Privacy-Preserving Search for a Similar Genomic Makeup in the Cloud

Xiaojie Zhu, Erman Ayday, Member, IEEE, Roman Vitenberg, Member, IEEE, and Narasimha Raghavan Veeraragavan

Abstract—Increasing affordability of genome sequencing and, as a consequence, widespread availability of genomic data opens up new opportunities for the field of medicine, as also evident from the emergence of popular cloud-based offerings in this area, such as Google Genomics [1]. To utilize this data more efficiently, it is crucial that different entities share their data with each other. However, such data sharing is risky mainly due to privacy concerns. In this article, we attempt to provide a privacy-preserving and efficient solution for the “similar patient search” problem among several parties (e.g., hospitals) by addressing the shortcomings of previous attempts. We consider a scenario in which each hospital has its own genomic dataset and the goal of a physician (or researcher) is to search for a patient similar to a given one (based on a genomic makeup) among all the hospitals in the system. To enable this search, we propose a hierarchical index structure to index each hospital’s dataset with low memory requirement. Furthermore, we develop a novel privacy-preserving index merging mechanism that generates a common search index from individual indices of each hospital to significantly improve the search efficiency. We also consider the storage of medical information associated with genomic data of a patient (e.g., diagnosis and treatment). We allow access to this information via a fine-grained access control policy that we develop through the combination of standard symmetric encryption and ciphertext policy attribute-based encryption. Using this mechanism, a physician can search for similar patients and obtain medical information about the matching records if the access policy holds. We conduct experiments on large-scale genomic data and show the high efficiency of the proposed scheme.

Index Terms—Genome privacy, dataset outsourcing, similar patient search

1 INTRODUCTION

Thanks to the sharp cost reduction in the whole genome sequencing, today, digital genomes are used in many applications such as paternity tests [2], personalized medicine [3], and genetic compatibility tests [4]. Among these uses of genomic data, arguably the most important one is in healthcare. Physicians now treat their patients based on their genetic makeup. They provide different prescriptions to patients having the same disease but with different genetic makeup. Thus, it is very valuable for a physician to identify other patients that are in similar conditions to their patient to get more insight about the diagnosis and treatment procedures. However, doing such a search in a broad fashion has many unique challenges as we discuss in the following.

Confidentiality of Genomic Data and Affiliated Sensitive Information (ASI). There has already been several privacy concerns related to genomic data [5], [6]. Since genomic data includes information about an individual’s phenotype, ethnicity, family members, disease conditions, and more, if it falls into wrong hands, the consequences may be as serious as genetic discrimination (e.g., in healthcare or employment). Genomic data is often associated with the medical condition of a patient, including diagnosis, treatment, and symptoms. We refer to this information as “affiliated sensitive information” (ASI) of the patient. For instance, a mutation in the BRCA gene is recognized as a major contributor for breast cancer [7], similarly the treatment of many cancer types are determined based on the genetic makeup of a patient. Protecting the confidentiality of patients’ genomic data and ASI is essential for the hospitals, and hence, hospitals are not willing to open their datasets to each other or share their datasets with a public cloud service provider (CSP). Therefore, genomic data sharing mechanisms that provide privacy guarantees to the hospitals about their datasets are required to pave the way to an efficient and privacy-preserving large-scale similar patient search protocol.

Efficiency of the Search Process. The search process to identify a target genome sequence (i.e., a similar patient) should be efficient. However, considering the scale of genomic data and the scale of the search (i.e., number of hospitals and the number of patients in each hospital), providing an efficient protocol along with the privacy goals is not trivial. One obvious approach is to apply index structures (e.g., suffix tree, prefix tree, or binary tree) to make the search process more efficient. However, such indexing techniques cannot be directly applied for the genome search due to (i) the large size of genome data: The traditional index structures, e.g., B-tree and R-tree, result in a disjoint decomposition of space [8]. Similarly, the prefix tree and suffix tree are inefficient in dealing with palindrome and sequence with many repeated
segments since both of them sequentially process all the data. Considering the size of genome data, the sequential scan of the genome sequence is inefficient; (ii) unpredictable variation of genomic data between individuals: The difference of genome sequences between two individuals is hard to predict since the mutations are not predictable. Thus, an index structure that strictly relies on a previously known data distribution (e.g., cell tree [9]) cannot be applied to index genome data; (iii) the privacy requirement: Existing index structure (e.g., B-tree and R-tree) is proposed to build an index based on plain data, which is not privacy-preserving. In order to achieve the privacy of indexed data and index itself, a new data structure should be proposed. Thus, new techniques are required to provide both privacy guarantees and efficiency for similar patient search problem.

