Informatics in Medicine Unlocked 23 (2021) 100525

Contents lists available at ScienceDirect

Informatics in Medicine Unlocked

journal homepage: http://www.elsevier.com/locate/imu

Privacy-preserving string search on encrypted genomic data using a generalized suffix tree

Md Safiur Rahman Mahdi [a], Md Momin Al Aziz [a,*], Noman Mohammed [a], Xiaoqian Jiang [b]

[a] Department of Computer Science, University of Manitoba, Canada
[b] School of Biomedical Informatics, University of Texas Health Science Center at Houston, USA

ARTICLE INFO

Keywords:
Privacy of Genomic Data
Substring search
Generalized suffix tree
Secure substring search

ABSTRACT

Background and objective: Efficient sequencing technologies generate a plethora of genomic data and make it available to researchers. To compute a massive genomic dataset, outsourcing the data to the cloud is often required. Before outsourcing, data owners encrypt sensitive data to ensure data confidentiality. Outsourcing helps data owners to eliminate the local storage management problem. Since genome data is large in volume, executing researchers' queries securely and efficiently is challenging.
Methods: In this paper, we propose a method to securely perform substring search and set-maximal search on a SNPs dataset using a generalized suffix tree. The proposed method guarantees the following: (1) data privacy, (2) query privacy, and (3) output privacy. It adopts the semi-honest adversary model, and the security of the data is guaranteed through encryption and garbled circuits.
Results: Our experimental results demonstrate that our proposed method can compute secure substring and set-maximal searches against a single-nucleotide polymorphism (SNPs) dataset of 2184 records (each record contains 10000 SNPs) in 2.3 and 2 s, respectively. Furthermore, we compared our results with existing techniques of secure substring and set-maximal search (Ishimaki et al., 2016; Shimizu et al., 2016) [1,2], and we achieved a 400 and 2 times improvement, respectively (Table 5).

1. Introduction

The volume of genomic data is growing enormously due to the advancement of the current sequencing technologies. Storing these massive data to the cloud could be a feasible solution if we can ensure data security and privacy. The string search is a well-known and frequently used problem, and several algorithms have been already proposed by researchers. Substring search and set-maximal search on genomic data are indispensable methods in computer science. There are many real-life applications of substring search, such as ancestor search [3], similar patient query, and create personalized medicine. Besides genomic data, substring and set-maximal searches are also essential for pattern matching and word searching. They are used to search any keyword in a text message, chat logs, or documents.

Genomic data are more sensitive than many forms of data since they contain heredity information. Thus, we have to preserve the security and privacy of uploaded genomic data in the cloud. A genomic data breach of an individual may reveal their susceptibility to a particular disease, which could affect their eligibility for health insurance [4]. Recently, human genomic data were employed in forensics to identify the Golden State Killer, and examples like these show the various applications of such datasets and exacerbate their privacy implications [5].

In this paper, our primary research objective is to outsource genomic data and compute substring and set-maximal searches on the outsourced data in a privacy-preserving way. In our proposed model, we utilize a cloud to store the data because of the large volume of the genomic data. We provide *data privacy* on the outsourced cloud data. Therefore, the cloud cannot infer any information from the uploaded data. Moreover, researchers execute a query on the cloud in a way that preserves *query privacy* and *output privacy*. Thus, any adversary or the cloud itself would learn nothing about a researcher's query, and the query output is only available to the researcher.

Privacy-preserving substring and set-maximal searches are fundamental problems, and many researchers have proposed various solutions. Existing works can be divided into several sub-methods according to the number of the search sequence, length of the query, starting position, exact match, and maximum match. We compared our proposed method for substring search with Ishimaki et al. [1], where the

* Corresponding author.
E-mail address: azizmma@cs.umanitoba.ca (M.M. Al Aziz).

https://doi.org/10.1016/j.imu.2021.100525
Received 2 October 2020; Received in revised form 17 January 2021; Accepted 18 January 2021
Available online 23 January 2021

researcher's query sequence length is smaller than the dataset sequence, and the researcher is interested in the exact match from a particular position. We also compared our proposed method for set-maximal search with Shimizu et al. [2], where the researcher is interested in a maximum match between the query sequence and dataset sequence from a start position. We utilize *generalized suffix tree* a data structure that stores the suffixes of a sequence. The novel contributions of this paper are summarized as follows:

- We propose a privacy-preserving protocol to solve two problems based on genomic data: substring search query and set-maximal matches. We utilize a generalized suffix tree to create an index of the genomic data.
- We present different algorithms to execute secure substring search and secure set-maximal search on encrypted tree via garbled circuit [6]. Our proposed method preserves *data privacy*, *query privacy*, and *output privacy.*
- We implement our proposed method and evaluated the secure substring search by comparing with Ishimaki et al. [1]. Experimental results show that on a dataset of *2184* records (each containing *10000* SNPs), it requires approximately *2.3* s to execute the secure substring search, while Ishimaki et al. [1] required *16* min.
- Moreover, we compare secure set-maximal search with Shimizu et al. [2] and Sotiraki et al. [7] work. Experimental results shows that on a dataset of *2184* records (each containing *10000* SNPs), it requires approximately *2* s to execute the secure set-maximal search, while Shimizu et al. [2] required *3.97* s.

2. Preliminaries

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

2.1. System model

Our proposed model has four entities: *Hospital*, *Certified Institute* (CI), *Cloud Server* (CS), and *researchers*. The hospital entity is responsible for possessing and managing SNPs data. Though we name the entity as *hospital*, it could be any organization that possesses or collects genomic data. The second entity in our protocol is the *CI* who collects data from the *hospital* and builds the generalized suffix tree. It also encrypts the tree, sends the encrypted tree to the *CS*, and manages the keys necessary for encryption and decryption. Moreover, the *CI* sends the necessary secret key for decryption to the *researchers*. Here, we could combine these two entities (*hospital* and the *CI*) into one entity (*hospital*) by granting the *CI*'s responsibilities (building and encrypting the tree) to the *hospital*. The *CS* stores the encrypted tree and communicates with the *researcher* while executing the query. Fig. 1 summarizes our proposed system model.

