Ph.D. DISSERTATION

Finding Highly Similar Regions of Genomic Sequences through Homomorphic Encryption

동형암호를 이용한 유전체 서열에서의 매우 유사한 영역을 찾는 알고리즘

FEBRUARY 2023

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Magsarjav Bataa

Finding Highly Similar Regions of Genomic Sequences through Homomorphic Encryption

동형암호를 이용한 유전체 서열에서의 매우 유사한 영역을 찾는 알고리즘

지도교수 박 근 수

이 논문을 공학박사학위논문으로 제출함

2022 년 12 월

서울대학교 대학원

컴퓨터 공학부

Magsarjav Bataa

Magsarjav Bataa의 박사학위논문을 인준함

2022 년 12 월

위 원 장	김 선	(인)
부위원장	박 근 수	(인)
위 원	천 정 희	(인)
위 원	송 용 수	(인)
위 원	김 현 준	(인)

Abstract

Finding Highly Similar Regions of Genomic Sequences through Homomorphic Encryption

Magsarjav Bataa
Department of Computer Science
and Engineering
College of Engineering
The Graduate School
Seoul National University

Finding highly similar regions of genomic sequences is a basic computation of genomic analysis. Genomic analyses on a large amount of data are efficiently processed in cloud environments, but outsourcing them to a cloud raises concerns over the privacy and security issues. Homomorphic encryption is a powerful cryptographic primitive that preserves privacy of genomic data in various analyses processed in an untrusted cloud environment.

First, we present an efficient algorithm for finding highly similar regions of two homomorphically encrypted sequences, based on the Smith-Waterman recurrence. With the efficient location retrieval, parallel computations, and a proper HE scheme, it shows good performances in the experiment so as to be useful in practice.

Second, we also propose an efficient algorithm for finding highly similar regions of two sequences represented by homomorphically encrypted variants, and conduct extensive experiments and parameter sensitivity analysis on real and synthetic datasets to show its performance. In the experiment, it finds highly similar regions of the sequences in real datasets in a feasible time.

Keywords: sequence alignment; homomorphic encryption; highly similar region; local alignment; privacy-preserving computation

Student Number: 2018-38143

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Due to the advancements in genomic sequencing technology over the past decade, genomic data are widely used in many different areas such as healthcare, biomedical research, and forensics. Human genomic data can reveal sensitive information of individuals, and therefore, protecting its privacy is becoming important in the applications. The privacy threats, attacking schemes, and privacy-preserving methods for genomic data were studied in [4, 71, 55].

Genomic analyses on a large amount of data are efficiently processed in cloud computing environments, but outsourcing genomic data to a cloud raises concerns over the privacy and security issues. Homomorphic Encryption (HE) is considered as one of the promising solutions for secure outsourced computation over genomic data [50]. HE is a cryptographic primitive that allows computations to be performed on encrypted data without decryption. Since Fully Homomorphic Encryption (FHE) [38] with support for an unlimited number

of addition and multiplication operations was introduced in 2009, several FHE schemes have been proposed and used in various applications. According to the provided operations, they can be classified into two categories: bit-wise and word-wise [25]. Generally, additions and multiplications of the word-wise HE schemes such as BGV [14] and HEAAN [24] are much faster than non-polynomial operations such as min/max, comparison, and equality test since they are often implemented by many additions and multiplications that are built-in operations. On the other hand, built-in operations of the bit-wise HE schemes such as FHEW [35] and TFHE [27] are logic gates such as NAND and NOT, and thus the non-polynomial operations can be implemented by them simply, and run faster than additions and multiplications which often consist of a number of logic gates to be executed sequentially.

Extensive research has been done to preserve privacy of genomic data using HE in various analyses: Hamming distance [48], edit distance [26, 48], logistic regression [21, 45, 13, 18, 51], genome-wide association studies (GWAS) [50, 8, 64, 46, 9, 56], and others [59, 19, 49, 67]. In particular, Cheon et al. [26] described how to compute edit distance between two homomorphically encrypted sequences. Kim and Lauter [48] proposed methods to compute Hamming distance and approximate edit distance between two homomorphically encrypted sequences represented by Variant Call Format (VCF) files. Moreover, HE is used to preserve the privacy of user data in AI/ML services. Most studies consider neural network training/inference with HE [43, 20, 34, 11, 10, 17, 54] while some works focus on specific tasks such as recognizing human action [47] and detecting COVID-19 [70] using HE.

In addition to HE, secure multiparty computation (SMPC) approaches are used to protect the privacy of user data in genomic analyses [42, 30, 12, 33, 31]. Solutions based on SMPC often require high communication costs which limit

their applications. Unlike the SMPC, HE does not require any interaction during the computation. Therefore, HE is more suitable for cloud computing environments where genomic analyses are mostly performed. Moreover, most HE schemes are based on strong security assumptions that are considered resistant to attack by even quantum computers. Even though HE is generally considered less efficient than SMPC, it has become more practical due to recent improvements.

Similarity between two genomic sequences may indicate functional, structural and/or evolutionary relationships between them, and it is mostly computed by hamming distance, edit distance, and sequence alignment algorithms. Finding highly similar regions (HSR) of two genomic sequences is a basic step of many applications such as homology (common ancestry) detection [60, 68, 22], database search [16, 37, 53], read alignment [3, 52, 41], and ancient DNA restoration [63]. For instance, in the context of whole human genome resequencing, a massive number of short reads generated by next-generation sequencing technologies are required to accurately resequence an individual genome, and it creates a problem of aligning those reads to a reference genome [41]. For the given read and the reference genome, HSR between them indicate the correct location of the read on the reference genome, and they are commonly computed by genome alignment tools employing the Smith-Waterman algorithm [66, 39].

1.2 Contribution

In this thesis we present an efficient algorithm [5] that finds HSR of two homomorphically encrypted genomic sequences, based on the Smith-Waterman recurrence. It includes an efficient method of finding the location of HSR by storing the location of the sources where each score comes from, in order to

avoid costly conditional branching in the backtracking. To reduce computation time, we use two level parallel computations: one for filling the entries of the dynamic programming recurrence, and the other for implementing the circuits. With the efficient location retrieval, parallel computations, and a proper HE scheme, our implementation shows good performances in the experiment so as to be useful in practice.

We also propose an efficient algorithm [6] that finds HSR of two genomic sequences represented by homomorphically encrypted variants. First, we align regions of the sequences and compute their scores efficiently. Second, we find HSR from the aligned regions utilizing classic algorithms for the maximum subarray problem. Furthermore, we describe how to implement our algorithm using the bit-wise and word-wise HE schemes. Finally we conduct extensive experiments and parameter sensitivity analysis on real and synthetic datasets to show the performance of our algorithm. The experiments show that it outperforms the previous algorithm by up to 2 orders of magnitude in terms of elapsed time, while the score accuracy of HSR is over 91% compared with the Smith-Waterman algorithm. In particular, it takes 2 minutes to find the score and location of HSR of two homomorphically encrypted sequences with length 30,000 bps each in a real dataset. Overall, it obtains HSR of the sequences in a feasible time. In addition, the experiments show that the bit-wise implementation using the TFHE scheme is faster than the word-wise implementation using the HEAAN scheme for the short sequences. However, the word-wise implementation outperforms the bit-wise implementation as the sequences get longer.

1.3 Organization

The thesis is organized as follows. Chapter 2 provides problem statements and two types of HE schemes. Chapter 3 describes homomorphic circuits used in the algorithms. Chapters 4 and 5 present algorithms for finding optimal and approximate HSR, respectively. Chapter 6 concludes the thesis.

Chapter 2

Preliminaries

2.1 Highly Similar Regions

X	A	T	A	A	T	A	A	T	A	C	T	G	A	G	A	A	–	–	G	C	A	A	A	G	T	T
Y			A	T	C	C	A	C	T	A	–	T	G	A	G	A	C	C	C	G	C	A	A	T	A	C	C	T
align							M	S	M	M	–	M	M	M	M	M	S	——		M	M	M	M				
score							5	-3	5	5	-9	5	5	5	5	5	-3	-9	-1	5	5	5	5				

Figure 2.1: Highly similar regions of two DNA sequences

For a pair of genomic sequences, the problem of finding a *local alignment* is to determine and align *highly similar regions* of them such that the alignment score is the maximum with respect to a scoring scheme. Highly similar regions (HSR) of two sequences with affine gap penalties are commonly computed by the Smith-Waterman algorithm (SW) [66, 39]. For example, the shadowed bases in Figure 2.1 indicate the HSR of DNA sequences X and Y of lengths 24 and 25, respectively. In the figure, row align shows the alignment between the HSR. In

the alignment, M denotes a match, S denotes a mismatch, and − (dash) denotes a gap. Row score shows the corresponding scores for the matches, mismatches, and gaps in the alignment. Here, we use a scoring scheme $\{s_m/s_s, g_o, g_e\}$ where the match score $s_m = 5$, mismatch score $s_s = -3$, gap opening penalty $g_o = -9$, and gap extending penalty $g_e = -1$, as in [44]. In this example, the alignment score is 35 (i.e., sum of the scores), and the regions start from the sixth base of X and fifth base of Y, and end at the twentieth bases of X and Y.

2.2 Smith-Waterman Algorithm

Given a pair of genomic sequences and a scoring scheme, the SW algorithm finds an optimal local alignment between the two sequences, allowing gaps. An arbitrary gap penalty was used in the original SW algorithm [66]. Later, Gotoh [39] and Altschul [2] improved the algorithm for the affine gap penalty.

Let g_e be the (negative) gap extension penalty, g_o be the (negative) gap opening penalty, $\mathsf{X} = \mathsf{X}_1 \dots X_n$ and $\mathsf{Y} = \mathsf{Y}_1 \dots \mathsf{Y}_m$ be the sequences to be aligned. Furthermore, let $sim(\mathsf{X}_i, \mathsf{Y}_j)$ be the similarity score defined as

$$sim(\mathsf{X}_i, \mathsf{Y}_j) = \begin{cases} s_m & \text{if } \mathsf{X}_i = \mathsf{Y}_j \\ s_s & \text{if } \mathsf{X}_i \neq \mathsf{Y}_j \end{cases}$$

where s_m is the (positive) score for a match and s_s is the (negative) score for a mismatch.

In the SW algorithm, the trace-back matrices H, P, and Q are initialized first. The size of each matrix is $(n+1) \times (m+1)$ with 0-based indexing. The first rows and columns are filled as

$$\begin{aligned} &H_{i,0} = 0, Q_{i,0} = 0 \text{ for } i = 0, \dots, n \\ &H_{0,j} = 0, P_{0,j} = 0 \text{ for } j = 0, \dots, m. \end{aligned} \tag{2.1}$$

Next, the matrices are filled by the recursions as

$$P_{i,j} = \max \begin{cases} P_{i-1,j} + g_e \\ H_{i-1,j} + g_o \end{cases} \tag{2.2}$$

$$Q_{i,j} = \max \begin{cases} Q_{i,j-1} + g_e \\ H_{i,j-1} + g_o \end{cases} \tag{2.3}$$

$$H_{i,j} = \max \begin{cases} P_{i,j} \\ Q_{i,j} \\ H_{i-1,j-1} + sim(\mathsf{X}_i, \mathsf{Y}_j) \\ 0 \end{cases} \tag{2.4}$$

for $i = 1, \ldots, n$ and $j = 1, \ldots, m$.

Finally, an optimal local alignment can be retrieved by the backtracking using the trace-back matrices. The backtracking starts from the highest score in H, traverses through the source of the current score in H, P, or Q recursively according to the trace-back matrices until 0 encounters in H.

2.3 Representation of Pair of Sequences

Suppose we have two genomic sequences X and Y stored in Variant Call Format (VCF) [32] files along with a same reference genome R. Then they can be represented as a pair of lists $\mathbb{X}$ and $\mathbb{Y}$ which summarize their variants compared with R, as shown in Figure 2.2. More specifically, $\mathbb{X}$ and $\mathbb{Y}$ are lists of variants, and each variant in the lists represents a variable-length substitution (e.g., insertion, deletion, or substitution) at given position of R. In the lists, column `pos` stores the positions of R in an increasing order, column `reflen` stores

$\mathbb{X}$

i	pos	reflen	altlen	alt
1	3	1	3	AAT
2	10	2	0	-
3	13	3	2	AG
4	18	0	0	-
5	22	4	2	AG

$\mathbb{Y}$

i	pos	reflen	altlen	alt
1	3	4	2	CC
2	10	1	1	A
3	13	4	2	AG
4	18	0	3	CCC
5	22	5	4	TACC

Figure 2.2: Lists of variants

the lengths of reference bases, column `altlen` stores the lengths of alternate bases, and column `alt` stores alternate bases. For example, in Figure 2.2, the second variant in $\mathbb{X}$, the second and fourth variants in $\mathbb{Y}$ (i.e., $\mathbb{X}_2$, $\mathbb{Y}_2$, and $\mathbb{Y}_4$) represent a deletion of two bases, a single nucleotide polymorphism (SNP), and an insertion of three bases (CCC) on the reference genome, respectively. For a variant x in $\mathbb{X}$ (i.e., $x = \mathbb{X}_i$), $x.\texttt{pos}$, $x.\texttt{reflen}$, $x.\texttt{altlen}$, and $x.\texttt{alt}$ denote the position of R, the length of reference bases, the length of alternate bases, and alternate bases, respectively (same for a variant y in $\mathbb{Y}$). For example, when $x = \mathbb{X}_1$ in Figure 2.2, $x.\texttt{pos} = 3$, $x.\texttt{reflen} = 1$, $x.\texttt{altlen} = 3$, and $x.\texttt{alt} = \text{AAT}$. For simplicity, we can make both columns of `pos` in $\mathbb{X}$ and $\mathbb{Y}$ the same by arranging the variants according to their positions and inserting empty variants ($\mathbb{X}_4$ in Figure 2.2). We assume that there is no variant whose length of the reference bases overlaps with its next position. If exists, it can be divided into two or more variants so that each of them satisfies the assumption. This representation can be obtained from the VCF files by simple preprocessing.

2.4 Bit-wise HE scheme

The Torus Fully Homomorphic Encryption (TFHE) scheme was introduced in [27], and the TFHE library [28] was released to public in 2016. The TFHE scheme supports homomorphic evaluations of binary gates, negation, and multiplexer. It automatically performs bootstrapping after each evaluation to keep the ciphertext noise small. Therefore, we can ignore the depth, which is the main obstacle for performing computation using most HE schemes. The TFHE scheme consists of the following homomorphic gates. Note that we use the symmetric version of THFE.

- $sk, ck \leftarrow \mathsf{KeyGen}(\lambda)$: For a security parameter λ, it outputs a secret key sk and a cloud key ck.
- $\overline{m} \leftarrow \mathsf{Enc}(m, sk)$: For a plain bit m and a secret key sk, it outputs a ciphertext $\overline{m}$ which is an encrypted m.
- $m \leftarrow \mathsf{Dec}(\overline{m}, sk)$: For a ciphertext $\overline{m}$ and a secret key sk, it outputs a plain bit m which is a decrypted $\overline{m}$.
- $\overline{m_1 \oplus m_2} \leftarrow \mathsf{Xor}(\overline{m}_1, \overline{m}_2, ck)$: For ciphertexts $\overline{m}_1, \overline{m}_2$, and a cloud key ck, it outputs a ciphertext which is a homomorphic evaluation of the logical XOR gate. Similarly, the input and output of other binary gates such as And, Or, and Xnor are defined in the same way as Xor.
- $\overline{\neg m} \leftarrow \mathsf{Not}(\overline{m}, ck)$: For a ciphertext $\overline{m}$, and a cloud key ck, it outputs a ciphertext which is a homomorphic evaluation of the negation.
- $\overline{m}_3 \leftarrow \mathsf{Mux}(\overline{s}, \overline{m}_1, \overline{m}_2, ck)$: For ciphertexts $\overline{s}, \overline{m}_1, \overline{m}_2$, and a cloud key ck, it outputs a ciphertext $\overline{m}_3$ which is a homomorphic evaluation of the multiplexer, i.e., $m_3 = m_1$ if $s = 1$, and $m_3 = m_2$ if $s = 0$.

- $\overline{m} \leftarrow \mathsf{Const}(m, ck)$: For a plain bit m and a cloud key ck, it outputs a ciphertext $\overline{m}$. This gate converts a bit to a ciphertext, whose plaintext m is known to the public.

2.5 Word-wise HE scheme

The HEAAN scheme was introduced in [24]. The scheme supports an approximate computation of real numbers in an encrypted state. Even though it is a leveled HE scheme, an efficient bootstrapping method was introduced in [23]. In HEAAN, a plaintext m is a real vector of size n, i.e., $m = (m_1, m_2, \ldots, m_n)$ ($m_i \in \mathbb{R}$), and we denote its ciphertext vector by $\overline{m}$ where $\overline{m}_i$ is an encrypted m_i for $1 \leq i \leq n$. The scheme consists of the following operations.

- $sk, pk, ek \leftarrow \mathsf{KeyGen}(N, \lambda, l, q)$: For a ring dimension N, a security parameter λ, a level parameter l, and a number of quantization bits q, it outputs a secret key sk, a public key pk, and an evaluation key ek.

- $\overline{m} \leftarrow \mathsf{Enc}(m, pk)$: For a plaintext vector m and a public key pk, it outputs a ciphertext vector $\overline{m}$.

- $m \leftarrow \mathsf{Dec}(\overline{m}, sk)$: For a ciphertext vector $\overline{m}$ and a secret key sk, it outputs a plaintext vector m which is a decrypted $\overline{m}$.

- $\overline{c} \leftarrow \mathsf{Add}(\overline{a}, \overline{b})$: For ciphertext vectors $\overline{a}$ and $\overline{b}$, it outputs a ciphertext vector $\overline{c}$ where $c_i = a_i + b_i$.

- $\overline{c} \leftarrow \mathsf{Subtract}(\overline{a}, \overline{b})$: For ciphertext vectors $\overline{a}$ and $\overline{b}$, it outputs a ciphertext vector $\overline{c}$ where $c_i = a_i - b_i$.

- $\overline{c} \leftarrow \mathsf{Multiply}(\overline{a}, \overline{b}, ek)$: For ciphertext vectors $\overline{a}$, $\overline{b}$, and an evaluation key ek, it outputs a ciphertext vector $\overline{c}$ where $c_i = a_i \cdot b_i$.