Search Over Several Parties. Searching for similar patients is more effective and helpful if the physician can search datasets of more hospitals. Existing studies [10], [11] support search over several parties by assuming that the physician queries all hospitals individually (one-by-one). However such a strategy is both time consuming and unreliable since it requires the cooperation of each hospital in real-time. Instead, it would be easier if all hospitals outsource their datasets to a common entity (e.g., a CSP) and the physician directly queries this CSP. However, such an approach is not trivial due to privacy concerns, mainly due to (i) the privacy requirement: Since the search process is across different parties, the privacy of each party should be provided. This means a party should have full control of its data and should decide whether a client is allowed to access its data or not. In such cases, the search process is only among the allowed parties’ data; (ii) dynamic consortium: A party could join in a consortium to allow a client to search genome data over its data. Meanwhile, a party is also allowed to leave from the consortium; (iii) efficiency requirement: With the increasing number of parties, efficiency should be considered. Especially, when the combined data volume is huge, genome search among different parties should be practical. Therefore, we need new solutions to share datasets among several hospitals in a privacy-preserving and efficient way.

In this paper, to the best of our knowledge, we propose the first framework to tackle all these challenges. We propose a scheme in which each hospital encrypts its own dataset (with its unique key) and outsources the storage and processing for the search operation to a CSP. For privacy, we encrypt genomic data with a standard encryption algorithm and propose a novel indexing mechanism for privacy-preserving search. The proposed indexing mechanism provides not only privacy, but also the ability to search over several hospitals’ datasets in an efficient way. Each hospital encrypts its own dataset independently while the searchability of ciphertext is enabled across all the hospitals through this indexing mechanism.

In order to achieve efficient search and outsource computation-intensive tasks to the CSP, we propose two mechanisms to advance the proposed indexing scheme. First, we propose a hierarchical clustering algorithm and a hierarchical index structure to accelerate the search process. Second, we introduce a privacy-preserving index merging algorithm to avoid CSP sequentially searching over all the stored hierarchical index structures (e.g., belonging to different hospitals) one-by-one. To enable the ASI to be properly accessible by legitimate clients, we also introduce an ASI sharing scheme. For this, considering the fine-grained access requirement, we adopt chosen policy attribute-based encryption (CPABE). In addition, to enable participants to use different secret keys to encrypt the ASI, we introduce a re-encryption mechanism. We implement and evaluate the proposed scheme under various scenarios. Also, we show that compared with the state-of-the-art, the proposed scheme performs more than 60 times faster than Wang et al.’s protocol [10] and more than 95 times faster than Asharov et al.’s [12] and Schneider et al.’s [13] schemes, especially for large query sizes.

In summary, our contribution is as follows.

1) As opposed to previous work, here, we provide a significantly more practical and efficient solution by letting the hospitals outsource the storage of their datasets to a single cloud service provider (CSP) in a privacy-preserving way.

2) To provide the privacy of outsourced data, we let each data owner (hospital) encrypt its data with its unique cryptographic key. Thus, as opposed to similar work that use a CSP to process data from multiple sources, we avoid single point-of-failure by encrypting all the outsourced data with different keys.

3) We also consider a dynamic system in which new hospitals join by uploading their datasets to the CSP in an efficient way. We provide these functionalities via a novel indexing scheme and a novel privacy-preserving index merging algorithm.

4) Furthermore, we consider controlled access to affiliated sensitive information (ASI) such as diagnosis, treatment, or symptoms that can be associated with genomic information. We provide fine-grained access control to ASI so that an authorized physician can not only identify similar patients, but she can also obtain medical information about them.

2 RELATED WORK

Privacy of genomic data has been recently a very active research topic [14]. Several privacy-preserving solutions have been proposed for processing of genomic data in different settings, including personalized medicine [15], research [16], [17], alignment [18], and management of raw genomic data [19].

There has been earlier work on privacy-preserving pairwise comparison of genomes (or identification of a pattern in a given DNA sequence). Atallah et al. proposed a privacy-preserving edit distance protocol based on dynamic programming [11]. Computational efficiency of this work was later improved by Jha et al. [20]. Troncoso-Pastoriza et al. proposed a protocol to execute finite state machine (FSM) in an oblivious manner [21]. Huang et al. utilized the order-preserving encryption to retrieve matching genomes [22]. Shimizu et al. constructed oblivious transfer through additively homomorphic encryption to evaluate the closeness of two sequences [23]. Yasuda et al. applied somewhat homomorphic encryption (SWHE) to implement privacy-preserving Hamming distance computation of two genome sequences [24]. Cheon et al. used SWHE to implement...
secure edit distance computation of two genome sequences [25]. Wang et al. proposed a scheme for DNA sequence matching with only one-round of interaction [26]. Sousa et al. combined SWHE and private information retrieval (PIR) to implement secure search over outsourced VCF files [27]. Furthermore, Cheng et al. proposed secret sharing (using two non-colluding public clouds) for similarity computation between genome sequences [28]. Although these schemes are useful for pairwise comparison of genomes (or comparison of a pattern and a genome), they cannot be generalized for 1-to-n comparison between the genomes easily due to efficiency and practicality issues, and hence they are not applicable for the similar patient search problem.