2.2. Data representation

The human genome can only differ $\approx 0.1\%$ of the positions between two individuals. This location difference in genomes is called Single-Nucleotide Polymorphisms (SNPs). In this paper, we consider a dataset of n human SNP sequences with m SNPs. The SNP data were simulated using Markovian Coalescent Simulator, as collected from iDASH competition 2018 [8]. Here, SNPs are bi-allelic represented either with 0 or 1. The most frequent value is known as the major allele, while the less frequent value is called the minor allele. Major alleles are coded as 0, whereas minor alleles are coded as 1. Notably, we only have the binary representation for simplicity, as our proposed method can work on any dataset with a fixed character set. Table 1 shows an example of the dataset.

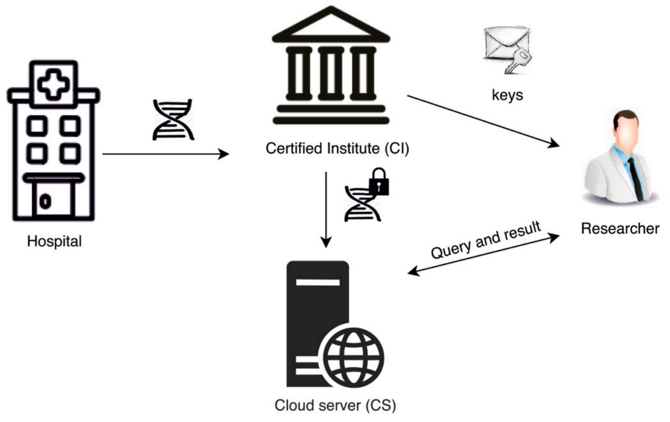


Fig. 1. Proposed system model with Hospital, Certified Institution, Cloud Server, and Researcher.

Table 1
Sample haplotype SNP data representation.

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

2.3. Query types

In this study, we present two different string queries performed against a genomic dataset. Let $\mathscr{D}$ be a dataset of genotypes ($\mathscr{D} \in \{0,1\}_m^n$), $\mathscr{D}_j = d_1, d_2, \dots, d_m$ where $\mathscr{D}^j$ represents a particular record (or sequence) in the dataset ($1 \le j \le n$). d_i represents the vector of genotypes for i^{th} genome for a vector of SNPs, where SNPs are sorted with respect to their locations on the genome. We address two different string search problems, as described below:

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

$$q = \mathscr{D}_j\left[d_s, d_{s+|q|-1}\right] \ \ where \ \ 1 \le j \le |\mathscr{D}| \tag{1}$$

Example 3.3.1. Consider researcher's submitted query, q = 010 with a start position, $s = 4$. If we execute this query q on Table 1, it should output the first sequence, d^1: 1 0 0 **0 1 0** and sequence d^2: 1 1 1 **0 1 0** since they match the query substring 010 from position 4. The details of substring search process are described in section 3.3.1.

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

1. $q[i_1, i_{max}] = \mathscr{D}_j{}'[d_s, d_s + d_{max} - 1]$

2. $q[i_{max} + 1] \neq \mathscr{D}_j{}'[d_s + d_{max}]$

3. $1 \le j' \le |\mathscr{D}|$

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

2.4. AES-Counter Mode encryption (CTR)

To encrypt the data, we utilize Counter Mode (*CTR*) of Advanced Encryption Standard (AES-CTR) encryption [9]. AES-CTR generates a keystream by encrypting *initialization vector (IV)* and consecutive values of a *counter*. The *counter* is a nonce which is never repeated in the encryption. Hence, AES-CTR produces a different cipher each time for the same plain text due to the unique counter (*padding IV and counter value*). Here, encryption function is defined as:

$$y_i = \mathscr{ENC}_k(\mathscr{IV} \mid\mid \mathscr{CTR}_i) \oplus x_i,\ i \geq 1$$

and decryption function is defined as:

$$x_i = \mathscr{ENC}_k(\mathscr{IV} \mid\mid \mathscr{CTR}_i) \oplus y_i,\ i \geq 1$$

where x_i, y_i, $\mathscr{ENC}_k$, *CTR* are i) plaintext (SNP sequence), ii) ciphertext, iii) encryption key, and iv) counter value, respectively. In our proposed model, *CI* encrypts the data to preserve data privacy and manages the keys necessary for the encryption and decryption (ξ_k, IV, *CTR*). Notably, AES-CTR is proven to be indistinguishable under chosen-plaintext attack or *IND-CPA* secure [10]. The details of utilizing AES-CTR are described in section 3.2.

2.5. Threat model

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

2.5.1. Privacy model

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

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

Formally, hospitals outsource genomic data to the CS with a security guarantee. The proposed method needs to ensure the privacy of the participants from any adversarial attacks over the data. The researchers and the CS aim to execute a private search operation $f(q, \mathscr{D})$ where both parties are unaware of each other's input. Therefore, q and $\mathscr{D}$ will not be revealed to the CS or the researchers, respectively. Furthermore, $f(q, \mathscr{D})$ should only be released to the researchers, as the CS and the hospitals should be oblivious of these results.

2.5.2. Security assumptions

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

- We assume the *CS* to be a semi-honest entity (also known as honest-but-curious) [11] where it follows the protocol but observes the communications and may attempt to derive additional information while executing the *researcher's* query. However, the CS will not act maliciously by deviating from the computation protocol.
- We consider the *CI* to be a trusted entity since it is responsible to create the suffix tree and share necessary keys to the *researcher* to decrypt the encrypted result.
- We consider neither the *hospital*, *CI*, nor the *CS* to behave maliciously to generate incorrect output.
- Moreover, we consider that the *CS* does not collude with *researchers* and the *CI*, and *researchers* do not collude with the *CS*.

2.5.3. Objectives

In the proposed method, we have two particular targets addressing the privacy of the data. Our first objective is to ensure the confidentiality of the genomic dataset $\mathscr{D}$ so that the *CS* does not learn anything about the outsourced data. Secondly, the *researcher*'s query needs to be hidden from the CS, as it can also be sensitive.

3. Methods

In this section, we present our proposed method of secure substring and set-maximal searches. We divide the proposed method into three subprocesses: 1) Generalized suffix tree building, 2) Tree encryption, and 3) Secure search on the encrypted tree: We summarize the notations in Table 2, and Fig. 2 represents the overall protocol of our proposed solution.