- $\overline{c} \leftarrow \mathsf{MultiplyConst}(\overline{a}, t)$: For a ciphertext vector $\overline{a}$ and a real vector t, it outputs a ciphertext vector $\overline{c}$ where $c_i = a_i \cdot t_i$. The second argument t can be a real number, not a vector. In such a case, $c_i = a_i \cdot t$.

- $\overline{c} \leftarrow \mathsf{LeftRotate}(\overline{a}, t, ek)$: For a ciphertext vector $\overline{a}$, an integer t, and an evaluation key ek, it outputs a ciphertext vector $\overline{c}$ where c is equal to vector a rotated left by t slots, i.e., $c = (a_{t+1}, \ldots, a_n, a_1, a_2, \ldots, a_t)$.

- $\overline{c} \leftarrow \mathsf{RightRotate}(\overline{a}, t, ek)$: For a ciphertext vector $\overline{a}$, an integer t, and an evaluation key ek, it outputs a ciphertext vector $\overline{c}$ where c is equal to vector a rotated right by t slots, i.e., $c = (a_{n-t+1}, \ldots, a_n, a_1, a_2, \ldots, a_{n-t})$.

In addition to the above built-in operations, we privately communicated with the authors of HEAAN and obtained the implementations of the following operations.

- $\overline{c}, \overline{g} \leftarrow \mathsf{MinMax}(\overline{a}, \overline{b}, u, ek)$: For ciphertext vectors $\overline{a}$, $\overline{b}$, an iteration number u, and an evaluation key ek, it outputs ciphertext vectors $\overline{c}$ and $\overline{g}$ where $c_i = \max(a_i, b_i)$ and $g_i = \min(a_i, b_i)$. Note that a_i and b_i must be real numbers between -1 and 1, inclusive [25].

- $\overline{c} \leftarrow \mathsf{Equals}(\overline{a}, \overline{b}, ek)$: For ciphertext vectors $\overline{a}$, $\overline{b}$, and an evaluation key ek, it outputs a ciphertext vector $\overline{c}$ where $c_i = 1$ if $a_i = b_i$, and $c_i = 0$ otherwise. Note that a_i and b_i must be integers between 0 to 70.

- $\overline{c} \leftarrow \mathsf{EqualsZero}(\overline{a}, ek)$: For a ciphertext vector $\overline{a}$, and an evaluation key ek, it outputs a ciphertext vector $\overline{c}$ where $c_i = 1$ if $a_i = 0$, and $c_i = 0$ otherwise. Note that a_i must be an integer between 0 to 70.

The above operations are performed on each slot of the input vector(s) in parallel (i.e., in a SIMD manner) [24]. Note that the ring dimension N and

size n of a plaintext vector are required to be powers of two and $n \leq N/2$ in HEAAN.

2.6 Problem Statement

Optimal HSR. Given two homomorphically encrypted genomic sequences, find HSR of them (the SW solution) efficiently without revealing any information of the sequences (Chapter 4).

Approximate HSR. Given two genomic sequences represented by a pair of lists of homomorphically encrypted variants (as described in Section 2.3), find HSR of them (which approximate the SW solution) efficiently without revealing any information of the sequences (Chapter 5).

2.7 Related Work

Edit Distance. Cheon et al. [26] described a way to compute (optimal) edit distance between two homomorphically encrypted sequences based on the Wagner-Fisher algorithm [69]. It is similar to the first problem (Optimal HSR) with respect to the computation method because both problems are solved by DP recurrences. Since computing the edit distance of only two sequences is inefficient in [26], the experiment was done for multiple pairs of sequences in order to reduce the amortized time, which is equal to the total running time divided by the number of pairs. In particular, it takes around 5 hours for 682 pairs of DNA sequences with lengths 8.

Hamming Distance and Approximate Edit Distance. Kim and Lauter [48] proposed methods to compute Hamming distance and approximate edit distance between two homomorphically encrypted sequences represented by VCF files. Both problems are similar to the second problem (Approximate HSR) in

terms of the input representations and computation approaches. They compute the distance between each pair of variants at the same position based on the set difference metric in encrypted states and then compute the total distance by accumulating them in plain states (non-encrypted) to achieve better performance. It takes around two minutes to compute the Hamming distance and approximate edit distance of the two sequences represented by 5,000 encrypted variants.

Chapter 3

Homomorphic Circuits

In this chapter, we discuss homomorphic circuits which perform polynomial and non-polynomial operations over encrypted bits, implemented by the built-in gates in TFHE. We consider both sequential and parallel implementations of the circuits.

3.1 Implementation

We assume that input and output of the circuits are a ciphertext or ciphertext array of length w which is an encrypted bit or integer represented by two's complement, respectively. For instance, if $\overline{a}$ is a ciphertext array of length w, $\overline{a}_i$ is an encrypted i-th bit in the two's complement representation of an integer a. The bit length w is not required to be a power of two in the circuits. Note that the cloud key ck is omitted from the input of each circuit and homomorphic gate for simplicity.

- $\overline{c} \leftarrow \mathsf{Add}(\overline{a}, \overline{b})$: For ciphertext arrays $\overline{a}$ and $\overline{b}$ of length w, it outputs a

Algorithm 1 $\mathsf{Compare}(a, b)$

1: $a_w \leftarrow \mathsf{Not}(a_w)$
2: $b_w \leftarrow \mathsf{Not}(b_w)$ ▷ Flipping sign bits
3: **for** $i \leftarrow 1$ **to** w **parallel do**
4: $\quad s_i \leftarrow \mathsf{Xnor}(a_i, b_i)$
5: **for** $i \leftarrow 1$ **to** $\lceil \log w \rceil$ **do**
6: $\quad$ **for** $j \leftarrow 1$ **to** w **by** 2^i **parallel do**
7: $\quad\quad$ **if** $j + 2^{i-1} \leq w$ **then**
8: $\quad\quad\quad s_j \leftarrow \mathsf{And}(s_j, s_{j+2^{i-1}})$
9: $\quad\quad\quad a_j \leftarrow \mathsf{Mux}(s_{j+2^{i-1}}, a_j, a_{j+2^{i-1}})$
10: $c \leftarrow \mathsf{Or}(s_1, a_1)$
11: **return** c

ciphertext array $\overline{c}$ of size w where $c = a + b$. Its sequential computation can be implemented recursively as $\overline{c}_i = \mathsf{Xor}(\mathsf{Xor}(\overline{a}_i, \overline{b}_i), r_{i-1})$ where $r_i = \mathsf{Xor}(\mathsf{And}(\overline{a}_i, \overline{b}_i), \mathsf{And}(\mathsf{Xor}(\overline{a}_i, \overline{b}_i), r_{i-1}))$ and $r_0 = \overline{0}$ for $i = 1, \ldots, w$, as a Ripple-carry adder (RCA). To implement the parallel one, we use Carry-lookahead adder (CLA)[15, 40]. In the first step, CLA computes generate bits $g_i = \mathsf{And}(\overline{a}_i, \overline{b}_i)$ and propagate bits $p_i = \mathsf{Xor}(\overline{a}_i, \overline{b}_i)$ in parallel. The sum $\overline{c}$ can be expressed by the bits as $\overline{c}_i = \mathsf{Xor}(p_i, r_{i-1})$ where $r_i = \mathsf{Or}(g_i, \mathsf{And}(p_i, r_{i-1}))$ and $r_0 = \overline{0}$ for $i = 1, \ldots, w$. In order to calculate them in parallel, we need the group generate bits G_i and the group propagate bits P_i defined by the recursion

$$(G_i, P_i) = \begin{cases} (g_1, p_1) & \text{if } i = 1 \\ (g_i, p_i) \circ (G_{i-1}, P_{i-1}) & \text{if } i = 2, \ldots, w \end{cases}$$

Algorithm 2 $\mathsf{MultiplyConst}(a, t)$

$c \leftarrow \mathsf{EncConst}(0)$
for $i \leftarrow 1$ **to** $\lceil \log(|t| + 1) \rceil$ **do** ▷ $|t|$ denotes absolute value of t
  if $|t|_i = 1$ **then** ▷ $|t|_i$ denotes i-th bit of $|t|$
    $c \leftarrow \mathsf{Add}(c, a)$
  for $j \leftarrow w$ **to** 2 **do**
    $a_j \leftarrow a_{j-1}$
  $a_1 \leftarrow \mathsf{Const}(0)$
if $t < 0$ **then**
  for $i \leftarrow 1$ **to** w **do**
    $c_i \leftarrow \mathsf{Not}(c_i)$
  $c \leftarrow \mathsf{Add}(c, \mathsf{EncConst}(1))$
return c

where operator $\circ$ is defined as

$$(g, p) \circ (\hat{g}, \hat{p}) = (\mathsf{Or}(g, \mathsf{And}(p, \hat{g})), \mathsf{And}(p, \hat{p}))$$

for any bits $g, p, \hat{g}$ and $\hat{p}$. In the second step, CLA computes (G_i, P_i) for $i = 1, \ldots, w-1$ in parallel. This type of calculation is called prefix sum or scan. Among several ways to calculate the prefix sum in parallel, we use Sklansky's implementation [65]. In the final step, the sum is computed by $\overline{c}_1 = p_1$ and $\overline{c}_i = \mathsf{Xor}(p_i, G_{i-1})$ in parallel for $i = 2, \ldots, w$ since $r_i = G_i$.

- $\overline{c} \leftarrow \mathsf{Compare}(\overline{a}, \overline{b})$: For ciphertext arrays $\overline{a}$ and $\overline{b}$ of length w, it outputs a ciphertext $\overline{c}$ where $c = 1$ if $a > b$, and $c = 0$ otherwise. If we flip both the sign bits $\overline{a}_w$ and $\overline{b}_w$, we can compare them as unsigned integers. Its sequential computation can be implemented recursively as $\overline{c} = r_w$ where $r_i = \mathsf{Mux}(s_i, r_{i-1}, \overline{a}_i), s_i = \mathsf{Xnor}(\overline{a}_i, \overline{b}_i)$, and $r_0 = \overline{1}$ for $i = 1, \ldots, w$.

Moreover, r_w can be computed in parallel using a binary tree order as shown in Algorithm 1. In the algorithm, And and Mux at lines 8 and 9 can also be computed in parallel since there is no data dependency.

- $\overline{c} \leftarrow \mathsf{EncConst}(m)$: For an integer m, it outputs a ciphertext array $\overline{c}$ of length w where $c = m$ (represented by two's complement). It can be implemented using Const simply.

- $\overline{c} \leftarrow \mathsf{Equals}(\overline{a}, \overline{b})$: For ciphertext arrays $\overline{a}$ and $\overline{b}$ of length w, it outputs a ciphertext $\overline{c}$ where $c = 1$ if $a = b$, and $c = 0$ otherwise. Its sequential implementation can be computed recursively as $\overline{c} = e_w$ where $e_i = \mathsf{And}(e_{i-1}, \mathsf{Xnor}(\overline{a}_i, \overline{b}_i))$ and $e_0 = \overline{1}$ for $i = 1, \ldots, w$. Moreover, we can compute e_w in parallel using a binary tree order.

- $\overline{c} \leftarrow \mathsf{EqualsZero}(\overline{a})$: For a ciphertext array $\overline{a}$ of length w, it outputs a ciphertext $\overline{c}$ where $c = 1$ if $a = 0$, and $c = 0$ otherwise. It can be implemented using a smaller number of gates compared to Equals.

- $\overline{c} \leftarrow \mathsf{Subtract}(\overline{a}, \overline{b})$: For ciphertext arrays $\overline{a}$ and $\overline{b}$ of length w, it outputs a ciphertext array $\overline{c}$ of length w where $c = a - b$. Its sequential and parallel computations can be implemented similarly to Add since we use two's complement for representing the integers. In the first step of parallel implementation, the group generate and propagate bits in Add should be changed to $g_i = \mathsf{And}(\overline{a}_i, \mathsf{Not}(\overline{b}_i))$, and $p_i = \mathsf{Xnor}(\overline{a}_i, \overline{b}_i)$ for $i = 1, \ldots, w$. In the second step, calculating the prefix sum is exactly same as the addition. In the final step, the result is computed as $\overline{c}_1 = p_1$ and $\overline{c}_i = \mathsf{Xor}(p_i, \mathsf{Xor}(G_{i-1}, P_{i-1}))$ in parallel for $i = 2, \ldots, w$.

- $\overline{c} \leftarrow \mathsf{SubtractOne}(\overline{a})$: For a ciphertext array $\overline{a}$ of length w, it outputs a ciphertext array $\overline{c}$ of length w where $c = a - 1$. It can be implemented

using fewer gates compared to Subtract.

- $\overline{c} \leftarrow \mathsf{MultiplyConst}(\overline{a}, t)$: For a ciphertext array $\overline{a}$ of length w and an integer t, it outputs a ciphertext array $\overline{c}$ of length w where $c = a \cdot t$. It can be implemented as a simple binary multiplier, as described in Algorithm 2, where Add in line 11 can be implemented similarly to SubtractOne, using fewer gates compared to Add.

- $\overline{c} \leftarrow \mathsf{Select}(\overline{s}, \overline{a}, \overline{b})$: For a ciphertext $\overline{s}$, ciphertext arrays $\overline{a}$ and $\overline{b}$ of length w, it outputs a ciphertext array $\overline{c}$ of length w where $c = a$ if $s = 1$, and $c = b$ if $s = 0$. It can be implemented using Mux simply.

- $\overline{c} \leftarrow \mathsf{SelectZero}(\overline{s}, \overline{a})$: For a ciphertext $\overline{s}$ and a ciphertext array $\overline{a}$ of length w, it outputs a ciphertext array $\overline{c}$ of length w where $c = a$ if $s = 1$, and $c = 0$ if $s = 0$. It can be implemented using fewer gates compared to Select.

3.2 Analysis

Based on the running times of the built-in gates in TFHE, we estimate the span and work of each circuit. We use the CPU time of each binary gate as the unit cost of our estimation. Mux is regarded as 2 binary gates while Const and Not can be regarded as 0 because they are executed in less than a millisecond.

Table 3.1 shows the spans and works of sequential and parallel implementations of the circuits. For the parallel implementation of a circuit, the span is the number of the longest series of gates that have to be performed sequentially. Moreover, the work is the total number of gates used in the circuit. For the sequential implementation of a circuit, its span and work are the same. For example, in Algorithm 1, the span of the flipping bits in lines 1-2 is 0, the

Table 3.1: Spans and works of sequential and parallel circuits

Circuit	Sequential	Parallel	
		Span	Work
Add	$5w$	$2\lceil\log(w-1)\rceil+2$	$3\lceil\frac{w-1}{2}\rceil\lceil\log(w-1)\rceil+3w-1$
Compare	$3w$	$2\lceil\log w\rceil+2$	$4w-2$
Equals	$2w-1$	$\lceil\log w\rceil+1$	$2w-1$
EqualsZero	$w-1$	-	-
MultiplyConst	$5\lceil\log(\lvert t\rvert+1)\rceil w+2w$	-	-
Select	$2w$	2	$2w$
SelectZero	w	1	w
Subtract	$5w$	$2\lceil\log(w-1)\rceil+3$	$3\lceil\frac{w-1}{2}\rceil\lceil\log(w-1)\rceil+4w-2$
SubtractOne	$2w$	-	-

span of the loop in lines 3-5 is 1, the span of the double loop in lines 5-9 is $2\times\lceil\log w\rceil$ (the maximum cost of Mux and And is 2, and they are performed in parallel), and the span of Or at line 10 is 1. Thus their sum, $2\lceil\log w\rceil+2$, will be the span of parallel implementation of Compare. Furthermore, the work of the flipping bits in lines 1-2 is 0, the work of the loop in lines 3-5 is w, the work of the double loop in lines 5-9 is $3\times(w-1)$, and the work of Or at line 10 is 1. Thus their sum, $4w-2$, will be its work. The sequential computation of Compare can be implemented using a smaller number of gates, $3w$, compared to its parallel implementation. Note that the parallel implementations of EqualsZero, MultiplyConst, and SubtractOne are not used in the algorithms.

Chapter 4

Finding Optimal HSR

Given two homomorphically encrypted sequences, we find the score and location (starting and ending positions) of HSR between them in encrypted forms using the SW algorithm. We assume that the score and location of HSR are the same as those of an optimal local alignment obtained by the SW algorithm (see Figure 4.1).

Finding Starting Positions. In SW, an optimal local alignment of two sequences is obtained by backtracking using the scores in the trace-back matrices. In our case, the homomorphic computation of backtracking is extremely heavy due to a lot of conditional branching, and it makes finding the starting positions of HSR expensive. Therefore, we propose a lighter method that retrieves the two starting positions of HSR efficiently based on an idea derived from the backtracking of SW. In the backtracking, the trace-back matrices (Section 2.2) store the direction information as to where each score comes from. In contrast, we store the location of the source where each score starts from. In other words, we keep the origin of each score in additional matrices. It allows us to find the

starting positions of HSR directly without backtracking. Unlike SW, the actual local alignment of the two sequences can not be retrieved by our algorithm since it does not employ backtracking.

Finding Score and Ending Positions. Since an optimal local alignment ends at the highest score stored in the trace-back matrix of SW, we can obtain the score and two ending positions of HSR by finding the maximum score of the matrix while keeping its location.

Organization. The rest of the chapter is organized as follows. Section 4.1 gives an overview of our approach. Section 4.2 describes how we encode the data for homomorphic computation. Section 4.3 introduces the homomorphic computation of our algorithm. Section 4.4 presents the complexity analysis of our algorithm. Section 4.5 shows the experimental results of our algorithm.

4.1 Algorithm Overview

We store the location of the source where each score starts from in extra matrices. According to (2.2)-(2.4), the source of a score in H can be in H, P, or Q. Therefore, we use extra matrices H^x, H^y, P^x, P^y, Q^x, and Q^y to store the source since its location is expressed by two positions, one in X and the other in Y. Our algorithm consists of the following two steps.