Similar to our proposed work, privacy-preserving similar patient search has been considered by a few works. Wang et al. proposed an efficient genome-wide, privacy-preserving similar patient query scheme for two parties [10]. In their scheme, the edit distance of two genome sequences is transformed into finding the number of different elements between two sets. Asharov et al. addressed the same problem by pre-processing genome sequences into proper fragments before comparison [12]. Both these works assume that genomic data is stored at local datasets (e.g., each hospital storing its own genomic dataset) and the client (physician) looks for the top k-closest sequences at each local dataset. This makes the search process impractical since both schemes require each hospital to be available all the time and responsive to the queries. Schneider et al. [13] adopted Asharov et al.’s solution to support outsourcing. In a nutshell, their scheme is a secret sharing-based mechanism, in which the data owners outsource the dataset storage to two semi-trusted service providers. A client’s query is generated by interacting with these service providers. Schneider et al.’s scheme relies on the existence of more than one (semi) trusted and non-colluding entities, which is hard to be realized in real-life. Furthermore, hospitals are typically reluctant to outsource their medical datasets to cloud-based service providers without encryption. Therefore, secret sharing-based solutions, although efficient, are not practical for real-life implementation of this scenario. Moreover, none of the above works consider affiliated sensitive information about genomic data, such as diagnosis, treatment, or symptoms, which plays a significant role in practical applications. Note that it is non-trivial to properly integrate this feature to existing works [10], [12], [13].

3 BACKGROUND
Here, we provide brief background information about genomics and less-common cryptographic primitives we use in this work.

3.1 Genomics Background
The most common mutation in human population is called single nucleotide polymorphism (SNP). It is the variation in a single nucleotide at a particular position of the genome [29]. There are about 5 million SNPs observed per individual and sensitive information about individuals (such as disease predispositions) are typically inferred by analyzing the SNPs. Two kinds of nucleotides (or alleles) are observed for each SNP: (i) major allele is the one that is observed with a high frequency and (ii) minor allele is the one that is observed with low frequency. The frequency of the minor allele in a given population is denoted as the minor allele frequency (MAF). Each SNP includes two nucleotides, one inherited from the father and the other one from mother. For simplicity, we represent the value of a SNP i as the number of its minor alleles, and hence $SNP_i \in \{0, 1, 2\}$. A SNP is represented by an (ID, value) pair, where the ID is taken from a large standardized set of strings and the value is in $\{0, 1, 2\}$. In the following sections, if we mention a SNP (or SNPs) without mentioning the ID or value, we mean both parts.

3.2 Ciphertext Policy Attribute-Based Encryption (CPABE)
CPABE enables controlled access to encrypted data [30]. It consists of the following four algorithms.

Setup. Outputs the public parameters $PK$ and a master key $MK$, given a security parameter.

Access structure generation - CPABE.AP(θ). Outputs an access structure $A$ given a set of attributes $θ$.

Encryption - CPABE.Enc($PK, M, A$). Takes as input the public parameters $PK$, a message $M$, and an access structure $A$ over the universe of attributes. The algorithm encrypts $M$ and produces a ciphertext $CT$ such that only a client that possesses a set of attributes that satisfy the access structure $A$ can decrypt $CT$.

Key generation - CPABE.KeyGen($MK_1, S$). Outputs a private key $sk$ given the master key $MK$, and a set of attributes $S$.

Decryption - CPABE.Dec($PK, CT, sk$). Takes as input the public parameters $PK$, a ciphertext $CT$, which contains an access structure $A$, and $sk$, which is a private key for a set $S$ of attributes. If $S$ satisfies the access structure $A$, then the algorithm decrypts the ciphertext $CT$ and returns a message $M$.

3.3 Customized Bloom Filter
Compared with the standard Bloom filter (BF), the customized Bloom filter (CBF) uses one perfect hash function instead of $k$ normal hash functions. The perfect hash function [31] for a set of data items is a hash function that maps distinct elements in the set to a set of integers with no collisions. These integers are further utilized as indices of a bit array and corresponding values are set to 1. In the remaining of the paper, if we do not specify the type of the Bloom filter, then it is the standard one.