3.1. Generalized suffix tree (GST) building

Let $\mathscr{S}$ be a sequence over an alphabet $\sum \in \{0,1\}$, $|\mathscr{S}|$ the length of $\mathscr{S}$, and $\mathscr{S}_i$ with $i \in [1 : |\mathscr{S}|]$ the suffix of $\mathscr{S}$ starting at position *i*. The suffix tree $\mathscr{T}$ of a sequence $\mathscr{S}$ is a rooted tree whose edges are labeled with substrings of $\mathscr{S}$. The *path label* of a node *v* is the addition of all edge label from the root to *v*. Each substring $\mathscr{S}'$ of $\mathscr{S}$ contains only one subtree $\mathscr{T}'$ of $\mathscr{T}$. Every node of the suffix tree is either a leaf node or has two child nodes, at most, starting with either *0* or *1*. The labels on the edges can have any length greater than zero, and no two edges going out from the same node can start with the same character. Therefore, a given (startNode, suffixString) pair can denote a unique path within the tree.

Another aspect of the suffix tree is *suffix link* [12] that constructs the suffix tree in linear time. A suffix link is a link, from node *v* with path label αw, to node v' with path label *w*, where $\alpha \in \sum$ and $w \in \overset{+}{\sum}$. Every inner vertex has a suffix link. Moreover, Ukkonen utilizes *edge-label compression* and *skip/count speed up*, which are described in Ref. [13].

Example 4.1.1. Fig. 3 presents a suffix tree using the sequence 1 of the Table 1. If we traverse the nodes in Fig. 3, we achieve 0, 10, 010, 0010, 00010, *100010*, which are actually the suffixes of the sequence 100010.

The *generalized suffix tree (GST)* presents multiple sequences $\mathscr{S}_1, \ldots, \mathscr{S}_n$ over $\sum \in \{0,1\}$. For this, we can build a suffix tree for $\mathscr{S}_1$ and then extend the tree by incorporating the sequences $\mathscr{S}_2, \ldots, \mathscr{S}_n$. The naive implementation for constructing a suffix tree requires $\mathscr{O}(\mathrm{n}^2)$ time complexity, where *n* is the sequence length [13]. However, Esko Ukkonen [12] reduced this to $\mathscr{O}(n)$ (linear) time using suffix link. The Ukkonen algorithm [12] proceeded forward from the first character to the last one, from the longest to the shortest suffix. However, Ukkonen's suffix tree operates on a single sequence only. Since our dataset contains multiple sequences (Table 1), we constructed the Generalized Suffix Tree (*GST*) presented below:

Example 4.1.2. Fig. 4 presents a GST using the sequence 1 and 2 of the Table 1. It shows the SNP characters along with the suffix positions in

Table 2
Notations.

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

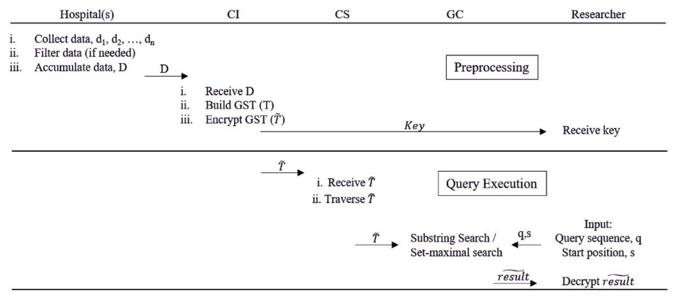


Fig. 2. The overall protocol of our proposed solution.

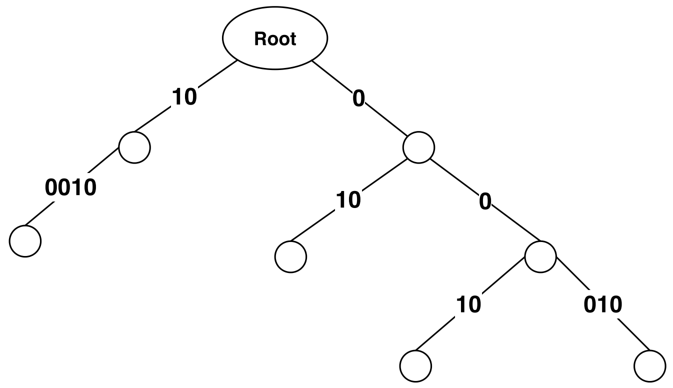


Fig. 3. Suffix tree for a single sequence(100010).

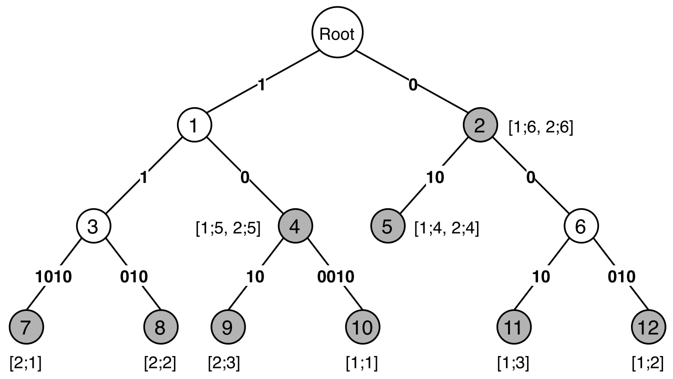


Fig. 4. Generalized Suffix Tree with suffix positions.

each vertex. Each filled circle (vertex) in the Fig. 4 represents suffix(es) including the suffix position and the sequence number that contains the suffix(ex). For example, if we traverse up to node 5, we get the suffix 010 and node attributes $[1;4,2;4]$ which represent both sequence 1 and 2 having the suffix 010 starting from position 4.