(1) Filling Matrices. First we initialize H, P, and Q as in (2.1) and the extra matrices as in the following.

$$\begin{aligned} H^x_{i,0} = i, H^y_{i,0} = 0, Q^x_{i,0} = i, \text{ and } Q^y_{i,0} = 0 \text{ for } i = 0, \ldots, n \\ H^x_{0,j} = 0, H^y_{0,j} = j, P^x_{0,j} = 0, \text{ and } P^y_{0,j} = j \text{ for } j = 0, \ldots, m \end{aligned} \tag{4.1}$$

Next we fill H, P, and Q as in (2.2)-(2.4) and the extra matrices by the recur-

sions

$$P^x_{i,j} = \begin{cases} P^x_{i-1,j} & \text{if } P_{i,j} = P_{i-1,j} + g_e \\ H^x_{i-1,j} & \text{if } P_{i,j} = H_{i-1,j} + g_o \end{cases} \tag{4.2}$$

$$P^y_{i,j} = \begin{cases} P^y_{i-1,j} & \text{if } P_{i,j} = P_{i-1,j} + g_e \\ H^y_{i-1,j} & \text{if } P_{i,j} = H_{i-1,j} + g_o \end{cases} \tag{4.3}$$

$$Q^x_{i,j} = \begin{cases} Q^x_{i,j-1} & \text{if } Q_{i,j} = Q_{i,j-1} + g_e \\ H^x_{i,j-1} & \text{if } Q_{i,j} = H_{i,j-1} + g_o \end{cases} \tag{4.4}$$

$$Q^y_{i,j} = \begin{cases} Q^y_{i,j-1} & \text{if } Q_{i,j} = Q_{i,j-1} + g_e \\ H^y_{i,j-1} & \text{if } Q_{i,j} = H_{i,j-1} + g_o \end{cases} \tag{4.5}$$

$$H^x_{i,j} = \begin{cases} H^x_{i-1,j-1} & \text{if } H_{i,j} = H_{i-1,j-1} + sim(\mathsf{X}_i, \mathsf{Y}_j) \\ P^x_{i,j} & \text{if } H_{i,j} = P_{i,j} \\ Q^x_{i,j} & \text{if } H_{i,j} = Q_{i,j} \\ i & \text{if } H_{i,j} = 0 \end{cases} \tag{4.6}$$

$$H^y_{i,j} = \begin{cases} H^y_{i-1,j-1} & \text{if } H_{i,j} = H_{i-1,j-1} + sim(\mathsf{X}_i, \mathsf{Y}_j) \\ P^y_{i,j} & \text{if } H_{i,j} = P_{i,j} \\ Q^y_{i,j} & \text{if } H_{i,j} = Q_{i,j} \\ j & \text{if } H_{i,j} = 0 \end{cases} \tag{4.7}$$

for $i = 1, \ldots, n$ and $j = 1, \ldots, m$.

(2) Finding HSR. To report the score and location of HSR, we find the maximum score of H since the optimal alignment ends at it. Assume that the maximum score is stored at indices i^* and j^* in H, i.e., H_{i^*,j^*} is the maximum

score. Then i^* and j^* will be ending positions of the HSR, in X and Y, respectively. Moreover, $H^x_{i^*,j^*} + 1$ and $H^y_{i^*,j^*} + 1$ will be the starting positions of the HSR since the source location (i.e., the starting positions) of score at H_{i^*,j^*} is stored at $H^x_{i^*,j^*}$ and $H^y_{i^*,j^*}$.

H

		0	1	2	3	4	5	6	7	8	9
		-	A	G	A	T	G	C	A	G	G
0	-	0	0	0	0	0	0	0	0	0	0
1	T	0	0	0	0	5	0	0	0	0	0
2	A	0	5	0	5	0	2	0	5	0	0
3	T	0	0	2	0	10	1	0	0	2	0
4	C	0	0	0	0	1	7	6	0	0	0
5	A	0	5	0	5	0	0	4	11	2	1
6	T	0	0	2	0	10	1	0	2	8	0
7	A	0	5	0	7	1	7	0	5	0	5

H^x

		0	1	2	3	4	5	6	7	8	9
		-	A	G	A	T	G	C	A	G	G
0	-	0	0	0	0	0	0	0	0	0	0
1	T	1	1	1	1	0	1	1	1	1	1
2	A	2	1	2	1	2	0	2	1	2	2
3	T	3	3	1	3	1	1	1	3	1	3
4	C	4	4	4	4	1	1	1	4	4	4
5	A	5	4	5	4	1	5	1	1	1	1
6	T	6	6	4	6	4	4	4	1	1	6
7	A	7	6	7	4	4	4	7	4	7	1

H^y

		0	1	2	3	4	5	6	7	8	9
		-	A	G	A	T	G	C	A	G	G
0	-	0	1	2	3	4	5	6	7	8	9
1	T	0	1	2	3	3	5	6	7	8	9
2	A	0	0	2	2	4	3	6	6	8	9
3	T	0	1	0	3	2	2	2	7	6	9
4	C	0	1	2	3	2	2	2	7	8	9
5	A	0	0	2	2	2	5	2	2	2	2
6	T	0	1	0	3	2	2	2	2	2	9
7	A	0	0	2	0	2	2	6	2	8	2

Figure 4.1: Starting and ending positions of HSR

For example, Figure 4.1 shows H, H^x, and H^y matrices for the DNA sequences X = TATCATA and Y = AGATGCAGG where $s_m = 5, s_s = -3, g_o = -9$, and $g_e = -1$. In the figure, the shaded scores in H represents the optimal local alignment ending at $i^* = 5$ and $j^* = 7$ with score $H_{5,7} = 11$. The indices inside the boxes in H represent the ending positions, and the positions inside the boxes in H^x and H^y represent its source location in X and Y, respectively. In other words, the score of HSR is 11, it starts from the positions $H^x_{5,7} + 1 = 2$

and $H^y_{5,7} + 1 = 3$, and ends at 5 and 7, in X and Y, respectively.

4.2 Data Encoding

Since TFHE allows only bit-by-bit encryption, all the data used in the computation must be encoded by bit arrays. We have three different types of data to encode.

The first one is the sequences X and Y. Assume that they are defined over an alphabet Σ. Then a single base X_i or Y_j can be encoded by $w_{alp} = \lceil \log |\Sigma| \rceil$ bits. Thus X of length n and Y of length m are encoded by bit arrays of lengths $w_{alp} \times n$ and $w_{alp} \times m$, respectively. Note that log denotes a logarithm to the base two.

The second one is integer scores in H, P, and Q matrices in (2.1)-(2.4). Since the scores in the matrices can be negative, we use the two's complement for representing them. Assume that w_{sco} is the bit length of the two's complement representation. Then each score can be encoded as a bit array of length w_{sco} which can be calculated by the following equation based on the lengths of the sequences and scoring scheme.

$$w_{sco} = \max \begin{cases} \lceil \log(\min(n, m) \times s_m + 1) \rceil + 1 \\ \lceil \log(\max(-g_o - g_e, -s_s)) \rceil + 1 \end{cases} \tag{4.8}$$

Here, the expressions at the first and second rows in (4.8) are the bit lengths of the maximum and minimum scores that can be encountered during the computation, respectively. The maximum possible score is $\min(n, m) \times s_m$ when one sequence is a substring of the other. Therefore we need $\lceil \log(\min(n, m) \times s_m + 1) \rceil + 1$ bits to represent it. The minimum possible score that can be stored in the matrices is g_o according to (2.2)-(2.4) since the scores in H are non-negative. However, we may encounter the scores $g_o + g_e$ and s_s during the computations

in (2.2)-(2.4). Therefore, we need $\lceil \log(\max(-g_o - g_e, -s_s)) \rceil + 1$ bits to represent the minimum score.

The last one is integer positions in H^x, H^y, P^x, P^y, Q^x, and Q^y matrices in (4.1)-(4.7). Since the positions of the sequences are non-negative and not greater than n and m, they can be encoded as bit arrays of lengths

$$w_{pos} = \lceil \log(max(n, m) + 1) \rceil, \tag{4.9}$$

like unsigned integers. Note that the w_{alp}, w_{sco}, and w_{pos} are not required to be a power of two in our algorithm.

4.3 Homomorphic Computation of Algorithm

Algorithm 3 SW-SCORE($\overline{\mathsf{X}}, \overline{\mathsf{Y}}, \{s_m/s_s, g_o, g_e\}$)

Initialize $H, P, Q, H^x, H^y, P^x, P^y, Q^x$, and Q^y ▷ By (2.1) and (4.1)
for $d \leftarrow 1$ **to** $n + m - 1$ **do**
  $start \leftarrow \max(1, d - m + 1)$
  $end \leftarrow \min(d, n)$
  for $k \leftarrow start$ **to** end **parallel do**
    $i \leftarrow k$
    $j \leftarrow d - k + 1$
    $T_1 \leftarrow \mathsf{Add}(P_{i-1,j}, \overline{g}_e)$
    $T_2 \leftarrow \mathsf{Add}(H_{i-1,j}, \overline{g}_o)$
    $S \leftarrow \mathsf{Compare}(T_1, T_2)$
    $P_{i,j} \leftarrow \mathsf{Select}(S, T_1, T_2)$
    $P^x_{i,j} \leftarrow \mathsf{Select}(S, P^x_{i-1,j}, H^x_{i-1,j})$
    $P^y_{i,j} \leftarrow \mathsf{Select}(S, P^y_{i-1,j}, H^y_{i-1,j})$
    $T_1 \leftarrow \mathsf{Add}(Q_{i,j-1}, \overline{g}_e)$
    $T_2 \leftarrow \mathsf{Add}(H_{i,j-1}, \overline{g}_o)$
    $S \leftarrow \mathsf{Compare}(T_1, T_2)$
    $Q_{i,j} \leftarrow \mathsf{Select}(S, T_1, T_2)$
    $Q^x_{i,j} \leftarrow \mathsf{Select}(S, Q^x_{i,j-1}, H^x_{i,j-1})$
    $Q^y_{i,j} \leftarrow \mathsf{Select}(S, Q^y_{i,j-1}, H^y_{i,j-1})$
    $S \leftarrow \mathsf{Compare}(P_{i,j}, Q_{i,j})$
    $H_{i,j} \leftarrow \mathsf{Select}(S, P_{i,j}, Q_{i,j})$
    $H^x_{i,j} \leftarrow \mathsf{Select}(S, P^x_{i,j}, Q^x_{i,j})$
    $H^y_{i,j} \leftarrow \mathsf{Select}(S, P^y_{i,j}, Q^y_{i,j})$
    $E \leftarrow \mathsf{Equals}(\overline{\mathsf{X}}_i, \overline{\mathsf{Y}}_j)$
    $T_1 \leftarrow \mathsf{Select}(E, \overline{s}_m, \overline{s}_s)$
    $T_2 \leftarrow \mathsf{Add}(H_{i-1,j-1}, T_1)$
    $S \leftarrow \mathsf{Compare}(T_2, H_{i,j})$
    $H_{i,j} \leftarrow \mathsf{Select}(S, T_2, H_{i,j})$
    $H^x_{i,j} \leftarrow \mathsf{Select}(S, H^x_{i-1,j-1}, H^x_{i,j})$
    $H^y_{i,j} \leftarrow \mathsf{Select}(S, H^y_{i-1,j-1}, H^y_{i,j})$
    $S \leftarrow \mathsf{Compare}(H_{i,j}, \overline{0})$
    $H_{i,j} \leftarrow \mathsf{Select}(S, H_{i,j}, \overline{0})$
    $H^x_{i,j} \leftarrow \mathsf{Select}(S, H^x_{i,j}, \overline{i})$
    $H^y_{i,j} \leftarrow \mathsf{Select}(S, H^y_{i,j}, \overline{j})$
return H, H^x, H^y

We describe the homomorphic computation of our algorithm using the circuits in Chapter 3.

(1) Filling Matrices. SW-Score in Algorithm 3 takes a scoring scheme $\{s_m/s_s, g_o, g_e\}$, ciphertext arrays $\overline{\mathsf{X}}$ of length n and $\overline{\mathsf{Y}}$ of length m where the sequences X and Y are encrypted into as input, and outputs matrices H,H^x, and H^y filled with the encrypted scores and positions. More specifically, $\overline{\mathsf{X}}_i$ is encrypted X_i, which is a ciphertext array of length w_{alp}. In the algorithm, s_m, s_s, g_o, and g_e are represented by two's complement and encoded as bit arrays with lengths w_{sco}. The lines over the variables denote their ciphertexts which are encrypted by $\mathsf{EncConst}$ of the TFHE scheme. For instance, $\overline{g}_o$ denotes encrypted g_o.

First, the first rows and columns of the matrices are initialized according to (2.1) and (4.1) in line 1. Then we fill the matrices according to (2.2)-(2.4) and (4.2)-(4.7) in lines 2-34. To reduce the running time, we use the anti-diagonal parallel computation for the filling. Since elements on an anti-diagonal of the matrices depend on the elements on the previous anti-diagonal, they can be computed in parallel as described in line 5. The matrices have $n + m - 1$ anti-diagonals, and each anti-diagonal can have at most $\min(n, m)$ elements that can be calculated simultaneously. Here, all the outputs except the indices (lines 6 and 7) are encrypted, so they do not leak any information.

Algorithm 4 SW-FIND(H, H^x, H^y, ck)

1: Initialize E^x and E^y ▷ By (4.10)
2: **for** $d \leftarrow 1$ **to** $\lceil \log(n \times m) \rceil$ **do**
3: **for** $k \leftarrow 1$ **to** $n \times m$ **by** 2^d **parallel do**
4: $i \leftarrow \lfloor \frac{k-1}{m} \rfloor + 1$
5: $j \leftarrow (k-1) \pmod m + 1$
6: $i' \leftarrow \lfloor \frac{k+2^{d-1}-1}{m} \rfloor + 1$
7: $j' \leftarrow (k + 2^{d-1} - 1) \pmod m + 1$
8: **if** $i' \leq n$ **then**
9: $S \leftarrow \mathsf{Compare}(H_{i,j}, H_{i',j'})$
10: $H_{i,j} \leftarrow \mathsf{Select}(S, H_{i,j}, H_{i',j'})$
11: $H^x_{i,j} \leftarrow \mathsf{Select}(S, H^x_{i,j}, H^x_{i',j'})$
12: $H^y_{i,j} \leftarrow \mathsf{Select}(S, H^y_{i,j}, H^y_{i',j'})$
13: $E^x_{i,j} \leftarrow \mathsf{Select}(S, E^x_{i,j}, E^x_{i',j'})$
14: $E^y_{i,j} \leftarrow \mathsf{Select}(S, E^y_{i,j}, E^y_{i',j'})$
15: **return** $H_{1,1}, H^x_{1,1}, H^y_{1,1}, E^x_{1,1}, E^y_{1,1}$

(2) Finding HSR. SW-FIND in Algorithm 4 takes the output of SW-SCORE as input, and outputs the encrypted score, two starting positions, and two ending positions of the HSR. First, we use two extra matrices E^x and E^y for keeping the ending positions of HSR and initialize as follows.

$$E^x_{i,j} = \overline{i}, E^y_{i,j} = \overline{j} \text{ for } i = 1, \ldots, n \text{ and } j = 1, \ldots, m \tag{4.10}$$

Then we find the maximum score of H in parallel using a binary tree order while keeping the starting and ending positions of the HSR corresponding to the current maximum score as in lines 2-14. Finally, the encrypted maximum score, two starting positions, and two ending positions in X and Y of HSR will

be stored at $H_{1,1}, H^x_{1,1}, H^y_{1,1}, E^x_{1,1}$, and $E^y_{1,1}$, respectively.

Note that bit lengths w_{sco}, w_{pos}, w_{alp}, and a cloud key ck are omitted from the input of the algorithms and circuits for simplicity's sake.

4.4 Complexity Analysis

We analyze the spans and works of Algorithms 3 and 4. The unit of our analysis is a binary gate of TFHE as in Table 3.1. The span and work of Algorithm 3 are $\alpha \times (n + m - 1)$ and $\beta \times nm$ where α and β denote the span and work of its parallel region (lines 6-34), respectively. Since 5 Add, 5 Compare, 16 Select, and 1 Equals are used in the parallel region,

$$\begin{aligned}
\alpha &= 5 \times (2\lceil \log (w_{sco} - 1) \rceil + 2) + 5 \times (2\lceil \log w_{sco} \rceil + 2) \\
&\quad + 16 \times 2 + (\lceil \log w_{alp} \rceil + 1) \\
&= 10\lceil \log (w_{sco} - 1) \rceil + 10\lceil \log w_{sco} \rceil + \lceil \log w_{alp} \rceil + 53 \\
\beta &= 5 \times (3\lceil \frac{w_{sco} - 1}{2} \rceil \lceil \log (w_{sco} - 1) \rceil + 3w_{sco} - 1) \\
&\quad + 5 \times (4w_{sco} - 2) + 10 \times (2w_{pos}) + 6 \times (2w_{sco}) + (2w_{alp} - 1) \\
&= 15\lceil \frac{w_{sco} - 1}{2} \rceil \lceil \log (w_{sco} - 1) \rceil + 47w_{sco} + 20w_{pos} + 2w_{alp} - 16
\end{aligned}$$

according to Table 3.1 if the parallel circuits are used in the algorithm. Otherwise, i.e., when the sequential circuits are used,

$$\begin{aligned}
\alpha = \beta &= 5 \times 5w_{sco} + 5 \times 3w_{sco} + 6 \times 2w_{sco} + 10 \times 2w_{pos} + 2w_{alp} - 1 \\
&= 52w_{sco} + 20w_{pos} + 2w_{alp} - 1
\end{aligned}$$

according to Table 3.1. In other words, the span and work of SW-Score with the parallel circuits are $O((n + m) \log (w_{sco} w_{alp}))$ and $O(nm(w_{sco} \log w_{sco} + w_{pos} + w_{alp}))$, respectively. With the sequential circuits, they are $O((n+m)(w_{sco} + w_{pos} + w_{alp}))$ and $O(nm(w_{sco} + w_{pos} + w_{alp}))$, respectively.