4 Problem Formulation
We now introduce our system, threat, and query models. Frequently used parameters and functions are listed in Table 1.

4.1 System Model
As shown in Fig. 1, our proposed model consists of four entities: data owner (DO), certificated institution (CI), cloud service provider (CSP), and client (e.g., physician). The DO can be considered as the hospital. The hospital collects biological samples from patients with their consent and sends the samples to the CI for sequencing. The CI is an authority or trusted institution that is responsible for sequencing DNA and generating the VCF files (the file format to store the SNPs of individuals). Upon receiving VCF files from the
TABLE 1
Key Parameters and Functions

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G_1/G_2/G_T$</td>
<td>a group of prime order $p$</td>
</tr>
<tr>
<td>$c$</td>
<td>a bilinear mapping from $G_1/G_2$ to $G_T$</td>
</tr>
<tr>
<td>$bf$</td>
<td>maps the input into a Bloom filter</td>
</tr>
<tr>
<td>$bf^{C^*}$</td>
<td>maps the input into a customized Bloom filter</td>
</tr>
<tr>
<td>$Dict_i$</td>
<td>a dictionary of hospital $i$ that stores pairs including (i) pseudonym of a patient and (ii) Bloom filter output of the corresponding patient’s genome</td>
</tr>
<tr>
<td>$Dict_i^{CBF}$</td>
<td>a dictionary of hospital $i$ that stores</td>
</tr>
<tr>
<td>$Dict_i^{MBF}$</td>
<td>a merged dictionary from multiple hospitals</td>
</tr>
<tr>
<td>$Dict_i^{ASI}$</td>
<td>a dictionary of hospital $i$ that stores</td>
</tr>
<tr>
<td>$S$</td>
<td>a set of SNPs</td>
</tr>
<tr>
<td>$S_{i,SNPs}$</td>
<td>a set of SNPs related to $ASI_i$ from hospital $i$</td>
</tr>
<tr>
<td>$H_0$</td>
<td>maps two strings to a random string</td>
</tr>
<tr>
<td>$H_1$</td>
<td>maps a string to an element of group $G_1$</td>
</tr>
<tr>
<td>$H_2$</td>
<td>maps two elements from $G_T$ to a string</td>
</tr>
<tr>
<td>$F$</td>
<td>a pseudorandom function (PRF)</td>
</tr>
<tr>
<td>$K$</td>
<td>a secret key, used by all hospitals and approved clients</td>
</tr>
<tr>
<td>$K_c$</td>
<td>a secret key selected by a client</td>
</tr>
<tr>
<td>$\varepsilon_c$</td>
<td>a threshold of minimum number of matching SNPs set by client $c$</td>
</tr>
<tr>
<td>$k_c$</td>
<td>a threshold of maximum retrieved result set by client $c$</td>
</tr>
<tr>
<td>$K_i$</td>
<td>a secret key selected by hospital $i$ for ASI encryption and shared key generation</td>
</tr>
<tr>
<td>$K_{asi}$</td>
<td>a symmetric encryption key of hospital $i$ for ASI encryption</td>
</tr>
<tr>
<td>$K_{asi}^{gb}$</td>
<td>a symmetric encryption key of hospital $i$ for ASI encryption, which is shared with authorized clients</td>
</tr>
<tr>
<td>$K_y$</td>
<td>a symmetric encryption key of hospital $i$ randomly selected from $G_T$</td>
</tr>
<tr>
<td>$PK_{asi}, SK_i$</td>
<td>a pair of public/private keys selected by hospital $i$ for signature</td>
</tr>
<tr>
<td>$PK_{asi}, MK_i$</td>
<td>a pair of public and master keys of CPABE selected by hospital $i$ and shared with the CSP</td>
</tr>
</tbody>
</table>

CI, hospital first processes them (e.g., generating the complementing ASI, indexing, and encryption) and then, outsources the storage of the encrypted dataset to the CSP. The CSP stores the uploaded encrypted datasets and responds to the queries of the clients for similar patient search. After a client is authenticated by a hospital (e.g., to make sure that she is a legitimate physician), she can issue a query to the CSP to search over the stored data belonging to the corresponding hospital(s). Upon receiving the search result from the CSP, the client further processes the retrieved result and obtains the plaintext response.

4.2 Threat Model
We assume that the CI is a trusted party, which is consistent with the previous work [19], [32]. The CI is only responsible for the sequencing. Due to the nature of today’s sequencing technology, existence of such a trusted CI is a mandatory assumption for all existing schemes.

