A call to scalar multiplication with parameters \( (b, v) \) for a vector \( v \) of length, say, \( n \) results into \( n \) exchange of shares between the participants. Below defined implementation of explained version of the secret indexing (as \( \text{Secret Index 1} \)):

```python
def secret_index_new(x, n):
    # \( si1 \) returns all-0 vector of length \( n-1 \) (if \( x=0 \)) and \((x-1)\)-st unit vector of length \( n-1 \) (if \( 1<=x<n \))

def si1(x, n):
    if n==1:
        return []
    elif n==2:
        return [x]
    else:
        b = rt.lsb(x)
        v = si1((x-b)*\sim Zp(2), (n+1)/2)
        w = rt.scalar_mul(b, v)
        return [b-sum(w)] + [v[i/2]-w[i/2] if i%2==0 else w[i/2] for i in xrange(n-2)]
    v = si1(x, n)
    return [1-sum(v)]+v
```

Listing A.1: \( \text{Secret Index 1} \) function [24]
APPENDIX A. VIFF: SECRET INDEXING

There is yet another version of this function (Secret Indexing 2) which can be used with both small and large fields. However, Secret Indexing 2 is less efficient in large field compared to the function defined in A.1.

```python
def secret_index_ix(rt, Zp, ix): return Share(Zp, rt.w_powers[-(ix + 1)])

def secret_index_xi(rt, Zp, om):
    for ix in range(len(rt.w_powers)):
        if rt.w_powers[-(ix + 1)] == om:
            return ix

def pow_list(a, x, n):
    if n == 1:
        return [a]
    xs = []
    x2s = pow_list(a, x ** 2, n // 2)
    for x2 in x2s:
        xs.append(x2)
        xs.append(x2 * x)
    if n % 2 == 1:
        xs.append(xs[-1] * x)
    return xs

def secret_index(rt, Zp, om, n):
    ninv = ~Zp(len(rt.w_powers))
    coefs = pow_list(om, om, len(rt.w_powers))
    deltas = []
    for i in range(n):
        deltas.append(ninv * sum([coefs[j] * rt.w_powers[((i + 1) * (j + 1)) % len(rt.w_powers)] for j in range(len(rt.w_powers))])
    return deltas

def secret_index_get(rt, Zp, om, arr):
    deltas = secret_index(rt, Zp, om, len(arr))
    return rt.in_prod(deltas, arr)
```

Listing A.2: Secret Index 2 function

In order to compare two different secret indexing functions in detail, Table A.1 can be considered. As it can be clearly seen, Secret indexing 1 which only works with large fields due to the VIFF: lsb and comparison functions, is more efficient compared to the Secret indexing 2. As in secret indexing 1 function does not work with small primes, comparison for small fields cannot be held between two functions. Secret indexing 2 is more efficient in small primes rather than the large primes.

Table A.1: Time Efficiency of the different Secret Indexing techniques

<table>
<thead>
<tr>
<th>Secret index</th>
<th>(Large Prime-LP), length-10</th>
<th>LP-20</th>
<th>LP-30</th>
<th>(Small prime-SP), 10</th>
<th>SP-10</th>
<th>SP-30</th>
</tr>
</thead>
<tbody>
<tr>
<td>Secret Index 1</td>
<td>0.07</td>
<td>0.13</td>
<td>0.17</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Secret Index 2</td>
<td>0.88</td>
<td>2.21</td>
<td>3.67</td>
<td>0.43</td>
<td>1.65</td>
<td>3.40</td>
</tr>
</tbody>
</table>

1implemented by Dr. Meilof Veeningen during the study of this thesis.
Appendix B

VIFF: Comparisons on BWT

Figure B.1: Time efficiency of BWT Transform Algorithm with different comparison methods in multi-party scheme
Appendix C

Performance analysis

C.1 Private Inexact Search Algorithm

Figure C.1: Time efficiency of Private Inexact Search Algorithm given reference string (length of 70) in one-party scheme
## APPENDIX C. PERFORMANCE ANALYSIS

### Figure C.2: Time efficiency of Private Inexact Search Algorithm given reference string (length of 10) in one-party scheme

<table>
<thead>
<tr>
<th>Len(W)</th>
<th>Z=0</th>
<th>Z=1</th>
<th>Z=2</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0.51</td>
<td>1.95</td>
<td>7.42</td>
</tr>
<tr>
<td>5</td>
<td>0.93</td>
<td>2.86</td>
<td>12</td>
</tr>
<tr>
<td>7</td>
<td>1.25</td>
<td>3.73</td>
<td>16</td>
</tr>
</tbody>
</table>

### Figure C.3: Time efficiency of Private Inexact Search Algorithm given reference string (length of 20) in one-party scheme

<table>
<thead>
<tr>
<th>Len(W)</th>
<th>Z=0</th>
<th>Z=1</th>
<th>Z=2</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0.84</td>
<td>3.26</td>
<td>7.42</td>
</tr>
<tr>
<td>5</td>
<td>1.7</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>8</td>
<td>16</td>
</tr>
</tbody>
</table>
APPENDIX C. PERFORMANCE ANALYSIS

Figure C.4: Time efficiency of Private Inexact Search Algorithm given reference string (length of 20) with all A’s (character variance) in one-party scheme.

Figure C.5: Time efficiency of Private Inexact Search Algorithm given reference string (length of 20) in MPC scheme.
Relation between mismatch limit (z) & search query (W) length
MPC, len(X) = 10

Figure C.6: Time efficiency of Private Inexact Search Algorithm given reference string (length of 10) in MPC scheme
C.2 Full Private Inexact Search Algorithm

![Graph showing the relationship between mismatch limit (z) and search query (W) length](image)

Figure C.7: Time efficiency of Full Private Inexact Search Algorithm given reference string (length of 10) in one-party scheme

<table>
<thead>
<tr>
<th>Len(W)</th>
<th>Z=0</th>
<th>Z=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>9.73</td>
<td>153</td>
</tr>
<tr>
<td>5</td>
<td>115</td>
<td>3192</td>
</tr>
<tr>
<td>7</td>
<td>1432</td>
<td></td>
</tr>
</tbody>
</table>
Figure C.8: Time efficiency of Full Private Inexact Search Algorithm given reference string (length of 20) in one-party scheme