Since Algorithm 4 finds the maximum score of matrix H in parallel using a binary tree order, its span and work are $\gamma \times \lceil \log nm \rceil$ and $\delta \times (nm - 1)$ where γ and δ denote the span and work of the parallel region (lines 4-14) of the algorithm, respectively. If we use the parallel circuits in the Algorithm 4,

$$\gamma = (2\lceil \log w_{sco} \rceil + 2) + 5 \times 2 = 2\lceil \log w_{sco} \rceil + 12$$
$$\delta = (4w_{sco} - 2) + (2w_{sco}) + 4 \times (2w_{pos}) = 6w_{sco} + 8w_{pos} - 2$$

otherwise (i.e., with the sequential circuits),

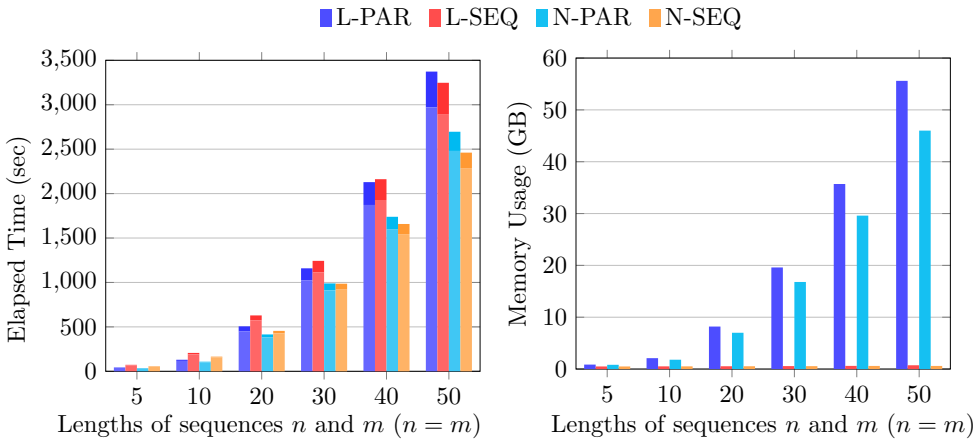
$$\gamma = \delta = 3w_{sco} + 2w_{sco} + 4 \times 2w_{pos} = 5w_{sco} + 8w_{pos}$$

according to Table 3.1. Thus, the span and work of SW-Find with the parallel circuits are $O(\log(nm) \log(w_{sco}))$ and $O(nm(w_{sco} + w_{pos}))$, respectively. With the sequential circuits, they are $O(\log(nm)(w_{sco} + w_{pos}))$ and $O(nm(w_{sco} + w_{pos}))$, respectively.

4.5 Performance Evaluation

Here we present experimental results to show the performance of our algorithm. We compare four versions of our algorithm:

- L-PAR: Reports the score and location of HSR using the parallel circuits.
- L-SEQ: Reports the score and location of HSR using the sequential circuits.
- N-PAR: Reports the score of HSR without the location using the parallel circuits.
- N-SEQ: Reports the score of HSR without the location using the sequential circuits.



(a) Elapsed time

(b) Memory usage

Figure 4.2: Comparison between four versions of our algorithm

We implement them in C++ using the TFHE library [28] and OpenMP. The experiments are conducted on a machine with two Intel Xeon Silver 4114 2.20 GHz CPUs (32 cores) and 256GB RAM running Debian Linux. Each binary gate of TFHE such as And, Or, Xor, and Xnor takes 22 milliseconds, and Mux takes 44 milliseconds on a single core of the server.

The security of TFHE relies on the hardness of the TLWE assumption which is a generalization of the Learning with Errors (LWE) problem over the torus [27]. The originally proposed security parameter in [27], $\lambda = 110$ bit, has been updated to $\lambda = 128$ bit due to the new estimates and attack models in [29]. Therefore, we use the recent one in the experiments. Since we use DNA sequences in the experiments, the bases can be encoded by $w_{alp} = 2$ bits as $A = 00_2, C = 01_2, G = 10_2$, and $T = 11_2$. For the scoring scheme, we use $\{5/-3, -9, -1\}$ as in [44]. Furthermore, the bit lengths $w_{sco} \in \{6, 7, 8, 9\}$ and $w_{pos} \in \{3, 4, 5, 6\}$ are used in the experiments, according to (4.8) and (4.9),

respectively.

In the experiment, we measure the average elapsed times and peak memory usages of the four versions in the different lengths of sequences $n, m \in \{5, 10, 20, 30, 40, 50\}$. Figure 4.2 shows the result of the experiment. In Figure 4.2a, the N-PAR and N-SEQ are faster than L-PAR and L-SEQ because the computations in (4.1)-(4.7) and keeping the location during the finding maximum (lines 11-14 in Algorithm 4) are excluded from Algorithms 3 and 4, respectively, in both N-PAR and N-SEQ. Since both the homomorphic circuits and the algorithms are performed in parallel in L-PAR and N-PAR, the nested parallelism of OpenMP is enabled in the experiment. L-SEQ outperforms L-PAR in $n, m > 40$, and N-SEQ outperforms N-PAR in $n, m \geq 30$. The reason is that when the number of the upper level threads (i.e., line 5 in Algorithm 3 and line 3 in Algorithm 4) is less than the number of the cores (32), we can get the speedup of the parallel circuits. However, when it becomes greater than the number of the cores, we cannot get the speedup due to the overhead. Thus, L-SEQ and N-SEQ outperform L-PAR and N-PAR as the sequences get longer (i.e., longer than the number of cores), respectively. Note that the light and bright colored sections of each bar in Figure 4.2a indicate the average elapsed times of SW-Score and SW-Find, respectively.

In Figure 4.2b, the peak memory usages of L-SEQ and N-SEQ are less than a gigabyte whereas those of L-PAR and N-PAR reach to several gigabytes due to the nested parallelism. Therefore, there is a trade-off between time and memory in using the sequential and parallel circuits when $n, m \leq 40$.

Computing the edit distance of homomorphically encrypted sequences [26] is similar to our problem with respect to the computation methods. Since computing the edit distance of only two sequences is inefficient in [26], the experiment is done for multiple pairs of sequences in order to reduce the amortized time

which is equal to the total running time divided by the number of pairs. In particular, it takes 5 hours 13 minutes for 682 pairs of DNA sequences of lengths $n, m = 8$ when $\lambda = 80$ bit. In addition, the lengths are limited to $n, m = 8$ due to the huge memory requirement in [26]. Even though the finding HSR is more costly computation than the edit distance, our implementation shows much better performances for a pair of sequences so as to be more practical.

Chapter 5

Finding Approximate HSR

Given two genomic sequences represented by lists of encrypted variants, we find the score and location of HSR between them approximately in encrypted forms. Unlike the previous problem in Chapter 4, we assume that the two input sequences are aligned to the same reference genome, and both are stored in VCF files, i.e., represented by lists of variants as described in Section 2.3. Then we define the approximate HSR of the two sequences, and propose an approach to find its score and location.

Since the two sequences have already been aligned to the same reference genome, we can align one sequence to the other based on the provided information, i.e., the variants. From our observations, two types of regions alternate in the alignment between them (see Figure 5.1); one appears around each pair of variants at the same positions; we call it a variant region (e.g., shadowed regions in Figure 5.1), and the other appears between those variant regions; we call it a matched region (e.g., non-shadowed regions in Figure 5.1). To find the HSR of the alignment, we need the total score of each region. Computing

the total score of each matched region is straightforward because it consists of only matches. However, computing the total score of each variant region is complicated because aligning the two subsequences in the variant region is required. We briefly describe our approaches for aligning the variant regions and finding the HSR of the alignment from the total scores of all the regions in the following.

Aligning Variant Region. To compute the total score of each variant region, we need to align each pair of variants at the same positions in the input lists. Since aligning a pair of variants using their individual bases is expensive in HE (i.e., global alignment by the Needleman–Wunsch algorithm [58] is needed), we present a method to align them efficiently. In our method, we assume that the alignment consists of two parts; a gap and consecutive matches/mismatches. In other words, the alignment consists of a gap (with any length) and consecutive matches if the two alternate bases of the pair are exactly the same, and it consists of a gap and consecutive mismatches otherwise. By this assumption, we can obtain the lengths of the gap and matches/mismatches efficiently. Once we have these lengths in each variant region, we can compute the starting and ending positions of each matched region simply. Moreover, the total scores of all the regions can be computed for a scoring scheme directly based on those lengths and positions.

Finding HSR. We assume that the HSR of the two sequences is identical to a contiguous subarray with the largest sum within the array of the total scores (e.g., the last row in Figure 5.4). More specifically, the score of HSR is equal to the sum of scores in the contiguous subarray, and the location of HSR is equivalent to the starting and ending positions of the subarray. Since it is a classic problem; called the maximum subarray problem, we can obtain the score and location of HSR using the known algorithms for the problem.

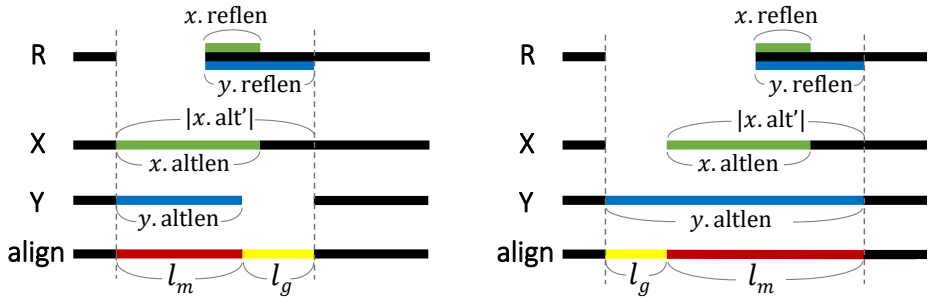
Organization. The remaining of this chapter is organized as follows. Section 5.1 defines the approximate HSR. Section 5.2 gives an overview of our approach. Sections 5.3 and 5.4 introduce the homomorphic computations of our algorithm using the bit-wise and word-wise HE schemes, respectively. Section 5.5 shows the experimental results of our algorithm.

5.1 Approximate HSR

pos	1	2			3	4	5	6	7	8	9	10	11	12	13	14	15	16	17				18	19	20	21	22	23	24	25	26	27
R	A	T			G	A	A	T	A	C	T	C	T	G	C	T	T	A	A				G	C	A	A	G	C	G	G	T	T
X	A	T	**A**	**A**	**T**	A	A	T	A	C	T	–	–	G	**A**	**G**	–	A	A				G	C	A	A	–	–	**A**	**G**	T	T
Y	A	T	**C**	**C**	–	–	–	–	A	C	T	**A**	T	G	**A**	**G**	–	–	A	**C**	**C**	**C**	G	C	A	A	–	**T**	**A**	**C**	**C**	T
align	M	M	S	S	–	–	–	–	M	M	M	–	–	M	M	M		–	M	–	–	–	M	M	M	M		–	S	S	S	M
score	5	5	-3	-3	-9	-1	-1	-1	5	5	5	-9	-1	5	5	5		-9	5	-9	-1	-1	5	5	5	5		-9	-3	-3	-3	5

Figure 5.1: Alignment between two DNA sequences represented by lists of variants

Since two sequences represented by lists $\mathbb{X}$ and $\mathbb{Y}$ are individually aligned to reference genome R already, they can be aligned with each other, for example, as shown in Figure 5.1. This alignment can be obtained from the lists by aligning each pair of variants at the same positions. Then we can approximate the HSR of them obtained by SW as a *highly similar region* of this alignment with the largest sum of scores. For example, row align in Figure 5.1 shows an alignment that is derived from $\mathbb{X}$ and $\mathbb{Y}$ in Figure 2.2. In the figure, row pos shows the positions of R, and row score shows the corresponding scores for the matches, mismatches, and gaps of the alignment. The alternate bases in Figure 2.2 are in boldface in X and Y. In this alignment, a HSR lies between reference positions 7 and 21, and its score is 25 (sum of the scores in the region). Here, we use the



(a) $|x.\texttt{alt}'| \geq |y.\texttt{alt}|$ (b) $|x.\texttt{alt}'| < |y.\texttt{alt}|$

Figure 5.2: Two ways to align variant region

same scoring scheme as in Figure 2.1.

5.2 Algorithm Overview

Our algorithm for finding approximate HSR consists of the following two steps.

(1) Computing Scores of Regions. There are two types of regions of the alignment between X and Y in Figure 5.1: shadowed and non-shadowed. A shadowed region appears at the position of a pair of variants; we call it a *variant region*. A variant region may consist of matches, mismatches, and gaps. A non-shadowed region can appear before the first, after the last, and between two variant regions; we call it a *matched region* since it consists of only matches. Therefore, instead of computing the scores letter by letter, we compute the total score of each variant and matched regions. To compute them, we need to align a pair of variants in each variant region and compute the location of each matched region. Note that there are $|\mathbb{X}|$ variant regions and $|\mathbb{X}| + 1$ matched regions in the alignment, where $|\mathbb{X}|$ denotes the number of the variants in $\mathbb{X}$ ($|\mathbb{X}| = |\mathbb{Y}|$).

a) Aligning variant region: Let x and y be a pair of variants in $\mathbb{X}$ and

$\mathbb{Y}$ at the same positions, i.e., $x = \mathbb{X}_i$ and $y = \mathbb{Y}_i$ for $1 \leq i \leq |\mathbb{X}|$. Assume first that $x.\texttt{reflen} \leq y.\texttt{reflen}$. Let $x.\texttt{alt}'$ denote the concatenation of $x.\texttt{alt}$ and the difference of reference bases in x and y. Our task is to align $x.\texttt{alt}'$ and $y.\texttt{alt}$. For example, in Figure 5.1, when $x = \mathbb{X}_1$ and $y = \mathbb{Y}_1$, we need to align $x.\texttt{alt}' = \text{AATAAT}$ and $y.\texttt{alt} = \text{CC}$, where the last three bases of $x.\texttt{alt}'$, AAT, is the difference of their reference bases GAAT and G. In homomorphic encryption, computing an alignment using individual bases is expensive. Therefore, we propose an approach to align them efficiently. Depending on $|x.\texttt{alt}'|$ and $|y.\texttt{alt}|$, we align them in two different ways as illustrated in Figure 5.2. In the figure, the green colored parts of R and X indicate the reference and alternate bases of x, respectively, and the blue colored parts of R and Y indicate the reference and alternate bases of y, respectively. Figure 5.2a shows the way to align if $|x.\texttt{alt}'| \geq |y.\texttt{alt}|$, and Figure 5.2b shows the other way to align, i.e., if $|x.\texttt{alt}'| < |y.\texttt{alt}|$. In the figures, the alignment between $x.\texttt{alt}'$ and $y.\texttt{alt}$ consists of only two parts; a gap (the yellow colored part) and consecutive matches/mismatches (the red colored part). Let l_g be the length of the gap and l_m be the length of consecutive matches/mismatches of the alignments (we decide l_m is the length of either matches or mismatches later). In Figure 5.2a, we can see that $l_m = |y.\texttt{alt}|$ and $l_g = |x.\texttt{alt}'| - |y.\texttt{alt}|$, i.e.,

$$l_m = y.\texttt{altlen} \tag{5.1}$$

$$l_g = x.\texttt{altlen} + (y.\texttt{reflen} - x.\texttt{reflen}) - y.\texttt{altlen} \tag{5.2}$$

since $|x.\texttt{alt}'| = x.\texttt{altlen} + (y.\texttt{reflen} - x.\texttt{reflen})$. For example, the first variant region, $\mathbb{X}_1$ and $\mathbb{Y}_1$, is aligned by this way in Figure 5.1, thus $l_m = 2$ and $l_g = 3 + (4 - 1) - 2 = 4$ according to (5.1) and (5.2), i.e., $l_m = |y.\texttt{alt}| = |\text{CC}| = 2$ and $l_g = |x.\texttt{alt}'| - |y.\texttt{alt}| = |\text{AATAAT}| - |\text{CC}| = 4$. Moreover, in Figure 5.2b,

$l_m = |x.\texttt{alt}'|$ and $l_g = |y.\texttt{alt}| - |x.\texttt{alt}'|$, i.e.,

$$l_m = x.\texttt{altlen} + (y.\texttt{reflen} - x.\texttt{reflen}) \tag{5.3}$$

$$l_g = y.\texttt{altlen} - x.\texttt{altlen} - (y.\texttt{reflen} - x.\texttt{reflen}). \tag{5.4}$$

For example, the last variant region, $\mathbb{X}_5$ and $\mathbb{Y}_5$, is aligned by this way in Figure 5.1, thus $l_m = 2 + (5 - 4) = 3$ and $l_g = 4 - 2 - (5 - 4) = 1$ according to (5.3) and (5.4), i.e., $l_m = |x.\texttt{alt}'| = |\text{AGT}| = 3$ and $l_g = |y.\texttt{alt}| - |x.\texttt{alt}'| = |\text{TACC}| - |\text{AGT}| = 1$. Note that if $|y.\texttt{alt}| = |x.\texttt{alt}'|$, (5.1) and (5.3) are equivalent, (5.2) and (5.4) as well.