Following the common practice in this area [10], [12], we assume the CSP to be semi-honest. Under this assumption, the CSP follows the protocol honestly and may be curious to infer stored data by analyzing the received queries and the stored data. Similarly, the client may be curious about hospitals’ sensitive information (genomic information and ASI stored at the CSP). In the proposed scheme, if the CSP and one of the other parties (a hospital or a client) colludes, the CSP may launch a brute-force attack to infer the SNP IDs and values contained in the indices of hospitals (that store their data at the CSP). However, the computational complexity of such an attack is exponential in the number of SNP (which is on the order of millions). In this paper, we do not consider such a collusion.

We briefly discuss the main threats against the proposed protocol in the following.

Ciphertext Analysis Attack. The CSP may attempt to infer the sensitive information of the hospitals (which includes the actual genomic data, ASI data, and the index), by analyzing stored encrypted data.

Query Analysis Attack. The CSP observes and processes the query from the client, and hence it may try to infer the query content (i.e., genomic data of the patient being queried).

Illegitimate Access Attack. The client may try to access genomic data or ASI from a hospital’s dataset without the authorization of the corresponding hospital.

We prove the robustness of the proposed scheme against these attacks in Section 6.

4.3 Query Model
The query model is designed to provide the following functionality: given a (partial) sequence of SNPs representing a set of mutations for a patient, retrieve ASIs of patients whose mutations are similar to those of the given sequence. The input sequence does not need to include all mutations for a patient because the focus of the query can be on a specific pattern that includes a number of SNPs. The search is performed across the data from multiple hospitals under the constraint of access control. We let the client customize the search query by introducing two search parameters as follows:

Threshold for Similarity Metric ($\varepsilon_c$). In the search phase, the parameter is used to evaluate whether the match for a patient exceeds the threshold.

Threshold for the Number of Retrieved Results ($k_c$). In the proposed scheme, this parameter is used to control the size of search result.

Fig. 1. System model.
5 PROPOSED SCHEME

5.1 Overview

We divide the solution into three phases: initialization, client authorization, and query processing, as shown in Figs. 2, 3, and 4, respectively. The initialization is performed infrequently, depending on how dynamic the system is. The client authorization can be performed periodically, in line with common practices. The query processing is performed each time a user wants to send a search query. Each phase consists of a number of procedures (the summary of core procedures is shown in [33]). We first present an overview of the procedures and then, provide their detailed descriptions.

In the initialization phase (shown in Fig. 2), each hospital first calls the Setup function. Setup chooses the initial parameters, configures library functions, and then preprocesses the dataset (e.g., by adding ASI to associated SNPs). After performing Setup, each hospital runs the IndexGen algorithm to build an index over its genomic data. The index is generated based on the genome similarity of its patients. Then, the encryption algorithms SNPEncrypt and ASIEncrypt are called to encrypt the genome sequences and corresponding ASIs. To outsource the computation-intensive tasks to the CSP, the hospital directly sends the generated index and encrypted data to the CSP without building a hierarchical index over it. Upon receiving the encrypted index, the CSP runs the HierarchicalIndexGen algorithm to build a hierarchical index in order to improve the search efficiency. Since each hospital outsources its indices to the CSP, the number of hierarchical indices stored at the CSP increases with the number of hospitals. If the number of hierarchical indices is beyond a certain threshold (that is determined by considering the efficiency of the search operation), the CSP calls the IndexMerge function to merge all the hierarchical indices into one. We analyze the value of this threshold in Section 7.

If a client wants to perform a similar patient search, she first needs to get authorization from a hospital before she can generate a valid query. Fig. 3 shows the process of client authorization. The client sends an authorization request to a hospital. If the hospital approves the request, it generates a token adjustment key (as will be described later) and sends it to the CSP. Then, the hospital sends additional secret keys to the client, as described in Section 5.3.

Once the client gets the authorization from a hospital, she can query the dataset of that hospital. An important advantage of our scheme is that the client can get an authorization from multiple hospitals and later send a single query to perform a search across all of them. Fig. 4 shows the flow of query processing. The client first calls the QueryGen function to generate the first part of her query, which is used to search for the pseudonyms of the target similar patients. Then, the TokenGen function is called to construct tokens as the second part of the query, which is used to retrieve the target ASIs. Token is constructed by encrypting client’s input SNPs and it is adjusted using the token adjustment key (that is generated by the hospital and sent to the CSP as part of client authorization). The adjusted token is used to provide controlled (or authorized) access to the client to the hospital’s data. Upon receiving the query, the CSP first calls either Search or SearchOverMergedIndex function, depending on whether the algorithm IndexMerge has been called or not, with the first part of the query to retrieve the pseudonyms of target similar patients.