3.2. Tree encryption

CI utilizes *AES-CTR* [9] to encrypt the contents in the GST to ensure *data privacy*. The detailed process of GST encryption are as following:

- ***Key generation:*** The *CI* generates the key ($\mathscr{ENC}_k$), Initialization Vector ($\mathscr{IV}$) and a counter ($\mathscr{CTR}$) to encrypt each edge label and node attributes (if any) of the GST.
- ***Encryption:*** At first, *CI* builds a *keystream* by $\mathscr{ENC}_k(\mathscr{IV} \,||\, \mathscr{CTR}_i)$, *where* $i \geq 1$. Then it encrypts each edge label and node attribute (suffix position) by performing *XOR* operation between the edge label/suffix position and *keystream*. We applied two different encryption settings: one with only GST label encryption, and another with both the GST label and the suffix position encryption. We denote the encrypted GST as $\widetilde{\mathscr{GST}}$.
- ***Share:*** *CI* sends the $\widetilde{\mathscr{GST}}$ to *CS* and sends the required keys to the *researcher.*

3.3. Secure search on the encrypted tree

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

Algorithm 1. Secure Substring search on GST

```
Algorithm 1: Secure Substring search on GST
  Input: R, q, pos
  Output: $matched_{seq}$
1 Function secureSearch (R, q, pos)
2     $node_{matched} \leftarrow Algorithm\ 2(q, R)$;
3     if $node_{matched}$ != Null then
4         $enc_{pos} \leftarrow retrieve\ suffix\ position$;
5         $flag \leftarrow eqGC(enc_{pos}, pos)$;
6         if $flag == True$ then
7             $add\ matched\ sequence\ to\ matched_{seq}$;
8         foreach $cNode \in node_{matched}.child$ do
9             $getLeafInfo(cNode, pos)$
10    return $matched_{seq}$;
11 Function getLeafInfo ($node_{matched}$, pos)
12    $enc_{pos} \leftarrow retrieve\ suffix\ position$;
13    $flag \leftarrow eqGC(enc_{pos}, pos)$;
14    if $flag == True$ then
15        $add\ matched\ sequence\ to\ matched_{seq}$;
16    foreach $cNode \in node_{matched}.child$ do
17        $getLeafInfo(cNode, pos)$
```

3.3.1. Secure substring search

We describe the secure substring search process according to the example from Section 2.3 where *researchers* send their query sequence *010* and position *4* to the *CS*. Algorithm 1 describes the secure substring search process. Initially, we declare an empty list called *matched$_{seq}$* to store the matched result (Line 1). Then, the Algorithm 2 finds the particular node that matches the query sequence and the encrypted $\widetilde{\mathscr{GST}}$. In particular, while traversing the $\widetilde{\mathscr{GST}}$, the GC executes an equality test (line no. 8) with the common sub-sequence between the query and the encrypted tree label comparing their minimum length. The detailed procedure is described in the following example:

Example 4.3.1. *Researchers* send the query sequence ($q = 010$) and start position ($pos = 4$) to GC. At first, we find a matched node between the query sequence and the suffix sequences in the $\widetilde{\mathscr{GST}}$ with Algorithm 2. Algorithm 2 starts with finding the edge that matches with the first character of the query and the first character of the encrypted label (line no. 2). This equality checking is performed by the GC. Since the SNPs data are bi-allelic (either 0 or 1), in the worst case, two GC equality checks are necessary to find the matching edge. Moreover, each encrypted edge can have different length of SNPs. Therefore, lines no. 4–7 of Algorithm 2 find equal length of query sequence and encrypted edge label. The GC decrypts the encrypted edge label and performs an equality test with the corresponding query sequence (line no. 8). The algorithm proceeds to the next node if the query and tree label matches (line no. 13). Therefore, for the query sequence $q = 010$, the Algorithm

2 will match encrypted edge label 0, 10 and return node #5. We demonstrate the example by presenting the edge label and node attributes as plaintext for better understanding. After matching the query sequence, we need to check the start position of the query and the suffix position of the matched node. This equality checking is also performed via the secure protocol GC (Algorithm 1, line no. 5). We observe that matched node #5 contains the attributes $[1;4,2;4]$ which matches the query start position $pos = 4$. The CS sends the encrypted matched sequence to the *researchers*, and the researchers obtain the desired result by decrypting. Therefore, it preserves the output *privacy*, as only the CI and the researcher with a specific key can decrypt the result(s).

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

3.3.2. *Secure set-maximal search*

We demonstrate the secure set-maximal search process from the example in Section 2.3 where *researchers* send their query $q = 00011$ and $pos = 2$ to the *CS*. Again, to preserve the *query privacy*, we send the *researcher's* query sequence and position as an input of the *GC*. Algorithm 3 describes the secure set-maximal search process. The difference between the secure substring search and set-maximal search is that in the set-maximal search, the GC checks the number of common prefixes between the query sequence and the encrypted edge label instead of the equality testing. The detailed procedure is described in the following example:

Algorithm 2. Match node (substring search) on GST

```
Algorithm 2: Match node (substring search) on GST
Input: q, node
Output: $matched_{node}$
1  for $i \leftarrow 1$ to $|q|$ do
2      $q \leftarrow q_i$ to $q_{end}$
       $enc_{label} \leftarrow get\ label\ of\ eqGC(q[i], enc_{label}[1])$;
3      $lenToMatch \leftarrow min(|q|, |label|)$;
4      $flag \leftarrow false$;
5      if $|label| > |q|$ then
6          $enc_{label} \leftarrow enc_{label}.substring(0, len_q)$;
7          $flag \leftarrow eqGC(q, enc_{label})$;
8      else
9          $q\prime \leftarrow q.substring(i, len_{label})$;
10         $flag \leftarrow eqGC(q\prime, enc_{label})$;
11     if $flag == True$ then
12         if $|label| >= |q|$ then
13             return $edge.Node()$;
14         else
15             $node = edge.Node()$;
16             $i += lenToMatch - 1$;
```

Example 4.3.2. Researchers send the query $q = 00011$ and start position $pos = 2$ to the GC. Similar to the previous example, we find a matched node between the query sequence and the suffix sequences in the $\mathscr{GST}$ by Algorithm 4. Again, Algorithm 4 starts with finding the edge that matches with the first character of the query and the first character of the encrypted label by the GC (line no. 3). Moreover, lines no. 5–8 of Algorithm 4 find equal length of the query sequence and encrypted edge label. The GC decrypts the encrypted edge label and achieves the number of common prefixes with the corresponding query sequence (line no. 9). The algorithm proceeds to the next node if the query and tree label matches (line no. 13). Therefore, for the query sequence $q = 00011$, the Algorithm 4 will traverse nodes: 2, 6, 12 and return node #12 as the matched node and 4 $(1+1+2)$ as the match count. We demonstrate the example by presenting the edge label and node attributes as plaintext for better understanding. Also, we need to check the start position of the query and suffix position of the matched node by the GC (line no. 5). We observe that matched node #12 contains the attributes $[1;2]$ which match the query start position $pos = 2$.