Now let us consider the case $x.\texttt{reflen} > y.\texttt{reflen}$. This case can be handled by swapping x and y in Figures 5.2a, 5.2b, and (5.1)-(5.4). Therefore, we have four different ways to align in the general case, i.e., each of l_m and l_g can be expressed by the four different equations. By observing the symmetry of x and y in the equations, we can combine the four equations into one, which is more suitable for HE, i.e.,

$$l_m = l_{ref} + l_{min} \tag{5.5}$$

$$l_g = l_{max} - l_{min} \tag{5.6}$$

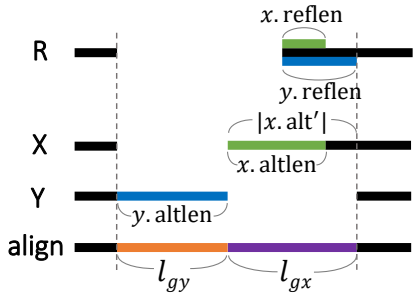
where l_{ref}, l_{max}, and l_{min} are defined as

$$l_{ref} = \max(x.\texttt{reflen}, y.\texttt{reflen}) \tag{5.7}$$

$$l_{max} = \max(x.\texttt{altlen} - x.\texttt{reflen}, y.\texttt{altlen} - y.\texttt{reflen}) \tag{5.8}$$

$$l_{min} = \min(x.\texttt{altlen} - x.\texttt{reflen}, y.\texttt{altlen} - y.\texttt{reflen}). \tag{5.9}$$

If $x.\texttt{alt}$ and $y.\texttt{alt}$ are exactly the same, the alignment consists of a gap of length l_g and consecutive matches of length l_m (e.g., the alignment between $\mathbb{X}_3$ and $\mathbb{Y}_3$ in Figure 5.1). Otherwise (i.e., at least one mismatch appears between $x.\texttt{alt}$ and $y.\texttt{alt}$ or their lengths are different), the alignment consists of a gap of



(a) Alignment

pos	...	20	21				22	23	24	25	26	27
R	...	A	A				G	C	G	G	T	T
X	...	A	A				–	–	A	G	T	T
Y	...	A	A	T	A	C	C	–				T
align	...	M	M	–	–	–	–	–	–	–	–	M
score	...	5	5	-9	-1	-1	-1	-9	-1	-1		5

(b) Aligning $\mathbb{X}_5$ and $\mathbb{Y}_5$

Figure 5.3: Alignment consists of two gaps

length l_g and consecutive mismatches of length l_m (e.g., the alignment between $\mathbb{X}_1$ and $\mathbb{Y}_1$ in Figure 5.1). Therefore, the total score of the alignment, denoted by v', with respect to a scoring scheme $\{s_m/s_s, g_o, g_e\}$ can be computed as

$$v' = W(l_g) + \begin{cases} l_m \times s_m & \text{if } x.\texttt{alt} = y.\texttt{alt} \\ l_m \times s_s & \text{if } x.\texttt{alt} \neq y.\texttt{alt} \end{cases} \tag{5.10}$$

where W is an affine gap penalty function for a gap of length l defined as follows.

$$W(l) = \begin{cases} 0 & \text{if } l = 0 \\ (l-1) \times g_e + g_o & \text{if } l > 0 \end{cases} \tag{5.11}$$

For example, l_m, l_g, and v' of each variant region of the alignment in Figure 5.1, computed according to (5.5)-(5.11), can be shown in Figure 5.4.

In addition, we consider another way to align a variant region, in which the alignment consists of two gaps, without any matches or mismatches (as indels), as illustrated in Figure 5.3a. Depending on a scoring scheme, the total score of this alignment can be higher than v', the total score of the previous way to align. Similarly to the previous way to align, let us assume $x.\texttt{reflen} \leq y.\texttt{reflen}$. Let l_{gx} and l_{gy} be the lengths of the two gaps that appear in Y and X, respec-

tively (i.e., the lengths of the purple and orange colored parts of the alignment in Figure 5.3a). In the figure, $l_{gx} = |x.\texttt{alt}'|$ and $l_{gy} = |y.\texttt{alt}|$, i.e., $l_{gx} = x.\texttt{altlen} + (y.\texttt{reflen} - x.\texttt{reflen})$ and $l_{gy} = y.\texttt{altlen}$. In the other case $x.\texttt{reflen} > y.\texttt{reflen}$, $l_{gx} = x.\texttt{altlen}$ and $l_{gy} = y.\texttt{altlen} + (x.\texttt{reflen} - y.\texttt{reflen})$. Considering the symmetry of x and y, we obtain the following equations that express them in the general case.

$$l_{gx} = x.\texttt{altlen} + (l_{ref} - x.\texttt{reflen}) \tag{5.12}$$

$$l_{gy} = y.\texttt{altlen} + (l_{ref} - y.\texttt{reflen}) \tag{5.13}$$

where l_{ref} is defined in (5.7). Figure 5.3b shows the variant region, $\mathbb{X}_5$ and $\mathbb{Y}_5$, aligned by this way. Thus $l_{gx} = 2+5-4 = 3$ and $l_{gy} = 4+5-5 = 4$ according to (5.12) and (5.13), i.e., $l_{gx} = |x.\texttt{alt}'| = |\text{AGT}| = 3$ and $l_{gy} = |y.\texttt{alt}| = |\text{TACC}| = 4$, as highlighted in purple and orange colors in the figure, respectively. Let v'' be the total score of this alignment. Then it can be computed simply as

$$v'' = W(l_{gx}) + W(l_{gy}). \tag{5.14}$$

For example, l_{gx}, l_{gy}, and v'' of each variant region of the alignment in Figure 5.1, computed according to (5.12)-(5.14), are also shown in Figure 5.4.

Finally, the higher one of v' and v'' will be selected as the total score of the variant region, and we denote it by v, i.e.,

$$v = \max(v', v''). \tag{5.15}$$

b) Computing location of matched region: Let x, y and x', y' be pairs of variants in two consecutive variant regions, i.e., $x = \mathbb{X}_i$, $y = \mathbb{Y}_i$, $x' = \mathbb{X}_{i+1}$ and $y' = \mathbb{Y}_{i+1}$ for $1 \leq i < |\mathbb{X}|$, and let s, e, and m be the starting position, ending position, and total score of the matched region between the two variant

regions, respectively. Then they can be computed as

$$s = \max(x.\texttt{reflen}, y.\texttt{reflen}) + x.\texttt{pos} \tag{5.16}$$

$$e = x'.\texttt{pos} - 1 \tag{5.17}$$

$$m = (e - s + 1) \times s_m \tag{5.18}$$

where s_m is the match score of a scoring scheme. For the first matched region, we use $s = 1$ and $x' = \mathbb{X}_1$, and for the last matched region, we use $x = \mathbb{X}_{|\mathbb{X}|}$, $y = \mathbb{Y}_{|\mathbb{X}|}$, and $e = |\mathsf{R}|$. For example, in Figure 5.1, the matched region between $\mathbb{X}_4, \mathbb{Y}_4$ and $\mathbb{X}_5, \mathbb{Y}_5$ starts from position $s = \max(0, 0) + 18 = 18$ and ends at position $e = 22 - 1 = 21$, thus, its total score $m = (21 - 18 + 1) \times 5 = 20$, according to (5.16)-(5.18). Moreover, s, e, and m of each matched region of the alignment in Figure 5.1 are shown in Figure 5.4.

(2) Finding HSR. Now we find a HSR between X and Y. It is equivalent to finding a contiguous subarray with the largest sum from the total scores of variant and matched regions. This problem is called the *maximum subarray problem* [7], and linear time sequential and logarithmic time parallel algorithms are proposed in [7, 61]. Homomorphic computations of our algorithms for finding a HSR based on [7] and [61] are described in Sections 5.3 and 5.4, respectively.

For example, Figure 5.4 shows the lengths, total scores and locations of variant and matched regions of the alignment in Figure 5.1, computed according to (5.5)-(5.18). In the figure, row $m \cup v$ shows the total score of each region. Here, a HSR (i.e., contiguous subarray with the largest sum) from the total scores in $m \cup v$ starts from position $s = 7$ and ends at position $e = 21$ (the shadowed scores), and its score is 25 (sum of the shadowed scores), as in Figure 5.1.

i	1	2	3	4	5
l_m	2	0	2	0	3
l_g	4	2	1	3	1
v'	-18	-10	1	-11	-18
l_{gx}	6	0	3	0	3
l_{gy}	2	2	2	3	4
v''	-24	-10	-21	-11	-23
v	-18	-10	1	-11	-18

s	1	7	12	17	18	27
e	2	9	12	17	21	27
m	10	15	5	5	20	5

$m \cup v$	10	-18	15	-10	5	1	5	-11	20	-18	5

Figure 5.4: Lengths, scores, and locations of variant and matched regions in Figure 5.1

5.3 Homomorphic Computation of Bit-wise Algorithm

We present a homomorphic computation of our algorithm using the bit-wise HE scheme in Section 2.4.

5.3.1 Data Encoding

Since the TFHE scheme allows only bit-by-bit encryption, all the data used in the computations must be encoded by bit arrays. In $\mathbb{X}$ and $\mathbb{Y}$, we have two types of data in each variant. The first one is integers in `pos`, `reflen`, and `altlen` columns. To encode the integers, we use two's complement for representing them. Thus, each of them can be encoded by a bit array of length w_{int}, where

Algorithm 5 BT-Score($\overline{\mathbb{X}}$, $\overline{\mathbb{Y}}$, $|\mathsf{R}|$, $\{s_m/s_s, g_o, g_e\}$)

$S_1 \leftarrow \mathsf{EncConst}(1)$
$E_1 \leftarrow \mathsf{SubtractOne}(\overline{\mathbb{X}}_1.\mathtt{pos})$
$M_1 \leftarrow \mathsf{MultiplyConst}(E_1, s_m)$
for $i \leftarrow 1$ **to** $|\overline{\mathbb{X}}|$ **parallel do**
    $V_i \leftarrow$ BT-VarScore($\overline{\mathbb{X}}_i, \overline{\mathbb{Y}}_i, \{s_m/s_s, g_o, g_e\}$) ▷ by (5.5)-(5.15)
    $S_{i+1} \leftarrow \mathsf{Add}(\bar{l}_{ref}, \overline{\mathbb{X}}_i.\mathtt{pos})$
    if $i < |\overline{\mathbb{X}}|$ **then**
        $T \leftarrow \overline{\mathbb{X}}_{i+1}.\mathtt{pos}$
    else
        $T \leftarrow \mathsf{EncConst}(|\mathsf{R}| + 1)$
    $E_{i+1} \leftarrow \mathsf{SubtractOne}(T)$
    $M_{i+1} \leftarrow \mathsf{MultiplyConst}(\mathsf{Subtract}(T, S_{i+1}), s_m)$
return V, M, S, E

w_{int} is a bit length of the representation calculated as follows.

$$w_{int} = \max \begin{cases} \lceil \log ((|\mathsf{R}| + |\mathbb{X}| \times b) \times s_m + 1) \rceil + 1 \\ \lceil \log (|\mathbb{X}| \times |W(a+b) + \max((a+b) \times s_s, W(b))|) \rceil + 1 \end{cases} \tag{5.19}$$

where a and b denote the maximum values in `reflen` and `altlen` columns, respectively. In (5.19), the first and second rows are the bit lengths of the maximum and minimum scores which can be encountered during the computation, respectively. The second type of data is alternate bases in `alt` columns. Assume that they are defined over an alphabet Σ. Then, the alternate bases can be encoded by bit arrays of length $w_{alt} = \lceil \log |\Sigma| \rceil \times k$, where k is a fixed length for the alternate bases.

5.3.2 Computing Scores of Regions

BT-Score in Algorithm 5 takes lists of encrypted variants $\overline{\mathbb{X}}$ and $\overline{\mathbb{Y}}$ (variants in $\mathbb{X}$ and $\mathbb{Y}$ encrypted by the TFHE scheme as described above), $|\mathsf{R}|$, and a scoring scheme as input, and outputs the total scores of variant and matched regions with locations. In the algorithm, list V of size $|\mathbb{X}|$ stores the total scores of variant regions (i.e., row v in Figure 5.4), list M of size $|\mathbb{X}|+1$ stores the total scores of matched regions (i.e., row m in Figure 5.4), lists S and E of size $|\mathbb{X}|+1$ store the starting and ending positions of matched regions (i.e., rows s and e in Figure 5.4), respectively. Initially, we compute the starting position, ending position, and total score of the first matched region (lines 1-3). Algorithm BT-VarScore computes the total score of a variant region for a given scoring scheme according to Equations (5.5)-(5.15) (line 5). Next, the starting position, ending position, and total score of a matched region are computed according to (5.16)-(5.18) in lines 6, 7-11, and 12, respectively. In line 6, $\bar{l}_{ref}$ denotes the encrypted l_{ref} computed in BT-VarScore according to (5.7). Since there are no overlapping positions between variant regions, the above computations can be performed in parallel as described in line 4. In Algorithm 5, we use the sequential circuits because using the parallel circuits may incur overhead due to the nested parallel computations. Note that BT-VarScore can be implemented using the sequential circuits in Chapter 3 simply according to the equations.

5.3.3 Finding HSR

BT-Find in Algorithm 6 takes the output of BT-Score as input, and outputs the score, starting position, and ending position of a HSR. More specifically, it finds a contiguous subarray with the largest sum from the total scores in V and

Algorithm 6 BT-FIND(V, M, S, E)

1: $m \leftarrow M_1$
2: $s \leftarrow S_1$
3: $e \leftarrow E_1$
4: $score \leftarrow m$
5: $start \leftarrow s$
6: **for** $i \leftarrow 1$ **to** $|V|$ **do**
7: $\quad t \leftarrow \mathsf{Add}(score, V_i)$
8: $\quad c \leftarrow \mathsf{Not}(t_w)$ ▷ t_w is the sign (i.e. leftmost) bit of t
9: $\quad score \leftarrow \mathsf{Add}(\mathsf{SelectZero}(c, t), M_{i+1})$ ▷ If $t < 0$, then $score \leftarrow M_{i+1}$
10: $\quad start \leftarrow \mathsf{Select}(c, start, S_{i+1})$
11: $\quad c \leftarrow \mathsf{Compare}(score, m)$
12: $\quad m \leftarrow \mathsf{Select}(c, score, m)$
13: $\quad s \leftarrow \mathsf{Select}(c, start, s)$
14: $\quad e \leftarrow \mathsf{Select}(c, E_{i+1}, e)$
15: **return** m, s, e

M while keeping track of its starting and ending positions in S and E, based on Kadane's algorithm [7]. Unlike [7], BT-FIND iterates only $|V|$ times whereas Kadane's algorithm iterates $|V| + |M|$ times. The reason is that a HSR always starts from and ends at matched regions (i.e., in M) because the total score of a matched region is always non-negative. Since Algorithm 6 is sequential, we use the parallel circuits in Chapter 3 in lines 7-14 to improve the performance.

In Algorithms 5 and 6, all the outputs except the indices are encrypted, so they do not leak any information. Note that bit lengths w_{int}, w_{alt}, and a cloud key ck are omitted from the input of the algorithms and circuits for simplicity's sake.

Table 5.1: Number of circuits in parallel region (lines 5-12) of Algorithm 5

Circuit	Line 5 (BT-VarScore)											Line 11	Line 12	Total
	Eq. (5.5)	(5.6)	(5.7)	(5.8)	(5.9)	(5.10)	(5.11)	(5.12)	(5.13)	(5.14)	(5.15)			
Add	1	-	-	-	-	2	1	1	1	3	-	-	-	8
Compare	-	-	1	1	-	-	-	-	-	-	1	-	-	3
Equals	-	-	-	-	-	2	-	-	-	-	-	-	-	2
EqualsZero	-	-	-	-	-	1	1	-	-	2	-	-	-	3
MultiplyConst	-	-	-	-	-	3	1	-	-	2	-	-	1	6
Select	-	-	1	1	1	1	-	-	-	-	1	-	-	5
SelectZero	-	-	-	-	-	1	1	-	-	2	-	-	-	3
Subtract	-	1	-	2	-	-	-	-	-	-	-	-	1	4
SubtractOne	-	-	-	-	-	1	1	-	-	2	-	1	-	4

5.3.4 Complexity Analysis

We analyze the spans and works of Algorithms 5 and 6. The unit of our analysis is a binary gate of TFHE as in Table 3.1. Table 5.1 shows the number of the circuits used in the parallel region (lines 5-12) of Algorithm 5. Since BT-VarScore in line 5 computes the total score of a variant region according to Equations (5.5)-(5.15), we show the number of circuits for each equation in separate columns with the equation numbers in the table. In columns (5.7), (5.8), and (5.15), finding the minimum or maximum of two (encrypted) integers can be implemented by two circuits Compare and Select. In column (5.9), one Select is enough to find the minimum by reusing the outputs in (5.8). In column (5.10), to check $x.\texttt{alt} = y.\texttt{alt}$, we use two Equals: one for $x.\texttt{alt}$ and $y.\texttt{alt}$ of bit length w_{alt}, and the other for $x.\texttt{altlen}$ and $y.\texttt{altlen}$ of bit length w_{int}. Then we merge their results into one variable using And. Since the number of the circuits in column (5.11) (the gray colored number) is included in columns

(5.10) and (5.14), it is excluded from the total number in the last column. In columns (5.12) and (5.13), we need one Add in each column by reusing the outputs in (5.8). The computations in lines 7-10 of Algorithm 5 can be ignored because their running times are negligible. Let us denote by λ the span of the parallel region (lines 5-12) of Algorithm 5. Since we use the sequential circuits inside the parallel region, λ can be calculated from Table 5.1 using Table 3.1 as

$$\begin{aligned}\lambda =& 8 \times 5w_{int} + 3 \times 3w_{int} + ((2w_{alt} - 1) + (2w_{int} - 1) + 1) \\ &+ 3 \times (w_{int} - 1) + 6 \times (5\alpha w_{int} + 2w_{int}) \\ &+ 5 \times 2w_{int} + 3 \times w_{int} + 4 \times 5w_{int} + 4 \times 2w_{int} \\ =& 30\alpha w_{int} + 107w_{int} + 2w_{alt} - 4\end{aligned} \tag{5.20}$$

where $\alpha = \lceil \log(|t|+1) \rceil$ and t is a multiplicand of MultiplyConst in Algorithm 2. Thus its work, i.e., the total number of the homomorphic gates in the parallel region, is equal to $\lambda \times |\mathbb{X}|$. Therefore, considering the two circuits in lines 2 and 3, the span and work of Algorithm 5 are $\lambda + 5\alpha w_{int} + 4w_{int}$ and $\lambda \times |\mathbb{X}| + 5\alpha w_{int} + 4w_{int}$, respectively. In other words, the span and work of BT-Score are $O(w_{int} + w_{alt})$ and $O(|\mathbb{X}|(w_{int} + w_{alt}))$, respectively. Since we use six MultiplyConst in each iteration of Algorithm 5, we can estimate α for a scoring scheme $\{s_m/s_s, g_o, g_e\}$ as follows.

$$\alpha = \frac{2\lceil \log(|s_m| + 1) \rceil + \lceil \log(|s_s| + 1) \rceil + 3\lceil \log(|g_e| + 1) \rceil}{6} \tag{5.21}$$

For example, $\alpha = \frac{11}{6}$ for scoring scheme $\{5/-3, -9, -1\}$ according to (5.21).