If the output of the first step is non-empty, the CSP adjusts the token using the second part of the query by running TokenAdjust. Finally, the CSP calls the ASISearch function to retrieve the ASIs belonging to the retrieved target patients. The result is sent back to the client. The client decrypts the received ciphertext by running ASIDecrypt and obtains the plaintext ASIs of the corresponding target patients.

5.2 Initialization

As shown in Fig. 2, the initialization consists of six modules: Setup, IndexGen, SNPEncrypt, ASIEncrypt, HierarchicalIndexGen, and IndexMerge. The first four modules are done at
each hospital and the remaining modules are done at the CSP.

To initialize the system, a trusted party that is in charge of key generation and distribution (such as the NIH) sets an asymmetric bilinear group \((G_1, G_2, G_T, p, e)\), where \(G_1\) and \(G_2\) are two distinct groups of order \(p\), and \(e\) is the mapping from these two groups to the target group \(G_T\). In addition, the settings (i.e., size and maximum false positive rate) of the standard Bloom filter \(BF\) and customized Bloom filter \(CBF\) are configured. After that, three hash functions \((H_0, H_1, H_2)\) are chosen. These hash functions are used in the algorithms that will be discussed later. Subsequently, a pseudo-random function (PRF) \(F\) is chosen and a secret key \(K\) is selected. \(K\) is shared by all the hospitals and it is used to prevent the CSP to launch a brute-force attack (to infer the genome data contained in the indices of the hospitals that store their datasets at the CSP). Even if \(K\) is leaked to a client by a hospital, it does not pose a privacy threat for the datasets of other hospitals. If \(K\) is learnt by the CSP, the CSP may launch a brute-force attack (to infer the genome data contained in the indices) by exhaustively trying all SNP values. However, the computational complexity of such an attack is exponential in the number of SNP (which is on the order of millions).

### 5.2.1 Setup (at the Hospital) - Setup

In the first step of the Setup, all the initial parameters and functions are selected. In the second step, dataset is preprocessed. The following steps are the same for all hospitals, and hence we describe them only for an hospital \(i\). Hospital \(i\) first generates two symmetric encryption keys, \(K_a\) and \(K_r\) (only shared with approved clients), for the SNP encryption and ASI encryption, respectively. Note that different hospitals use different secret keys. Then, it chooses a secret key \(K_i\) for the shared key generation. Furthermore, a public/private key pair \((PK_{i,1}, SK_{i})\) is generated for the signature and a public/master key pair \((PK_{i,2}, MK_i)\) is generated for CPABE.

Next, the hospital starts preprocessing the dataset. The two phases of dataset preprocessing are shown in Fig. 5. In the first phase, the hospital adds a pseudonym for each patient record and ASIs for various groups of SNPs. The association between the ASIs and the SNPs can be determined based on several factors such as disease, phenotype, or treatment. Eventually, data record belonging to a patient \(ID_i\) is represented as \(\{ID_i, \{S_{ID_i,SNP, ASI_1}^{ASI_1}, \ldots, S_{ID_i,SNP, ASI_k}^{ASI_k}\}\}\), where \(S_{ID_i,SNP}^{ASI_k}\) represents a set of SNPs of individual \(ID_i\) (and their values) that are associated with \(ASI_k\). In the second phase, the concatenation of each SNP ID (represented by SNP.ID) and SNP value (represented by SNP.val) is mapped into a Bloom filter, denoted as \(bf_{ID_i}\) (shown in Fig. 5). Eventually, this process generates the non-zero elements of the Bloom filter output that are associated with the ASIs of the corresponding patient.

### 5.2.2 Index Generation (at the Hospital) - IndexGen

Each hospital indexes the records of its patients using a Bloom filter. The index generation algorithm is used to encrypt and randomize the non-zero elements of the Bloom filter. The same algorithm is also used to generate the query (as discussed in Section 5.4.1). The details of the index generation (IndexGen) algorithm are given in Algorithm 1. The input of the IndexGen algorithm at hospital \(i\) are the secret key \(K\), the pseudo-random function (PRF) \(F\), the dictionary \(Dict_i^{BF}\), and the public/private key pair \((PK_{i,1}, SK_{i})\).