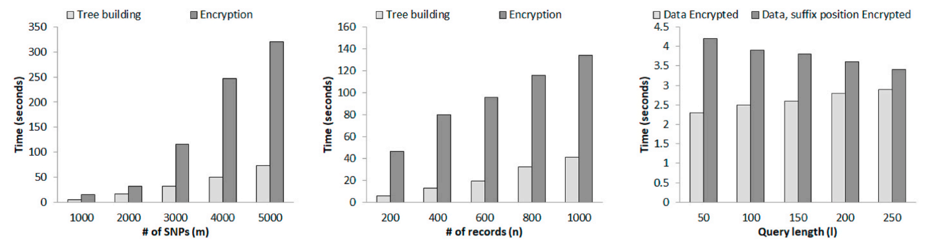
While traversing $\mathscr{GST}$, a number of common prefixes between a particular query and encrypted tree label actually determine the maximum matches between them (line no. 9 of Algorithm 4: noOfMatch). Since the description of the *getLeafInfo* is similar to the Algorithm 1, we do not describe it in Algorithm 3. Moreover, we are not required to execute the GC equality test in line no. 5 in both Algorithm 1 and 3 if we keep the suffix positions unencrypted in secure substring search and set-maximal search.

3.3.2.1. *Complexity analysis.*

Algorithm 2 and 4 represents search complexity of the secure substring and set-maximal search procedure, respectively. For both algorithms, the worst case time complexity of only searching a query sequence are in order of $\mathscr{O}(|q|)$, where $|q|$ is the query length. Since we are only traversing $|q|$ characters using GC on the $\mathscr{GST}$, it should result in a maximum of $|q|$ searches. This is also experimentally presented in Section 4 on Fig. 5c (data encrypted only). Moreover, if we encrypt the tree label and the suffix position of each node, then the worst case time complexity of searching a query sequence depends on $\mathscr{O}(|q|t)$, where $|q|$ is the length of the query sequence and t is the number of records in the dataset that contains the matched sequence. However, due to the properties of GST, the query start positions do not affect the search complexity for equal query length.

Algorithm 3. Secure set-maximal search on GST

```
Algorithm 3: Secure set-maximal search on GST
Input: R, q, pos
Output: $matched_{maxseq}, match_{count}$
1  Function $secureSearch\ (R, q, pos)$
2      $node_{matched}, match_{count} \leftarrow Algorithm\ 4(q, R)$;
3      if $node_{matched}\ != Null$ then
4          $enc_{pos} \leftarrow retrieve\ suffix\ position$;
5          $flag \leftarrow eqGC(enc_{pos}, pos)$;
6          if $flag == True$ then
7              $add\ matched\ sequence\ to\ matched_{maxseq}$;
8          foreach $cNode \in node_{matched}.child$ do
9              $getLeafInfo(node_{matched}, pos)$
10 return $matched_{maxseq}, match_{count}$;
```



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

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

(c) Substring search query execution time for different $|q|$ with 1000×3000 data.

Fig. 5. Tree building, encryption and query execution time.

Algorithm 4. Match node (set-maximal search) on GST

```
Algorithm 4: Match node (set-maximal search) on GST
    Input: q, node
    Output: $matched_{node}, match_{count}$
 1  $match_{count} \leftarrow 0$;
 2  for $i \leftarrow 1$ to $|q|$ do
 3      $q \leftarrow q_i$ to $q_{end}$
         $enc_{label} \leftarrow get\ label\ of\ eqGC(q[i], enc_{label}[1])$;
 4      $lenToMatch \leftarrow min(|q|, |label|)$;
 5      $noOfMatch \leftarrow 0$;
 6      if $|label| > |q|$ then
 7          $enc_{label} \leftarrow enc_{label}.substring(0, len_q)$;
 8          $noOfMatch \leftarrow matchPrefix(q, enc_{label})$;
 9      else
10          $q' \leftarrow q.substring(i, len_{label})$;
11          $noOfMatch \leftarrow matchPrefix(q', enc_{label})$;
12      if $noOfMatch > 0$ then
13          $match_{count} += noOfMatch$;
14          if $|label| >= |q|$ then
15              return $edge.Node(), match_{count}$;
16          else
17              $node = edge.Node()$;
18              $i += lenToMatch - 1$;
```

4. Results

Implementation details: The proposed method of secure substring and set-maximal search was implemented in Java as we utilized a synthetic dataset from the 2018 iDASH competition [8]. The dataset contained *1000* individuals' records with 5000 SNPs. We ran the *CS* and the *CI* in two different machines with Intel Core i7 (*3*.41 GHz processors) and 16 GB memory. We implemented the custom GC protocols using the *FlexSC* [14] library. The code is available in https://github.com/Safiur-Mahdi/SecSS.

To understand the scalability of the proposed model, we experimented with four parameters: the number of SNPs on the genomic sequence *m*, the total number of sequences *n*, the query start position *s*, and the length of the query $|q|$. We further analyzed two encryption settings: 1) GST label encryption, and 2) both GST label and suffix position encryption, which have a direct impact on the privacy and execution time. Notably, the execution time and communication overheads are averaged over 10 random trials.

Experimental Setup and Details: In the following experiments, we are interested in analyzing the GST generation and secure query processing time. Since building (and encrypting) the $\widetilde{\mathcal{GST}}$ is a one-time preprocessing step, it is needed to understand the time required for a specific dataset of n,m. While processing the queries, the query length $|q|$ is also important, as it warrants a different level of search throughout the $\widetilde{\mathcal{GST}}$.

Fig. 5a shows the GST building and encryption time with the edge label and suffix position encrypted. We notice that the data preprocessing time (tree building and encryption) increases linearly with the increase of m as it increases the total number of suffixes. On the other hand, Fig. 5b shows the GST building and encryption time for different number of records $n = \{200, 400, \ldots, 1000\}$ and $m = 3000$. Similarly, preprocessing time increases linearly with the increases of n. Interestingly, the tree construction time is much less in comparison to the encryption time. Notably, this can be done in parallel which can speed up the preprocessing step.