Now we analyze the span and work of Algorithm 6. Each iteration of the algorithm consists of two Add, one Compare, four Select, and one SelectZero. Thus its span is equal to

$$\begin{aligned}&\big(2 \times (2\lceil \log (w_{int} - 1) \rceil + 2) + (2\lceil \log w_{int} \rceil + 2) + 4 \times 2 + 1\big) \times |\mathbb{X}| \\ &= \big(4\lceil \log (w_{int} - 1) \rceil + 2\lceil \log w_{int} \rceil + 15\big) \times |\mathbb{X}|\end{aligned} \tag{5.22}$$

and its work is

$$\begin{aligned}&\Big(2 \times (3 \left\lceil \frac{w_{int}-1}{2} \right\rceil \lceil \log{(w_{int}-1)} \rceil + 3w_{int} - 1) \\ &+ (4w_{int} - 2) + 4 \times 2w_{int} + w_{int}\Big) \times |\mathbb{X}| \\ &= \Big(6 \left\lceil \frac{w_{int}-1}{2} \right\rceil \lceil \log{(w_{int}-1)} \rceil + 19w_{int} - 4\Big) \times |\mathbb{X}|\end{aligned} \tag{5.23}$$

according to Table 3.1 (the parallel circuits) since $|V| = |\mathbb{X}|$. In other words, the span and work of BT-Find are $O(|\mathbb{X}| \log w_{int})$ and $O(|\mathbb{X}| w_{int} \log w_{int})$, respectively.

Since w_{alt} is a fixed length, we can conclude that BT-Score and BT-Find use $O(|\mathbb{X}| w_{int})$ and $O(|\mathbb{X}| w_{int} \log w_{int})$ homomorphic gates, respectively.

5.4 Homomorphic Computation of Word-wise Algorithm

We present a homomorphic computation of our algorithm using the word-wise HE scheme in Section 2.5.

5.4.1 Data Encoding

Since the HEAAN scheme supports computation of real numbers, we do not need a special encoding for integers in columns `pos`, `reflen`, and `altlen` of $\mathbb{X}$ and $\mathbb{Y}$. Instead, integers in each column must be stored into a single plaintext vector with n slots. Since n must be a power of two, we select $n = 2^{\lceil \log(2|\mathbb{X}|+1) \rceil + 1}$. To obtain the total scores of matched and variant regions in different slots of two ciphertext vectors (i.e., at the odd and even slots, as shown in Figure 5.6), we store variants in $\mathbb{X}$ and $\mathbb{Y}$ at the even slots of plaintext vectors as shown in Figure 5.5 (variants in $\mathbb{X}$ are omitted in the figure). Moreover, because of the limited input range of Equals (between 0 and 70), we divide alternate bases in

#slot	1	2	3	4	5	6	7	8	9	10	11	12	...	32
$\mathbb{Y}$.pos	0	3	0	10	0	13	0	18	0	22	0	0	...	0
$\mathbb{Y}$.reflen	0	4	0	1	0	4	0	0	0	5	0	0	...	0
$\mathbb{Y}$.altlen	0	2	0	1	0	2	0	3	0	4	0	0	...	0
$\mathbb{Y}$.alt$_1$	0	20	0	0	0	8	0	21	0	49	0	0	...	0
$\mathbb{Y}$.alt$_2$	0	0	0	0	0	0	0	0	0	16	0	0	...	0

Figure 5.5: Encoding of variants

column `alt` into $\lceil w_{alt}/6 \rceil$ plaintext vectors where w_{alt} denotes a bit length of the alternate bases ($\lfloor \log 71 \rfloor = 6$), the same as in the bit-wise algorithm, i.e., $w_{alt} = \lceil \log |\Sigma| \rceil \times k$ where k is a fixed length.

For example, Figure 5.5 shows the encoding of variants in $\mathbb{Y}$ in Figure 2.2 (variants in $\mathbb{X}$ in Figure 2.2 can be encoded in the same way as in $\mathbb{Y}$). In the figure, the variants are stored at the even slots of the plaintext vectors. We select $k = 4$, thus $w_{alt} = 2 \times 4 = 8$. Therefore the alternate bases can fit in two vectors since $\lceil w_{alt}/6 \rceil = 2$. For instance, AG is encoded by two 6 bits as $001000_2 = 8$ and $000000_2 = 0$ since a DNA base can be encoded by 2 bits as $A = 00_2, C = 01_2, G = 10_2$, and $T = 11_2$. Even though AGA can be encoded the same as AG, their lengths (i.e., `altlen`) are different. Thus, there is no ambiguity when checking equality between them.

5.4.2 Computing Scores of Regions

WD-Score in Algorithm 7 takes lists of encrypted variants $\overline{\mathbb{X}}$ and $\overline{\mathbb{Y}}$, $|\mathsf{R}|$, and a scoring scheme as input, and outputs the total scores of variant and matched regions. In the algorithm, a vector V stores the total scores of variant regions at its even slots, and a vector M stores the total scores of matched

Algorithm 7 WD-Score($\overline{\mathbb{X}}$, $\overline{\mathbb{Y}}$, $|\mathsf{R}|$, $\{s_m/s_s, g_o, g_e\}$)

1: $V \leftarrow$ WD-VarScore($\overline{\mathbb{X}}$, $\overline{\mathbb{Y}}$, $\{s_m/s_s, g_o, g_e\}$) ▷ by (5.5)-(5.15)
2: $P \leftarrow (-1, 0, \ldots, 0, |\mathsf{R}|+1, 0, \ldots, 0)$ ▷ Set $|\mathsf{R}|+1$ at the slot $2|\mathbb{X}|+1$
3: $S \leftarrow \mathsf{RightRotate}(\mathsf{Add}(\overline{\mathbb{X}}.\mathsf{pos}, \bar{l}_{ref}), 1)$
4: $E \leftarrow \mathsf{Add}(\mathsf{LeftRotate}(\overline{\mathbb{X}}.\mathsf{pos}, 1), P)$
5: $M \leftarrow \mathsf{MultiplyConst}(\mathsf{Subtract}(E, S), s_m)$
6: **return** V, M

regions at its odd slots, as shown in Figure 5.6. Algorithm WD-VarScore computes the total scores of variant regions according to (5.5)-(5.15), and it can be implemented simply using the operations in Section 2.5. We compute the total scores of matched regions according to (5.16)-(5.18) as in lines 2-5. In line 3, $\bar{l}_{ref}$ denotes the encrypted l_{ref} computed in WD-VarScore according to (5.7).

#slot	1	2	3	4	5	6	7	8	9	10	11	12	...	32
V	0	-18	0	-10	0	1	0	-11	0	-18	0	-	...	-
$\bar{l}_{ref}$	0	4	0	2	0	4	0	0	0	5	0	-	...	-
P	-1	0	0	0	0	0	0	0	0	0	28	-	...	-
S	0	0	7	0	12	0	17	0	18	0	27	-	...	-
E	2	0	10	0	13	0	18	0	22	0	28	-	...	-
M	10	0	15	0	5	0	5	0	20	0	5	-	...	-

Figure 5.6: Running example of Algorithm 7

For example, Figure 5.6 shows the running example of WD-Score which takes the encrypted variants in $\mathbb{X}$ and $\mathbb{Y}$ in Figure 2.2, and the scoring scheme in Figure 2.1 as input. In the algorithm, slots from $2|\mathbb{X}|+2$ to n of the vectors

Algorithm 8 WD-FIND(V, M)

1: $mask \leftarrow (1, \ldots, 1, 0, \ldots, 0)$ ▷ Set 1 at the first $2|\mathbb{X}| + 2$ slots
2: $minf \leftarrow (0, \ldots, 0, -1, \ldots, -1)$ ▷ Set -1 at the slots from $2|\mathbb{X}| + 3$ to n
3: $mv \leftarrow \mathsf{MultiplyConst}(\mathsf{Add}(V, M), mask)$
4: $psum \leftarrow \mathsf{RightRotate}(mv, 1)$
5: **for** $i \leftarrow 1$ **to** $\log n - 1$ **do**
6: $\quad t \leftarrow \mathsf{RightRotate}(psum, 2^{i-1})$
7: $\quad psum \leftarrow \mathsf{Add}(psum, t)$
8: $smax \leftarrow \mathsf{Add}(\mathsf{MultiplyConst}(psum, mask), minf)$
9: **for** $i \leftarrow 1$ **to** $\log n - 1$ **do**
10: $\quad t \leftarrow \mathsf{LeftRotate}(smax, 2^{i-1})$
11: $\quad \_, smax \leftarrow \mathsf{MinMax}(smax, t, d)$
12: $sub \leftarrow \mathsf{MultiplyConst}(\mathsf{Subtract}(smax, psum), mask)$
13: **for** $i \leftarrow 1$ **to** $\log n - 1$ **do**
14: $\quad t \leftarrow \mathsf{LeftRotate}(sub, 2^{i-1})$
15: $\quad \_, sub \leftarrow \mathsf{MinMax}(sub, t, d)$
16: **return** sub

are not used, i.e., from 12 to 32 in the figure (denoted by dashes). They will be used in the next step, i.e., in WD-FIND.

5.4.3 Finding Score of HSR

Since Kadane's algorithm [7] is sequential, it cannot be implemented efficiently using the HEAAN scheme in which the operations are performed on the slots of vectors in parallel. Therefore, we derive Algorithm 8 based on a parallel algorithm for the maximum subsequence sum in [61]. It takes the output of WD-SCORE as input, and outputs the encrypted score of a HSR (i.e., sum of

scores in a contiguous subarray with the largest sum from the total scores in V and M). At the end of the algorithm, the score will be stored at the first slot of a vector *sub* in Algorithm 8. Unlike the bit-wise algorithm, here we find the score without starting and ending positions because keeping track the location of a HSR using HEAAN requires a large amount of homomorphic computations.

#slot	1	2	3	4	5	6	7	8	9	10	11	12	13	...	32
mv	10	-18	15	-10	5	1	5	-11	20	-18	5	-	0	...	0
psum	0	10	-8	7	-3	2	3	8	-3	17	-1	4	-	...	-
smax	17	17	17	17	17	17	17	17	17	17	4	4	-	...	-
sub	25	25	25	20	20	20	20	20	20	5	5	0	-	...	-

Figure 5.7: Running example of Algorithm 8

For example, Figure 5.7 shows the running example of WD-FIND which takes the total scores in Figure 5.6 as its input. The score of a HSR which we are looking for, 25, is stored at the first slot of vector *sub*.

In Algorithms 7 and 8, all the outputs are encrypted, so they do not leak any information. Note that w_{alt}, and an evaluation key ek are omitted from the input of the algorithms and operations for simplicity.

5.4.4 Complexity Analysis

We analyze the number of homomorphic operations used in Algorithms 7 and 8. Here, we assume that the size n of a plaintext vector is always not greater than $N/2$, the total number of slots in HEAAN (N is a ring dimension), i.e., integers in each column `pos`, `reflen`, and `altlen` can fit into a single plaintext vector. Table 5.2 shows the number of operations used in Algorithm 7. Similarly to Table 5.1, we show the number of operations for each equation in WD-

Table 5.2: Number of homomorphic operations in Algorithm 7 ($\beta = \lceil w_{alt}/6 \rceil$)

Operation	Line 1 (WD-VarScore)											Line 3	Line 4	Line 5	Total
	(5.5)	(5.6)	(5.7)	(5.8)	(5.9)	(5.10)	(5.11)	(5.12)	(5.13)	(5.14)	(5.15)				
Add	1	-	-	-	-	3	1	1	1	3	-	1	1	-	11
Equals	-	-	-	-	-	$\beta+1$	-	-	-	-	-	-	-	-	$\beta+1$
EqualsZero	-	-	-	-	-	1	1	-	-	2	-	-	-	-	3
MinMax	-	-	1	1	-	-	-	-	-	-	1	-	-	-	3
Multiply	-	-	-	-	-	$\beta+3$	1	-	-	2	-	-	-	-	$\beta+5$
MultiplyConst	-	-	3	3	-	3	1	-	-	2	3	-	-	1	15
LeftRotate	-	-	-	-	-	-	-	-	-	-	-	-	1	-	1
RightRotate	-	-	-	-	-	-	-	-	-	-	-	1	-	-	1
Subtract	-	1	-	2	-	3	2	-	-	4	-	-	-	1	11

VarScore in separate columns with the equation numbers in Table 5.2. In columns (5.7), (5.8), and (5.15), we need to scale down the two inputs of MinMax into the interval $[-1, 1]$ by using MultiplyConst, and to scale up its result by using MultiplyConst again, and therefore, three MultiplyConst are used in each column. In column (5.9), we do not need any operation because the minimum is already found in column (5.8). In column (5.11), an affine gap penalty function $W(l)$ can be computed as follows, using Add, EqualsZero, Multiply, MultiplyConst, and Subtract.

$$W(l) = (1 - \mathsf{EqualsZero}(l)) \times \big((l-1) \times g_e + go\big) \tag{5.24}$$

In column (5.10), since $x.\texttt{alt}$ and $y.\texttt{alt}$ are divided into $\lceil w_{alt}/6 \rceil$ vectors, we need the same number of Equals to check $x.\texttt{alt} = y.\texttt{alt}$. Moreover, we need one more Equals to check $x.\texttt{altlen} = y.\texttt{altlen}$. To merge these results of Equals into one variable, we need to multiply all of them. Let us denote by e the result of the multiplications. Then v' in (5.10) can be computed as follows, using Add,

Multiply, MultiplyConst, and Subtract.

$$v' = W(l_g) + \mathsf{e} \times (l_m \times s_m) + (1 - \mathsf{e}) \times (l_m \times s_s) \tag{5.25}$$

Since the number of the circuits in column (11) (the gray colored number) is included in columns (10) and (14), it is excluded from the total number in the last column. In the table, the total number of homomorphic operations in Algorithm 7 does not depend on $|\mathbb{X}|$, i.e., WD-Score uses $O(1)$ homomorphic operations since w_{alt} is a fixed length that can be selected empirically.

Now we analyze the number of operations in Algorithm 8. We can count them directly from the algorithm, i.e., we use one Subtract, 3 MultiplyConst, $\log n$ RigthRotate, $\log n + 1$ Add, $2\log n - 2$ LeftRotate, and $2\log n - 2$ MinMax. In addition, we use four more MultiplyConst to scale up and down $smax$ and sub right before and after the last two loops (for MinMax), in the same way as in columns (5.7),(5.8), and (5.15). Overall, WD-Find uses $O(\log |\mathbb{X}|)$ homomorphic operations since $n = 2^{\lceil \log(2|\mathbb{X}|+1) \rceil + 1}$.

Since HEAAN is a leveled HE scheme, we also consider an important factor, the depth of homomorphic multiplication of the two algorithms. The depth of operation Multiply in the HEAAN scheme is regarded as 1, and the other built-in operations are 0. For the additional operations, we denote the depth of MinMax by d_M and Equals/EqualsZero by d_E in the following. Table 5.3 shows the depths of variables used in Algorithm 7. We show the depth of each variable in WD-VarScore with the equation number in the table. The largest one in the table is the depth of V, and thus $\max(d_M, \beta) + d_M + d_E + 1$ will be the depth of Algorithm 7, i.e., the multiplicative depth of WD-Score is $O(d_M + d_E + w_{alt})$. The depth of Algorithm 8 is equal to $2 \times d_M \times (\log n - 1) = 2d_M \lceil \log(2|\mathbb{X}| + 1) \rceil$ since it uses only MinMax in the two loops. In other words, the multiplicative depth of WD-Find is $O(d_M \log |\mathbb{X}|)$. Therefore, the total depth of our word-

Table 5.3: Depth of homomorphic multiplication in Algorithm 7 ($\beta = \lceil w_{alt}/6 \rceil$)

Variable	Equation/Line	Depth
l_{ref}	(5.7)	d_M
l_m	(5.5)	d_M
l_g	(5.6)	d_M
W	(5.11)	$d_E + 1$
$x.\texttt{alt} = y.\texttt{alt}$ (i.e., e in (5.25))	(5.10)	$d_E + \beta$
v'	(5.10)	$\max(d_M, d_E + \beta) + 1$
v''	(5.14)	$d_E + d_M + 1$
v (i.e., V in Line 2)	(5.15)	$\max \begin{cases} \max(d_M, d_E + \beta) + 1 \\ d_E + d_M + 1 \end{cases} + d_M$
S	Line 3	d_M
E	Line 4	0
M	Line 5	d_M

wise implementation is $2d_M\lceil \log(2|\mathbb{X}| + 1) \rceil + \max(d_M, \lceil w_{alt}/6 \rceil) + d_M + d_E + 1$ since WD-Find takes the output of WD-Score as input.

5.5 Performance Evaluation

We present experimental results to show the performances of our algorithms. In the experimental results, BT stands for the bit-wise algorithm, i.e., the combination of BT-Score and BT-Find, and WD stands for the word-wise algorithm, i.e., the combination of WD-Score and WD-Find. Both BT and WD are implemented in C++. We use OpenMP for the parallel computations in

BT. Experiments are conducted on a machine with two Intel Xeon Silver 4114 2.20GHz CPUs (32 cores) and 256GB memory running Debian Linux.

Datasets. We use a synthetic dataset SYN and two real datasets PGP and

Table 5.4: Characteristics of datasets

Dataset	$\lvert\mathbb{X}\rvert, \lvert\mathbb{Y}\rvert$	$\lvert\mathsf{R}\rvert$	Distribution of variants				
			SNP	Insertion	Deletion	Substitution	Empty
SYN	1.6K	10K	52%	5%	12%	4%	26%
PGP	28K	14M	57%	5%	4%	4%	29%
IDASH	14K	249M	48%	10%	11%	1%	29%

IDASH in the experiments. To generate SYN, we select a genomic sequence from NCBI database as the reference genome R and use its first 10K bases (i.e., $|\mathsf{R}| = 10\text{K}$). Then we generate 1.6K variants in $\mathbb{X}$ and $\mathbb{Y}$ along R randomly with the similar distributions of variants as in PGP and IDASH. PGP consists of the complete genomics data of two participants of Personal Genome Project. More specifically, PGP stores the variants in Chromosome 1 of two participants hu0D879F and hu9385BA. IDASH is a dataset for a task given at iDASH Privacy & Security analysis challenge, which is used in [48]. The characteristics of the datasets are summarized in Table 5.4. In the table, we show the percentage of SNP in each dataset separately from substitution. Note that each dataset consists of two lists of variants $\mathbb{X}$ and $\mathbb{Y}$ ($|\mathbb{X}| = |\mathbb{Y}|$) as shown in Figure 2.2.