#### Algorithm 1. IndexGen

**Input:** a secret key \(K\) of PRF, dictionary \(Dict_i^{BF}\), public key \(PK_{i,1}\) of signature, private key \(SK_{i,1}\) of signature

**Output:** a pair \(\Delta_i\) of dictionary and public key, random string \(r_i\), and signature \(\sigma_i\)

1: initialize a dictionary \(Dict_i^{CBF}\)
2: \(r_i \leftarrow \{0, 1\}^\lambda\)
3: for all \(ID_i \in Dict_i^{BF}\) do
4: initialize a customized Bloom filter \(cbf_{ID_i}\)
5: \(bf_{ID_i} \leftarrow Dict_i^{BF}[ID_i]\)
6: for all integer \(pos \in [1,|bf_{ID_i}|]\) s. t. \(bf_{ID_i}[pos]=1\) do
7: \(\xi_1 \leftarrow F(K, pos)\)
8: \(\xi_2 \leftarrow F(r_i, \xi_1)\)
9: \(cbf_{ID_i.add(\xi_2)}\)
10: \(Dict_i^{CBF}[ID_i] \leftarrow cbf_{ID_i}\)
11: \(hr \leftarrow H_0(PK_{i,1}, r_i)\)
12: \(\sigma_i \leftarrow Sign(SK_i, hr)\)
13: \(\Delta_i = (Dict_i^{CBF}, PK_{i,1}, r_i, \sigma_i)\)
14: return \(\Delta_i, \sigma_i\)

For each pseudonym \(ID_i\) in the dictionary \(Dict_i^{BF}\), hospital \(i\) connects the entry \(Dict_i^{BF}[ID_i]\) to the corresponding Bloom filter \(bf_{ID_i}\) that is constructed using the genome of patient with pseudonym \(ID_i\). If the value of a position \(pos\) in the Bloom filter \(bf_{ID_i}[bf_{ID_i}[pos]]=1\) (non-zero) then that position is extracted and input into the PRF \(F\) with the secret key \(K\). However, this way, the CSP may understand if a patient exists in different hospitals’ datasets and obtain more information about the patient. To avoid this, the hospital selects a random string \(r_i\) and invokes \(F\) with the inputs \(r_i\) and previous outcome of \(F\). Since \(r_i\) is a random string, the newly generated result is indistinguishable from a random input. The output is added into a customized Bloom filter \(cbf_{ID_i}\). Once all the non-zero elements of the Bloom filter \(bf_{ID_i}\) are mapped into the customized Bloom filter \(cbf_{ID_i}\), the pair \((ID_i, cbf_{ID_i})\) is added into the dictionary \(Dict_i^{CBF}\). This process is also shown in Fig. 6. To verify the authenticity of the index, hospital \(i\) digitally signs the hash of the concatenation of \(PK_{i,1}\) and \(r_i\) by using \(SK_i\). The output of the algorithm is a tuple \((\Delta_i, r_i, \sigma_i)\), where \(\sigma_i\) is
the signature, \( r_i \) is the random string applied to build the index, and \( \Delta_i \) consists of \( \text{Dict}_i^{AS} \) and \( \text{PK}_i \).

### 5.2.3 Data Encryption (at the Hospital) - SNPEncrypt and ASIEncrypt

Data encryption consists of two parts. The first part is the encryption of the genome (i.e., SNPs) and the second part is the ASI encryption. For genome encryption, we propose SNPEncrypt which utilizes the AES encryption algorithm. The input of the algorithm is the secret key \( K_i \) and a set \( \text{S}_{i,SNP} \) of SNPs stored at hospital \( i \). The output is a set \( \text{C}_{i,SNP} \) of encrypted genomes.

The ASI encryption algorithm at hospital \( i \) includes two rounds of AES encryption with two different secret keys (as also shown in Fig. 7). In the first round, the secret key \( K_{\beta_i} \) is used and in the second round, a secret key \( K_{\gamma_i} \) is randomly selected from the group \( G_T \) and is only shared with the approved clients. \( K_{\gamma_i} \) is encrypted using CPABE, which enables the access policy for the ASI. We describe the ASIEncrypt algorithm in the following. The details of the algorithm are given in Algorithm 2.

**Algorithm 2. ASIEncrypt**

**Input:** a secret key \( K_i \) of hospital \( i \), a secret key \( K_{\beta_i} \), and a dictionary \( \text{Dict}_i^{AS} \) storing ASI plaintext