Variable SNPs m: Table 3 represents the query execution time and the communication overhead (fixed n and variable m). In both experiments (substring/set-maximal search), we considered five different $m = \{1000, 2000, \ldots, 5000\}$, $|q| = 25$, and $n = 1000$. Notably, we achieve faster query execution time and lower communication overhead in set-maximal search compared to the substring search, as it requires fewer matches to be performed.

In substring search, we were interested in the exact matching results, whereas set-maximal search demanded the maximum matches between the query and encrypted tree labels. Therefore, a mismatch on initial positions between the query and tree labels can incur faster execution time leading to a more efficient set-maximal search compared to its substring counterpart. We also observed that increasing m neither affects the execution time nor the communication overhead. However, query execution time and communication overhead increases if we encrypt both the data (tree edge label) and suffix position.

Variable number of sequence n: Table 4 presents the same metrics on variant *number of records* $n = \{200, 400, \ldots, 1000\}$, $|q| = 25$ and $m = 3000$. We observe that both query execution time and communication overhead increase linearly with the increase of n since the total matches increase with larger n. Notably, both times and communications are almost similar for both search methods for only encrypted data, whereas it increases linearly for encrypted suffix positions. This is because increasing n demands more equality testing (line 5 in Algorithm 1, 3) on the GC along the encrypted numeric positions.

Variable query length $|q|$: Fig. 5c shows the substring search query execution time for *different query length*, $|q| = \{50, 100, \ldots 250\}$ with a fixed query start position (*s=1*), $n = 1000$ records, and $m = 3000$ SNPs. We also execute this experiment according to two different encryption settings: only GST label (data) encryption, and both the data and the suffix position encryption. While both the data and the suffix position are encrypted, we report that different query length with a fixed query start position affects the query execution time.

We also analyzed the size of the query on the execution time, as it decreases with the increase of $|q|$ partially because the total match count decreases. It is counter-intuitive, as it should require less time to find a match with $|q| = 50$ rather than $|q| = 150$. However, the matched node with query length 50 contains more suffix positions to check compared to the $|q| = 150$. On the other hand, if we execute the same queries with only encrypted data, query execution time increases linearly with the increment of the $|q|$. However, set-maximal search execution time is

Table 3
Query Execution Time (QET) and Communication Overhead (CO) for substring search and set-maximal search on 1000 records with $m \in [1000, 5000]$.

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

Table 4
QET and CO for substring search and set-maximal search on 3000 SNPs with different number of records, $n \in [200, 1000]$.

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

independent of this query length and depends on the number of matches (Tables 3 and 4).

Benchmark. We evaluated our proposed method by comparing with Ishimaki et al. [1] (substring search) and Shimizu et al. [2] (set-maximal search) in Table 5. For the substring search [1], required 16 min to process a secure query consisting of 10 SNPs, whereas our proposed method takes only ≈ 2.3 seconds. Shimizu et al. [2] required around 4 s to solve the secure set-maximal search containing a query of 25 SNPs. In contrast, our proposed method required around 2 s to process. Notably, the implementation was not available for Ishimaki et al. [1] compared to Shimizu et al. [2]'s work, as we could not benchmark it on the same machine. However, Ishimaki et al. employed a far superior computational server, in comparison to ours, to perform their experiments.

5. Discussion

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

5.1. Advantages of our approach

5.1.1. Generalized suffix tree

In our proposed method, the *CI* builds a generalized suffix tree from the accumulated data submitted by the *Hospital*. This is a preprocessing step in our proposed method where we store all suffixes of each sequence of the data in the GST along with the suffix position and sequence number. The main advantage of building the GST is two-fold. Firstly, it categorizes the similar sequence according to the suffix position and sequence number. Secondly, increasing the number of SNPs in the dataset does not affect the query execution time. We can observe this in Table 3. Moreover, we only need one-time pre-processing to build and encrypt the GST.

Table 5
Comparison of secure substring/set-maximal search execution time on 2184 records, each containing 10000 SNPs [2]. Sotiraki et al. [7]'s work was evaluated on $|q|, m = 300$ and $n = 1000$.

Problem	Comparison	Exec time
Secure Substring	Ishimaki et al. [1]	16 min
	Our work	2.3 s
Secure Set-maximal	Shimizu et al. [2]	3.97 s
	Sotiraki et al. [7]	60 s
	Our work	2 s

5.1.2. Secure centralized model

We utilize a centralized (outsourced) model in which the *Hospital(s)* outsource their dataset to the *CS* via a trusted entity, the *CI*. Once outsourced, the *Hospital(s) and CI* can remain offline afterwards. On the other hand, in the distributed (federated) model, the hospitals/data owners do not outsource their dataset to a cloud server. Therefore, the distributed model requires the data owners to remain online during the computation [15].

5.1.3. Privacy guarantees

- ***Data privacy.*** The data is encrypted with AES-CTR and stored on the cloud server. AES-CTR provides a semantically secure encryption scheme and ensures the privacy of the data even if the data gets leaked from a compromised cloud server.
- ***Query privacy.*** As the query is submitted to the GC directly, it is only known to the researcher, and it is not known to any of the other participating entities in the system.
- ***Output privacy.*** The researcher is only aware of the results after decrypting the results, whereas the cloud server only handles the encrypted counterpart. However, we do not prevent any inference attack followed by the outputs of a query, as we assume the researchers to be honest.

5.2. Limitations

5.2.1. Leakage of search pattern in CS

In our proposed method, during the query execution, the *CS* learns the tree traversal pattern since it depends on the *researcher* query input and the GC output. However, the *CS* will not learn or infer any sensitive information, since the value of SNPs value is encrypted, and we assume the *CS* does not collude with the *researcher*. We could solve this leakage by utilizing obliviousRAM [16,17], which will accumulate further computational complexity.