Parameters of HE schemes. The security of TFHE relies on the hardness of the TLWE assumption which is a generalization of the Learning with Errors (LWE) problem over the torus [27]. In TFHE, we use the security parameter $\lambda = 128$ proposed in [29] according to an LWE estimator in [1]. The security of HEAAN relies on the hardness of the RLWE assumption [24]. In HEAAN, we

Table 5.5: Parameters of HE schemes

Parameter	SYN	PGP	IDASH
TFHE			
w_{int}	11-18	20-24	25-32
w_{alt}	24	24	24
HEAAN			
N	2^{16}	2^{16}	2^{16}
n	2^5-2^{11}	2^5-2^{11}	2^5-2^{11}
l	71-107	65-118	112-148
q	29-32	32-39	45-52
u	7-13	15-16	17-23
w_{alt}	24	24	24

use the security parameter $\lambda \geq 128$ determined by the LWE estimator [1], based on a ring dimension N, a level parameter l, and a number of quantization bits q. Since HEAAN outputs an approximate result, we select l, q, and a number of iteration u empirically so that the computation error is extremely small. A bit length of two's complement representation w_{int} for the homomorphic circuits used in BT is computed according to (5.19) for each dataset. We fix the length of alternate bases to $w_{alt} = 24$ bits in both BT and WD. The selected parameters are given in Table 5.5.

Specifications of WD. Table 5.6 shows the multiplicative depth, the number of bootstrapping performed, and the size of unit ciphertext of WD in the experiment. Since the depth of MinMax $d_M = 5u$ (u is the number of iterations) and the depth of Equals/EqualsZero $d_E = 18$, the depth of WD reaches 2,779 in the experiment. However, thanks to the automatically performed bootstrapping in

Table 5.6: Specifications of WD

	SYN	PGP	IDASH
Multiplicative depth	439-1,579	919-1,939	1,039-2,779
Number of bootstrapping	11-23	11-23	11-24
Size of unit ciphertext (MB)	32-54	33-72	79-121

MinMax, Equals, and EqualsZero of the HEAAN scheme, we use much smaller depths for l instead of the actual multiplicative depths in the table. Since bootstrapping is the slowest operation of the word-wise HE schemes and the most time consuming part of our computation, we use the HEAAN scheme rather than the other schemes such as BGV[14] and BFV [36] because of its efficient bootstrapping.

5.5.1 Sensitivity Analysis

The performances of BT and WD are evaluated in several aspects: (1) varying the number of variants, (2) varying the length of reference genome, (3) varying the length of alternate bases, (4) varying the scoring scheme, (5) varying the genome complexity, and (6) varying the number of cores. Table 5.7 shows the parameters of the experiments. Values in boldface in the table are used as default parameters. Since the elapsed times and memory usages of the encryption, decryption, and key generation of TFHE and HEAAN are negligible compared to the evaluation, we report the performances of the evaluation only, i.e., BT and WD here. To report the average elapsed times and memory usages of BT and WD, we conduct the experiment three times and take their average.

Varying Number of Variants. In this experiment, we measure the average elapsed times and the average peak memory usages of BT and WD in the dif-

Table 5.7: Experiment settings

Parameter	Value
Number of variants $\|\mathbb{X}\|$	10, 20, 50, 100, 200, **500**, 1000
Length of reference genome $\|\mathsf{R}\|$	**3K**, 10K, 100K, 1M
Length of alternate bases k (i.e., $k = w_{alt}/2$)	1, 3, 6, **12**, 24, 48
Scoring scheme	{3/-2,-11,-1}, **{5/-3,-9,-1}**, {5/-4,-10,-4}
Genome complexity	0.1, 0.5, **0.9**
Number of cores	1, 2, 4, 8, 16, **32**

ferent numbers of variants for SYN, PGP, and IDASH. Since a homomorphic computation generally takes a long time for larger instances, we set a 4-hour ($\approx 14,000s$) time limit. If some experiment reaches the limit, we record its elapsed time as 4 hours. To vary $|\mathbb{X}|$, we use the first 10 to 1,000 variants in each dataset. Figure 5.8 shows the result of the experiment. In Figures 5.8a and 5.8b, BT outperforms WD when $|\mathbb{X}| \leq 500$ for SYN and PGP. In particular, BT takes 2 minutes when $|\mathbb{X}| = 10$ ($|\mathsf{X}|, |\mathsf{Y}| \approx 30,000$ bps) for PGP. In Figure 5.8c, even though BT outperforms WD in all $|\mathbb{X}|$ for IDASH, the gap between their elapsed times reduces significantly as $|\mathbb{X}|$ gets larger. The reason is that WD uses $O(\log |\mathbb{X}|)$ homomorphic operations, whereas BT uses $O(|\mathbb{X}| w_{int} \log w_{int})$ homomorphic gates. These growths of complexities can be observed in the three figures. Generally, WD computes the total scores of each region efficiently while BT effectively finds a HSR. Furthermore, the average elapsed time of WD-Score gradually increases even though it uses constant number of homomorphic operations. The reason is that the bigger parameters for HEAAN in the larger $|\mathbb{X}|$ make the operations slower. In Figures 5.8d, 5.8e,

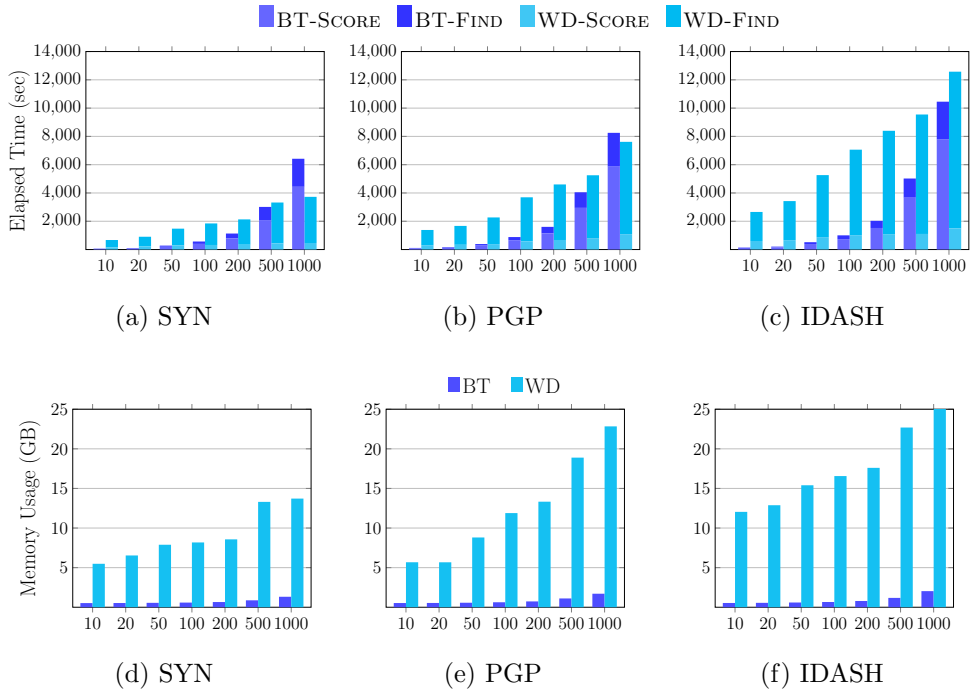
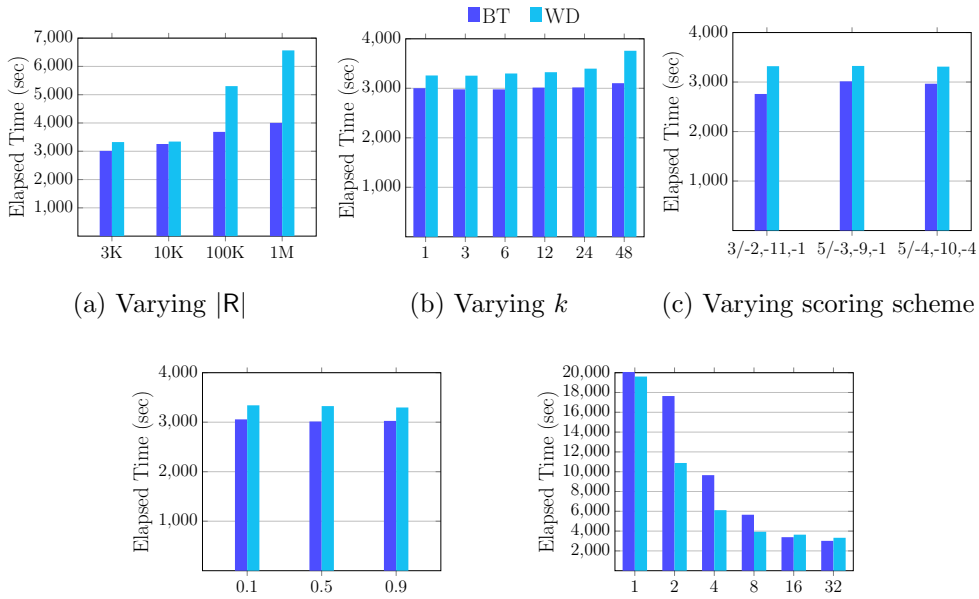


Figure 5.8: Varying $|\mathbb{X}|$

and 5.8f, WD uses memory 10-25 times more than BT because of the size of unit ciphertext. The size of unit ciphertext is a few KB in TFHE while it takes several MB in HEAAN (see Table 5.6). Therefore, there is a time memory trade-off between BT and WD. In general, both BT and WD run slower for IDASH than SYN and PGP. This is because the much longer R of IDASH requires the bigger parameters for HE schemes that lead to the poor performance. Note that the parameters of HE schemes used in the experiment are summarized in Table 5.5. Since HEAAN is an approximate HE scheme, the two scores of a HSR computed by BT and WD for the same input can be slightly different. In the experiment, the differences between them (i.e., the errors in WD) are less than 0.02% of the actual scores for the three datasets.

Varying Length of Reference Genome. We measure the average elapsed



(a) Varying |R| (b) Varying k (c) Varying scoring scheme

(d) Varying genome complexity (e) Varying number of cores

Figure 5.9: Varying parameters

times of BT and WD in the different lengths of R for SYN. We fix $|\mathbb{X}| = 500$ and vary |R| from 3K to 1M. To vary |R|, we simply increase the positions in column `pos` in SYN. Figure 5.9a shows the result of the experiment. When |R| increases, the elapsed times of BT and WD are increased in the figure. The reason is that the larger positions require bigger parameters for HE schemes what cause the slower elapsed times.

Varying Length of Alternate Bases. We measure the average elapsed time in the different lengths of alternate bases for SYN. To vary k, we change parameter w_{alt} from 2 to 96 in both BT and WD since $k = w_{alt}/2$. Figure 5.9b shows the result of the experiment. The average elapsed time of BT remains the same in every k since w_{alt} has almost no effect on the span and work of BT-Score (i.e., in Equation (5.20)). However, even though w_{alt} slightly affects

the numbers of equality check and multiplication operations in WD-Score (see Table 5.2), the average elapsed time of WD increases in the larger k due to the costly equality check operation of HEAAN.

Varying Scoring Scheme. We measure the average elapsed time in the different scoring schemes for SYN. We select three widely used scoring schemes for nucleotide sequences in FASTA [57] and [44] (see Table 5.7). Figure 5.9c shows the result of the experiment. The average elapsed time of BT in the first scoring scheme is a little bit shorter than the other two because of the smaller w_{int} on which the spans and works of BT-Score and BT-Find highly depend. In other words, w_{int} for the match score $s_m = 3$ is shorter than for $s_m = 5$ according to (5.19). The average elapsed time of WD remains the same in all the scoring schemes because the difference between match scores 3 and 5 is too small to affect the parameters of HEAAN.

Varying Genome Complexity. We measure the average elapsed time in the three different genome complexities. For the genome complexity, we use a length-sensitivity measure D_k [62] which is a rate of distinct substrings of length k in R, defined as follows.

$$D_k(\mathsf{R}) = \frac{|\{s : f(s) > 0.\ |s| = k\}|}{|\mathsf{R}| - k + 1}$$

where $f(s)$ is the number of occurrences of a substring s in R. We use D_{12} in the experiment. To vary the genome complexity, we use two other synthetic datasets generated similarly to SYN. To generate them, we select two genomic sequences with the genome complexities 0.1 and 0.5 from NCBI database as the reference genomes, then generate the variants for the two datasets in the same way as we do for SYN. Note that the genome complexity of R of SYN is 0.9. Figure 5.9d shows the result of the experiment. In the figure, we can observe that the different genome complexities have no effect on the average elapsed

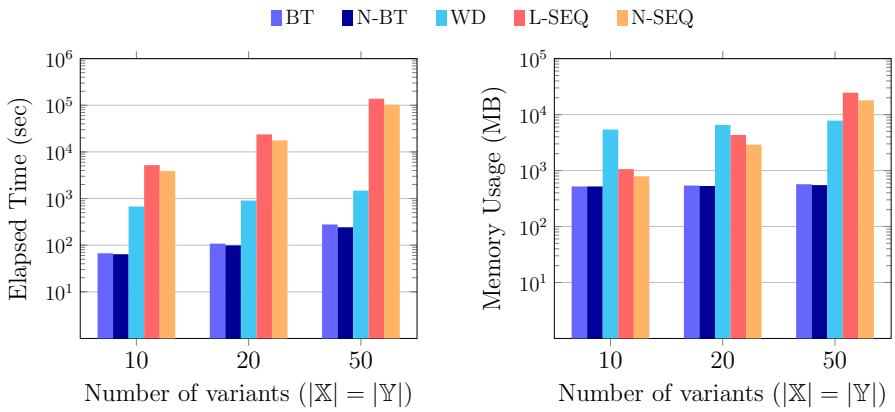


Figure 5.10: Comparing with L-SEQ and N-SEQ

times of BT and WD.

Varying Number of Cores. We measure the average elapsed time in the different numbers of cores for SYN. Figure 5.9e shows the result of the experiment. In general, the average elapsed times decrease as the number of cores increases since the algorithms in both BT and WD are designed to take advantage of parallelism. In particular, the speedups of BT and WD are 9.7 and 5.3 in 16 cores, respectively. Furthermore, WD is 1.4-2 times faster than BT when the number of cores is less than 16. In 16 and 32 cores, their average elapsed times are almost equal. To measure the speedups, we do not set a time limit in the experiment. Note that we use the sequential circuits in BT-Find when running BT on a single core.

5.5.2 Comparing with Optimal HSR Algorithm

We compare the average elapsed times and peak memory usages of BT and WD with L-SEQ and N-SEQ (Chapter 4) in the different numbers of variants for SYN. Similarly to N-SEQ, N-BT is a version of BT that finds only the

score without the positions, i.e., the computations related to lists S and E are excluded from BT-Score and BT-Find. Since L-SEQ and N-SEQ take two sequences as inputs, not lists of variants, we use the corresponding sequences of the variants in SYN to measure their performances. Figure 5.10 shows the result of the experiment. Since L-SEQ and N-SEQ take large amounts of time and memory, we use the logarithmic scale to display the elapsed time and memory usage in the figures. Furthermore, the number of variants is limited to 50 due to the huge elapsed times of L-SEQ and N-SEQ. In this experiment, BT, N-BT, and WD outperform L-SEQ and N-SEQ in the average running times by up to 2 orders of magnitude while consuming less memories.

5.5.3 Quality of HSR

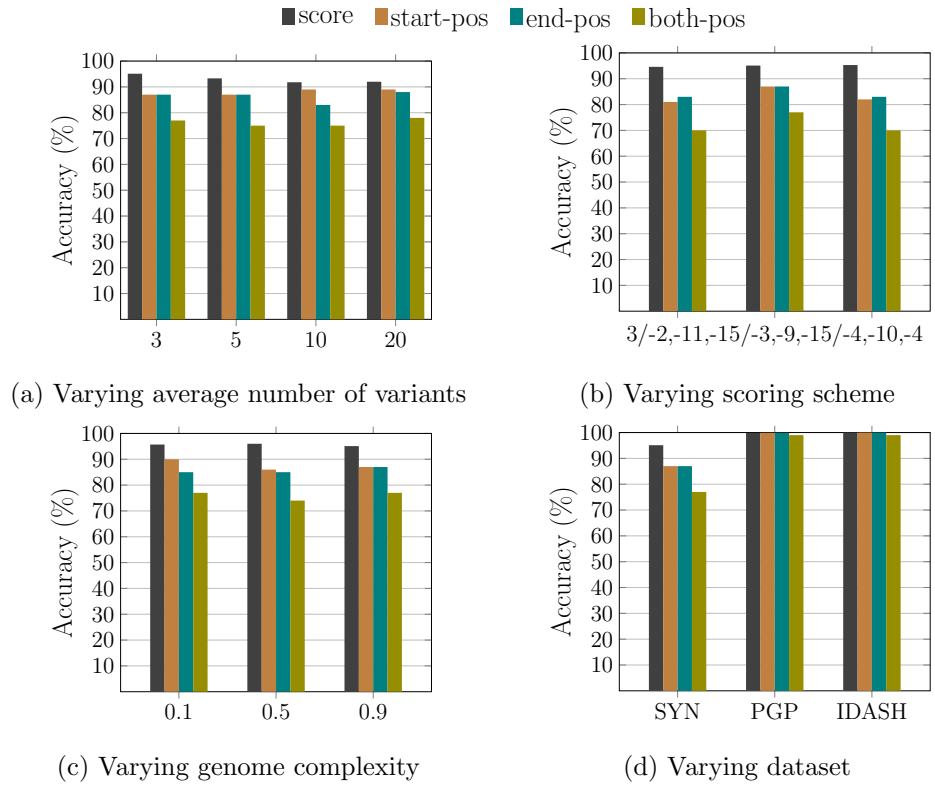
Since our algorithm finds HSR approximately to SW, it is important to take into account the quality of the obtained region, i.e., how closely our algorithm finds the HSR. To express the quality, we measure the score and location accuracies of our result by comparing with the HSR obtained by SW. To measure the accuracies in the experiment, we split variants in a dataset into subsets by their positions so that subsequences of the reference genome for the subsets have equal length r (we call r subreference length). If there is a variant region that lies in two subsequences (i.e., a variant region is not fully contained in one subsequence), that pair of variants is excluded from the subset. Then for each subset, we find the score, starting position, and ending position by our algorithm and the score, two starting positions, and two ending positions of the HSR by SW. To compute the score accuracy, we calculate the Mean Absolute Percentage Error (MAPE) between the scores obtained by our algorithm and SW. To compute the location accuracy, we calculate the percentage of the starting (ending) positions that are "correct". To decide whether a starting (ending)

Table 5.8: Experiment parameters

Parameter	Value
Average number of variants	**3**, 5, 10, 20
Scoring scheme	{3/-2, -11, -1}, **{5/-3, -9, -1}**, {5/-4, -10, -4}
Genome complexity	0.1, 0.5, **0.9**
Dataset	**SYN**, PGP, IDASH

position is "correct", we check that the starting position of our result and the two starting (ending) positions obtained by SW lie in exactly the same matched region. If these three starting (ending) positions lie in the same matched region, it is regarded to be "correct". If a starting (ending) position obtained by SW lies inside a variant region, we assume that it lies in its next (previous) matched region. The reason we use this metric is that SW retrieves the location of the regions letter by letter, while our algorithm retrieves it variant by variant and a starting (ending) position of the our result always lies in a matched region.