5.2.2. *Malicious researcher*

As described in Section 3.2, *researcher* receives the keys ($\mathscr{ENC}_k$, $\mathscr{IV}$) from the *CI* to decrypt the encrypted result. Therefore, we do not consider any dishonest *researcher* in this article. The *CI* should create and store a profile for each *researcher* and authenticate before sending keys to the *researcher*. Then, as the *researcher* is not anonymous while querying the *CS*, the *CS* monitors the *researcher*'s action to detect any abnormality. If the *CI* catches any misbehaviour, it can block the *researcher*.

5.3. *Future work*

In this paper, both of our queries (secure substring search and secure set-maximal search) need a query string with a smaller length than the dataset sequence length, and a starting position. In future work, we intend to extend our work to support *set-maximal matches* [18] where the query length will be same as the data sequence length and no start position will be given. Moreover, we can add a threshold along with the substring search query so that *researchers* can achieve results with a certain number of matches between the query sequence and any given dataset.

6. Security analysis

The security of the proposed framework relies on two primitives: AES in Counter mode (AES-CTR) and garbled circuits. The security guarantees of AES-CTR [9] are well-analyzed and understood, as it is indistinguishable under chosen plain text (IND-CPA) attacks [10]. Since we are utilizing a different random *IV* in each iteration of the AES-CTR, it will randomize the ciphertexts for same input genomic data (or nucleotide values). In our method, we encrypted the nucleotide values on different nodes and edge values, which need to be hidden from the CS.

Garbled circuits support secure two-party computation against semi-honest adversaries [6]. After running garbled circuits, both parties learn the output of a function but are oblivious about each others' input. The CI encrypts the tree and sends it to the cloud server. The cloud server will not be able to learn and decrypt the tree because it does not have the corresponding keys. The researcher sends the query to the GC directly, and accordingly, the cloud server will not be able to learn it. The researcher and cloud server use garbled circuits to execute queries such that the researcher and the cloud server will not be able to learn each others' input.

Proposition 6.1. *The proposed privacy-preserving method is secure against known cipher-text attacks.*

Proof. Any cryptographic scheme is deemed secure against known ciphertext attacks if it can be proven that the cloud server or adversary cannot get any meaningful information from the encrypted ciphertexts. In our proposed method, the nucleotide values of all genomic data are encrypted using AES-128 CTR cryptosystem. AES-CTR essentially turns AES from block to stream cipher, as the output ciphertext is random, independent of the plaintext. For example, the nucleotide values (A, T, G, C) are repetitive, and they need to be randomized after encryption.

Since the cloud server will only have these randomized sensitive encrypted values on the outsourced suffix tree, it will require an impractical amount of computational time to break the cryptosystem. Furthermore, the 128-bit secret key utilized in the AES-CTR will be secure according to NIST [19].

7. Related work

We can divide the existing body of work for secure string search into three classes as described below:

7.1. *Substring search without a query start position*

Secure substring matching is a familiar method and a popular research area. Katz et al. [20] proposed a secure exact DNA sequence search using garbled circuit [6]. Atallah et al. [21] utilized dynamic programming to compare between two sequences using homomorphic encryption. Other researchers utilized finite automata with `recursive oblivious transfer (ROT)` to solve DNA searching problem [22, 23]. Sudo et al. [24] proposed a unique algorithm where they utilized secure `wavelet matrix` and `additively homomorphic encryption` to search in logarithmic time.

7.2. *Substring search*

Although related, the aforementioned works do not address secure exact substring search based on a starting position. In this paper, we proposed a secure model to find a string match between query and dataset sequences, where the query mentions the starting position. Moreover, we are interested in finding the exact match count rather than the maximal matches. To the best of our knowledge, Ishimaki et al. [1] only addressed this problem as they created a look-up table from `positional Burrows-Wheeler transform (PBWT)` and utilized `FHE` with bootstrapping optimization. Moreover, to reduce the incorrect decryption (due to random noise inside the ciphertexts), the authors proposed two solutions: i) Set a threshold noise to ensure correct decryption, and ii) Reset the noise by utilizing the bootstrapping method, which is crucially expensive.

7.3. *Set-maximal search*

Privacy-preserving set-maximal search is another indispensable method that helps in the identification of distant relatives. Shimizu et al. [2] introduced a secure variable length prefix/suffix match on SNP sequences or regular text. For preprocessing, the authors utilized PBWT proposed by Durbin et al. [3] as the data structure, whereas we utilized a generalized suffix tree to preprocess the data. To assure the privacy, we utilized the garbled circuit, whereas the authors used the `ROT` method of the additive homomorphic encryption as the secure protocol. In a recent solution in 2019, Yamada [25] employed both fully and somewhat both homomorphic encryption schemes for a string search framework. However, the query lengths were only selected from $[5, 25]$ taking a maximum of 80 s, which is not realistic in a real-life genomic search operation.

In 2019, iDASH [8] organized a competition for securely computing set-maximal matching in a two-party setting. The most accurate solution was proposed by Microsft Research [7], and it used Goldreich-Micali-Wigderson (GMW) protocol [26]. Here, the solution required a private evaluation of XOR, AND and NOT operation between the two parties. Nevertheless, the proposed solution takes around 60 s for a database of 1000×1000 and query size 300. Unfortunately, none of the submitted solutions in iDASH 2020 were successful for 100k SNPs.

Table 6 summarizes the existing approaches for privacy-preserving string search. We categorize the previous approaches based on query type and the requirement of the query start position.

8. Conclusion

This paper presents a secure protocol for efficient substring and set-maximal search over genomic data. We proposed a *generalized suffix tree* index and proposed the private query execution ensured by the garbled circuit. Comparison with earlier works (Ishimaki et al. [1] and Shimizu et al. [2]) demonstrates a significant improvement in query execution time. The state-of-the-art results from Sotiraki et al. [7] are also slower due to the GST index. In the future, we intend to extend the proposed method to perform other variants of substring and approximate string searches (i.e., without a predefined query position) using the

Table 6
Previous research works in secure and privacy-preserving string search in genomic sequence (HE = Homomorphic Encryption, ROT = Recursive Oblivious Transfer, GC = Garbled Circuit, AHE = Additively Homomorphic Encryption, FHE = Fully Homomorphic Encryption).