Furthermore, the accuracies are measured in several aspects: (1) varying the average number of variants in subsets, (2) varying the scoring scheme, (3) varying the genome complexity, and (4) varying the dataset. Table 5.8 shows the parameters of the experiments. Values in boldface in Table 5.8 are used as default parameters. Figure 5.11 shows the result of the experiments. To vary the average number of variants in subsets, we select the subreference length r between 20 and 130 for SYN in Figure 5.11a. In the figure, the score and position accuracies remain the same in the different average numbers of variants. Figure 5.11b shows the result of the experiment in the three different scoring schemes for SYN. In the figure, the location accuracies in the default scoring scheme are slightly better than the other two. To vary the genome complexity, we use



(a) Varying average number of variants

(b) Varying scoring scheme

(c) Varying genome complexity

(d) Varying dataset

Figure 5.11: Quality of HSR

the other two datasets used in Figure 5.9d. We can conclude that the genome complexity does not affect the accuracies in Figure 5.11c. In Figures 5.11a, 5.11b and 5.11c, the score accuracy is 91-96%, the starting and ending position accuracies are 81-90%. When we consider the case that both starting and ending positions are correct, its accuracy is 70-78% in the figures. To fix the average number of variants to 3 (default value) in PGP, we select $r = 1000$, much longer than the lengths used in SYN. Moreover, when we select $r = 1000$ for IDASH, most subsets are empty because of the extremely long matched regions between the variants. Therefore, we use the non-empty subsets in the experiment for IDASH. In Figure 5.11d, the score and position accuracies are more than 99%

for PGP and IDASH because of the long matched regions between the variants that lead to the trivial alignment (i.e., the starting and ending positions are 1 and $|\mathsf{R}|$, respectively).

Chapter 6

Conclusion

We propose an algorithm that finds HSR of two homomorphically encrypted genomic sequences, based on the Smith-Waterman recurrence. With the efficient location retrieval, parallel computations, and a proper HE scheme, it shows good performances in the experiment so as to be useful in practice.

We also propose an algorithm that finds HSR of two genomic sequences represented by homomorphically encrypted variants and describe how to implement it using the bit-wise and word-wise HE schemes. Moreover, we conduct extensive experiments and parameter sensitivity analysis on real and synthetic datasets to show its performance. The experiments show that it outperforms the previous algorithm by up to 2 orders of magnitude in terms of elapsed time, while the score accuracy of HSR is over 91% compared with the Smith-Waterman algorithm. In particular, it takes 2 minutes to find the score and location of HSR of two homomorphically encrypted sequences with length 30,000 bps each in a real dataset. Overall, it obtains HSR of the sequences in a feasible time.

Bibliography

[1] M. R. Albrecht, R. Player, and S. Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015.

[2] S. F. Altschul and B. W. Erickson. Optimal sequence alignment using affine gap costs. *Bulletin of Mathematical Biology*, 48:603–616, 1986.

[3] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[4] M. Aziz, M. N. Sadat, D. Alhadidi, S. Wang, X. Jiang, C. L. Brown, and N. Mohammed. Privacy-preserving techniques of genomic data-a survey. *Briefings in Bioinformatics*, 20(3):887–895, 2019.

[5] M. Bataa, S. Song, K. Park, M. Kim, J. H. Cheon, and S. Kim. Homomorphic computation of local alignment. In *IEEE International Conference on Bioinformatics and Biomedicine*, pages 2167–2174, 2020.

[6] M. Bataa, S. Song, K. Park, M. Kim, J. H. Cheon, and S. Kim. Finding highly similar regions of genomic sequences through homomorphic encryption. *Submitted*, 2022.

[7] J. Bentley. Programming pearls: Algorithm design techniques. *Communications of the ACM*, 27(9):865–873, 1984.

[8] M. Blatt, A. Gusev, Y. Polyakov, and S. Goldwasser. Secure large-scale genome-wide association studies using homomorphic encryption. *Proceedings of the National Academy of Sciences*, 117(21):11608–11613, 2020.

[9] M. Blatt, A. Gusev, Y. Polyakov, K. Rohloff, and V. Vaikuntanathan. Optimized homomorphic encryption solution for secure genome-wide association studies. *BMC Medical Genomics*, 13(7):1–13, 2020.

[10] F. Boemer, A. Costache, R. Cammarota, and C. Wierzynski. Ngraph-he2: A high-throughput framework for neural network inference on encrypted data. In *Proceedings of the 7th ACM Workshop on Encrypted Computing and Applied Homomorphic Cryptography*, pages 45–56, 2019.

[11] F. Boemer, Y. Lao, R. Cammarota, and C. Wierzynski. Ngraph-he: A graph compiler for deep learning on homomorphically encrypted data. In *Proceedings of the 16th ACM International Conference on Computing Frontiers*, pages 3–13, 2019.

[12] D. Bogdanov, L. Kamm, S. Laur, and V. Sokk. Implementation and evaluation of an algorithm for cryptographically private principal component analysis on genomic data. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 15(5):1427–1432, 2018.

[13] C. Bonte and F. Vercauteren. Privacy-preserving logistic regression training. *BMC Medical Genomics*, 11(4):13–21, 2018.

[14] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (Leveled) Fully Homomorphic Encryption without Bootstrapping. In *Innovations in Theoretical Computer Science Conference*, pages 309–325, 2012.

[15] R. P. Brent and H. T. Kung. A regular layout for parallel adders. *IEEE Transactions on Computers*, C-31:260–264, 1982.

[16] D. L. Brutlag, J.-P. Dautricourt, S. Maulik, and J. Relnh. Improved sensitivity of biological sequence database searches. *Bioinformatics*, 6(3):237–245, 1990.

[17] A. Brutzkus, R. Gilad-Bachrach, and O. Elisha. Low latency privacy preserving inference. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 812–821, 2019.

[18] S. Carpov, N. Gama, M. Georgieva, and J. R. Troncoso-Pastoriza. Privacy-preserving semi-parallel logistic regression training with fully homomorphic encryption. *BMC Medical Genomics*, 13(7):1–10, 2020.

[19] G. S. Çetin, H. Chen, K. Laine, K. Lauter, P. Rindal, and Y. Xia. Private queries on encrypted genomic data. *BMC Medical Genomics*, 10(2):1–14, 2017.

[20] H. Chen, W. Dai, M. Kim, and Y. Song. Efficient multi-key homomorphic encryption with packed ciphertexts with application to oblivious neural network inference. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 395–412, 2019.

[21] H. Chen, R. Gilad-Bachrach, K. Han, Z. Huang, A. Jalali, K. Laine, and K. Lauter. Logistic regression over encrypted data from fully homomorphic encryption. *BMC Medical Genomics*, 11(4):3–12, 2018.

[22] J. Chen, M. Guo, X. Wang, and B. Liu. A comprehensive review and comparison of different computational methods for protein remote homology detection. *Briefings in Bioinformatics*, 19(2):231–244, 2016.

[23] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song. Bootstrapping for approximate homomorphic encryption. In *Advances in Cryptology–EUROCRYPT*, pages 360–384, 2018.

[24] J. H. Cheon, A. Kim, M. Kim, and Y. Song. Homomorphic encryption for arithmetic of approximate numbers. In *Advances in Cryptology–ASIACRYPT*, pages 409–437, 2017.

[25] J. H. Cheon, D. Kim, D. Kim, H. H. Lee, and K. Lee. Numerical method for comparison on homomorphically encrypted numbers. In *Advances in Cryptology–ASIACRYPT*, pages 415–445, 2019.

[26] J. H. Cheon, M. Kim, and K. Lauter. Homomorphic computation of edit distance. In *Financial Cryptography and Data Security*, pages 194–212, 2015.

[27] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *Advances in Cryptology–ASIACRYPT*, pages 3–33, 2016.

[28] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. TFHE: Fast fully homomorphic encryption library, 2016. https://tfhe.github.io/tfhe/.

[29] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. Tfhe: Fast fully homomorphic encryption over the torus. *Journal Of Cryptology*, 33(1):34–91, 2020.

[30] H. Cho, D. J. Wu, and B. Berger. Secure genome-wide association analysis using multiparty computation. *Nature Biotechnology*, 36:547–551, 2018.

[31] S. D. Constable, Y. Tang, S. Wang, X. Jiang, and S. Chapin. Privacy-preserving gwas analysis on federated genomic datasets. *BMC Medical Informatics and Decision Making*, 15(5):1–9, 2015.

[32] P. Danecek, A. Auton, G. Abecasis, C. Albers, E. Banks, M. DePristo, R. Handsaker, G. Lunter, G. Marth, S. Sherry, G. McVean, R. Durbin, and 1000 Genomes Project Analysis Group. The variant call format and vcftools. *Bioinformatics*, 27(15):2156–2158, 2011.

[33] M. De Cock, R. Dowsley, A. C. Nascimento, D. Railsback, J. Shen, and A. Todoki. High performance logistic regression for privacy-preserving genome analysis. *BMC Medical Genomics*, 14(23):1–18, 2021.

[34] N. Dowlin, R. Gilad-Bachrach, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, pages 201–210, 2016.

[35] L. Ducas and D. Micciancio. Fhew: Bootstrapping homomorphic encryption in less than a second. In *Advances in Cryptology–EUROCRYPT*, pages 617–640, 2015.

[36] J. Fan and F. Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Paper 2012/144, 2012.

[37] R. Fuchs and P. Stoehr. EMBL–Search: a CD–ROM based database query system. *Bioinformatics*, 9(1):71–77, 1993.

[38] C. Gentry. Fully Homomorphic Encryption Using Ideal Lattices. In *Annual ACM Symposium on Theory of Computing*, pages 169–178, 2009.

[39] O. Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162(3):705–708, 1982.

[40] T. Han and D. A. Carlson. Fast area-efficient vlsi adders. In *1987 IEEE 8th Symposium on Computer Arithmetic (ARITH)*, pages 49–56, 1987.

[41] N. Homer, B. Merriman, and S. F. Nelson. Bfast: An alignment tool for large scale genome resequencing. *PLoS ONE*, 4(11):e7767, 2009.

[42] K. A. Jagadeesh, D. J. Wu, J. A. Birgmeier, D. Boneh, and G. Bejerano. Deriving genomic diagnoses without revealing patient genomes. *Science*, 357(6352):692–695, 2017.

[43] X. Jiang, M. Kim, K. Lauter, and Y. Song. Secure outsourced matrix computation and application to neural networks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1209–1222, 2018.

[44] A. Khajeh-Saeed, S. Poole, and J. Blair Perot. Acceleration of the smith–waterman algorithm using single and multiple graphics processors. *Journal of Computational Physics*, 229(11):4247–4258, 2010.

[45] A. Kim, Y. Song, M. Kim, K. Lee, and J. H. Cheon. Logistic regression model training based on the approximate homomorphic encryption. *BMC Medical Genomics*, 11(4):23–31, 2018.

[46] D. Kim, Y. Son, D. Kim, A. Kim, S. Hong, and J. H. Cheon. Privacy-preserving approximate gwas computation based on homomorphic encryption. *BMC Medical Genomics*, 13(7):1–12, 2020.

[47] M. Kim, X. Jiang, K. Lauter, E. Ismayilzada, and S. Shams. Secure human action recognition by encrypted neural network inference. *Nature Communications*, 13(4799), 2022.

[48] M. Kim and K. Lauter. Private genome analysis through homomorphic encryption. *BMC Medical Informatics and Decision Making*, 15(5):1–12, 2015.

[49] M. Kim, Y. Song, and J. H. Cheon. Secure searching of biomarkers through hybrid homomorphic encryption scheme. *BMC Medical Genomics*, 10(2):69–76, 2017.

[50] M. Kim, Y. Song, B. Li, and D. Micciancio. Semi-parallel logistic regression for gwas on encrypted data. *BMC Medical Genomics*, 13(7):1–13, 2020.

[51] M. Kim, Y. Song, S. Wang, Y. Xia, and X. Jiang. Secure Logistic Regression Based on Homomorphic Encryption: Design and Evaluation. *JMIR Medical Informatics*, 6(2):e19, 2018.

[52] B. Langmead and S. L. Salzberg. Fast gapped-read alignment with bowtie 2. *Nature Methods*, 9(4):357–359, 2012.

[53] Y. Liu, A. Wirawan, and B. Schmidt. Cudasw++ 3.0: accelerating smith-waterman protein database search by coupling cpu and gpu simd instructions. *BMC Bioinformatics*, 14(1):1–10, 2013.

[54] Q. Lou and L. Jiang. She: A fast and accurate deep neural network for encrypted data. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, 2019.

[55] D. Lu, Y. Zhang, L. Zhang, H. Wang, W. Weng, L. Li, and H. Cai. Methods of privacy-preserving genomic sequencing data alignments. *Briefings in Bioinformatics*, 22(6), 2021. bbab151.

[56] W.-J. Lu, Y. Yamada, and J. Sakuma. Privacy-preserving genome-wide association studies on cloud environment using fully homomorphic encryption. *BMC Medical Informatics and Decision Making*, 15(5):1–8, 2015.

[57] F. Madeira, Y. M. Park, J. Lee, N. Buso, T. Gur, N. Madhusoodanan, P. Basutkar, A. R. N. Tivey, S. C. Potter, R. D. Finn, and R. Lopez. The embl-ebi search and sequence analysis tools apis in 2019. *Nucleic acids research*, 47(W1):W636–W641, 2019.

[58] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.

[59] S. Park, M. Kim, S. Seo, S. Hong, K. Han, K. Lee, J. H. Cheon, and S. Kim. A secure snp panel scheme using homomorphically encrypted k-mers without snp calling on the user side. *BMC Genomics*, 20:163–174, 2019.

[60] W. R. Pearson. An introduction to sequence similarity ("homology") searching. *Current Protocols in Bioinformatics*, 42(1):3.1.1–3.1.8, 2013.

[61] K. Perumalla and N. Deo. Parallel algorithms for maximum subsequence and maximum subarray. *Parallel Processing Letters*, 5(3):367–373, 1995.

[62] V. Phan, S. Gao, Q. Tran, and N. S. Vo. How genome complexity can explain the difficulty of aligning reads to genomes. *BMC Bioinformatics*, 16(17):1–15, 2015.

[63] M. Poullet and L. Orlando. Assessing dna sequence alignment methods for characterizing ancient genomes and methylomes. *Frontiers in Ecology and Evolution*, 8, 2020.

[64] J. J. Sim, F. M. Chan, S. Chen, B. H. Meng Tan, and K. M. Mi Aung. Achieving gwas with homomorphic encryption. *BMC Medical Genomics*, 13(7):1–12, 2020.

[65] J. Sklansky. Conditional-sum addition logic. *IRE Transactions on Electronic Computers*, EC–9:226–231, 1960.

[66] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[67] J. S. Sousa, C. Lefebvre, Z. Huang, J. L. Raisaro, C. Aguilar-Melchor, M.-O. Killijian, and J.-P. Hubaux. Efficient and secure outsourcing of genomic data storage. *BMC Medical Genomics*, 10(2):15–28, 2017.

[68] J. Söding. Protein homology detection by HMM–HMM comparison. *Bioinformatics*, 21(7):951–960, 2004.

[69] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.

[70] F. Wibawa, F. O. Catak, M. Kuzlu, S. Sarp, and U. Cali. Homomorphic encryption and federated learning based privacy-preserving cnn training: Covid-19 detection use-case. In *Proceedings of the 2022 European Interdisciplinary Cybersecurity Conference*, pages 85–90, 2022.

[71] A. M. Yakubu and Y. P. Chen. Ensuring privacy and security of genomic data and functionalities. *Briefings in Bioinformatics*, 21(2):511–526, 2019.

요약

유전체 분석의 기초적인 연산 중 하나는 유전체 서열에서 높은 유사도를 가지는 부분을 찾는 것이다. 클라우드 환경에서는 대량의 유전체 데이터를 효율적으로 처리할 수 있지만, 클라우드로 외주하는 것은 개인 정보 및 보안 문제가 발생할 수 있다. 동형 암호 체계는 신뢰할 수 없는 클라우드 환경에서 처리되는 다양한 분석에서 유전체 데이터의 개인 정보를 보존하는 강력한 암호화 기법이다.

먼저 본 논문에서는 스미스-워터만 알고리즘을 기반으로 동형암호화된 두 서열 간의 유사도가 높은 부분을 찾는 효율적인 알고리즘을 제안한다. 효율적인 위치 탐색, 병렬 연산과 적절한 동형암호 구성을 갖추고 있으므로 실험에서 좋은 성능을 보여 실제로도 유용할 것이다.

다음으로는 두 동형암호화된 시퀀스 사이에서 고도로 흡사한 부분을 찾는 효율적인 알고리즘을 제시하였다. 알고리즘의 성능을 보이기 위해 실제와 합성 데이터셋에 대해 광범위한 실험을 실시했고 매개변수 응용 정도 분석을 수행하여 성능을 제시하였다. 실험에서는 실제 데이터셋에서 시퀀스와 매우 유사한 영역을 적절한 시간 안에 찾을 수 있었다.

주요어: 문자어 정렬; 동형 암호; 높은 유사도 영역; 부분 정렬; 개인 정보 보호 연산
학번: 2018-38143