Existing techniques	Year	Query type	Query start position	Principal Method
Atallah et al. [21]	2003	Substring search	×	Dynamic programming with HE
Troncoso et al. [23]	2007	Substring search	×	Finite automata with ROT
Sudo et al. [24]	2015	Substring search	×	Wavelet matrix with AHE
Shimizu et al. [2]	2016	Set-maximal search	✓	BWT with ROT
Ishimaki et al. [1]	2016	Substring search	✓	BWT with FHE
Sotiraki et al. [7]	2016	Set-maximal match	✓	GMW [27]
Our method	2019	Substring & set-maximal search	✓	GST with GC

Burrows-Wheeler transformation technique. Furthermore, the expensive encrypted GST building and searching can be improved with parallel data and query processing utilizing parallel HE frameworks.

Statements of ethical approval

None to be provided.

Funding

The research is supported in part by the Amazon Research Grant, NSERC Discovery Grants (RGPIN-2015-04147). Jiang, X is CPRIT Scholar in Cancer Research, and he was supported in part by Christopher Sarofim Family Professorship, the CPRIT RR180012, UT Stars award, the National Institute of Health (NIH) under award number R01GM114612 and U01TR002062.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

We cordially thank the iDASH Privacy and Security Workshop for providing the dataset.

References

[1] Ishimaki Y, Imabayashi H, Shimizu K, Yamana H. Privacy-preserving string search for genome sequences with FHE bootstrapping optimization. In: 2016 IEEE international conference on big data (big data). IEEE; 2016. p. 3989–91.
[2] Shimizu K, Nuida K, Rätsch G. Efficient privacy-preserving string search and an application in genomics. Bioinformatics 2016;32(11):1652–61.
[3] Durbin R. Efficient haplotype matching and storage using the positional Burrows–Wheeler transform (PBWT). Bioinformatics 2014;30(9):1266–72.
[4] Naveed M, Ayday E, Clayton EW, Fellay J, Gunter CA, Hubaux J-P, Malin BA, Wang X. Privacy in the genomic era. ACM Comput Surv 2015;48(1):6.
[5] Guerrini CJ, Robinson JO, Petersen D, McGuire AL. Should police have access to genetic genealogy databases? capturing the golden state killer and other criminals using a controversial new forensic technique. PLoS Biol 2018;16(10):e2006906.
[6] Yao A. How to generate and exchange secrets. In: Foundations of computer science, 1986., 27th annual symposium on. IEEE; 1986. p. 162–7.
[7] Sotiraki K, Ghosh E, Chen H. Privately computing set-maximal matches in genomic data. BMC Med Genom 2020;13(7):1–8.
[8] T.-T. Kuo, X. Jiang, H. Tang, X. Wang, T. Bath, D. Bu, L. Wang, A. Harmanci, S. Zhang, D. Zhi, et al., Idash secure genome analysis competition 2018: blockchain genomic data access logging, homomorphic encryption on gwas, and dna segment searching, BMC Med Genom 13 (Suppl 7).
[9] Paar C, Pelzl J. Understanding cryptography: a textbook for students and practitioners. Springer Science & Business Media; 2009.
[10] Katz J, Lindell Y. Introduction to modern cryptography. CRC press; 2014.
[11] G. Oded, Foundations of cryptography: Volume vol. 2, basic applications.
[12] Ukkonen E. On-line construction of suffix trees. Algorithmica 1995;14(3):249–60.
[13] Gusfield D. Algorithms on strings, trees and sequences: computer science and computational biology. Cambridge university press; 1997.
[14] Wang X, Chan H, Shi E. Circuit oram: on tightness of the goldreich-ostrovsky lower bound. In: Proceedings of the 22nd ACM SIGSAC conference on computer and communications security. ACM; 2015. p. 850–61.
[15] Salem A, Berrang P, Humbert M, Backes M. Privacy-preserving similar patient queries for combined biomedical data. In: Proceedings on Privacy Enhancing Technologies; 2019. p. 47–67. 1.
[16] Gentry C, Goldman KA, Halevi S, Julta C, Raykova M, Wichs D. Optimizing ORAM and using it efficiently for secure computation. In: International symposium on privacy enhancing technologies symposium. Springer; 2013. p. 1–18.
[17] Stefanov E, Van Dijk M, Shi E, Fletcher C, Ren L, Yu X, Devadas S. Path ORAM: an extremely simple oblivious RAM protocol. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. ACM; 2013. p. 299–310.
[18] iDASH Privacy and security workshop 2018-secure genome analysis competition. http://www.humangenomeprivacy.org/2018/competition-tasks.html,online. accessed 24 January, 2019.
[19] Keylength - cryptographic key length recommendation. https://www.keylength.com/,online. accessed 29 May 2019.
[20] Katz J, Malka L. Secure text processing with applications to private DNA matching. In: Proceedings of the 17th ACM conference on Computer and communications security. ACM; 2010. p. 485–92.
[21] Atallah MJ, Kerschbaum F, Du W. Secure and private sequence comparisons. In: Proceedings of the 2003 ACM workshop on Privacy in the electronic society. ACM; 2003. p. 39–44.
[22] Blanton M, Aliasgari M. Secure outsourcing of DNA searching via finite automata. In: IFIP annual conference on data and applications security and privacy. Springer; 2010. p. 49–64.
[23] Troncoso-Pastoriza JR, Katzenbeisser S, Celik M. Privacy preserving error resilient DNA searching through oblivious automata. In: Proceedings of the 14th ACM conference on Computer and communications security. ACM; 2007. p. 519–28.
[24] Sudo H, Jimbo M, Nuida K, Shimizu K. Secure wavelet matrix: alphabet-friendly privacy-preserving string search. BioRxiv. 2016. 085647.
[25] Yamada Y, Rohloff K, Oguchi M. Homomorphic encryption for privacy-preserving genome sequences search. In: 2019 IEEE international conference on smart computing. SMARTCOMP; 2019. p. 7–12. https://doi.org/10.1109/SMARTCOMP.2019.00021.
[26] Goldreich O, Micali S, Wigderson A. How to play any mental game, or a completeness theorem for protocols with honest majority. In: Providing sound foundations for cryptography: on the work of shafi goldwasser and silvio micali; 2019. p. 307–28.
[27] G. O. Foundations of cryptography:. first ed., vol. 2. New York, NY, USA: Cambridge University Press; 2009. Basic Applications.