**Output:** dictionary \( \text{Dict}_i^{C} \) storing ASI ciphertext

1. initialize a dictionary \( \text{Dict}_i^{AS} \)
2. for all \( \text{ID}_i \) in \( \text{Dict}_i^{AS} \) do
3. \( \text{C}_{\text{ID}_i} \leftarrow \phi \)
4. for all (ASI, \( \text{S}_{\text{ID}_i,SNP}^{\text{AS}} \)) in \( \text{Dict}_i^{AS}[[\text{ID}_i]] \) do
5. \( \tau \leftarrow \{0,1\}^{\lambda} \)
6. \( \theta \leftarrow \phi \)
7. for all (SNP.ID, SNP.val) in \( \text{S}_{\text{ID}_i,SNP} \) do
8. \( v \leftarrow \text{SNP.ID} \odot \text{SNP.val} \)
9. \( h \leftarrow H_2(\tau, e(H_1(v), g_2)^{1/k_i}) \)
10. \( \theta \leftarrow \theta \cup h \)
11. \( C_1 \leftarrow \text{AES.Enc}(K_{\beta_i}, \text{ASI}) \)
12. \( K_{\gamma_i} \leftarrow G_T \)
13. \( C_2 \leftarrow \text{AES.Enc}(K_{\gamma_i}, C_1) \)
14. \( A \leftarrow \text{CPABE.Enc}(\theta) \)
15. \( C_3 \leftarrow \text{CPABE.Enc}(\text{PK}_{i,2}, K_{\gamma_i}, A) \)
16. \( \text{C}_{\text{ID}_i} \leftarrow \text{C}_{\text{ID}_i} \cup (C_1, C_2, \tau) \)
17. \( \text{Dict}_i^{C}[[\text{ID}_i]] \leftarrow \text{C}_{\text{ID}_i} \)
18. return \( \text{Dict}_i^{C} \)

The input of the algorithm includes two keys \( K_i \) and \( K_{\beta_i} \) and a dictionary \( \text{Dict}_i^{AS} \). Each item in the dictionary consists of two components. The first component is the pseudonym of a patient (ID), and the second component is a list of ASIs belonging to the patient. For each pseudonym \( ID_i \) in the dictionary \( \text{Dict}_i^{AS} \), hospital \( i \) conducts the following operations. For each pair of \( \text{ASI}_k \) and \( \text{S}_{\text{ID}_i,SNP} \) inside the \( \text{Dict}_i^{AS}[[\text{ID}_i]] \), the hospital executes following four steps. First, for each SNP in set \( \text{S}_{\text{ID}_i,SNP} \) the hospital calls the hash function \( H_2 \) with a randomly selected value \( \tau \) and bilinear mapping \( e(H_1(v), g_2)^{1/k_i} \), where \( v \) is the concatenation of SNP.ID and its corresponding value SNP.val and \( g_2 \) is the generator of group \( G_2 \). The random value \( \tau \) enables the hash result \( h \) to be indistinguishable from a random string. The result \( h \) is added into a set \( \theta \). Second, the AES encryption algorithm (AES.Enc) is called to encrypt ASI with input key \( K_{\beta_i} \) and it outputs the ciphertext \( C_1 \). Third, a key \( K_{\gamma_i} \) is chosen from \( G_T \) and AES.Enc is called again to encrypt \( C_1 \) with \( K_{\gamma_i} \), resulting in ciphertext \( C_2 \). Fourth, the secret key \( K_{\gamma_i} \) is encrypted using CPABE with access structure \( A \) built from the set \( \theta \).

Specifically, all the elements inside \( \theta \) are considered as attributes of the access policy. With input of \( \text{PK}_{i,2} \), \( K_{\gamma_i} \), and \( A \), the algorithm \( \text{CPABE.Enc} \) outputs ciphertext \( C_3 \), which together with \( C_2 \) and \( \tau \) is added into a ciphertext set \( \text{C}_{\text{ID}_i} \). After all the ASIs of \( \text{Dict}_i^{AS}[[\text{ID}_i]] \) are encrypted, the pair \((\text{ID}_i, \text{C}_{\text{ID}_i})\) is inserted into a dictionary \( \text{Dict}_i^{C} \). Once this operation is done for all the patients (i.e., all the IDs have been processed), the algorithm outputs the dictionary \( \text{Dict}_i^{C} \).

### 5.2.4 Hierarchical Index Generation (at the CSP) - HierarchicalIndexGen

We develop a hierarchical clustering algorithm to cluster the Bloom filters representing the genome sequences into hierarchical clusters. Also, we develop a hierarchical index structure to index all the hierarchical clusters with small memory requirement. The CSP builds hierarchical index based on the received (unclustered) index from each hospital \( i \). The hierarchical index allows to search the target patient efficiently. Fig. 8 illustrates the hierarchical index structure and construction of hierarchical customized Bloom filters. The details of the algorithm are also given in Algorithm 3.
Algorithm 3. HierarchicalIndexGen

Input: unclustered index \( \Delta \)
Output: hierarchical index \( \Delta^H \)
1: \((\text{Dict}^{CBF}_i, PK_{i,1}, r_i) \rightarrow \Delta \)
2: \((Tr_i, \text{Dict}^H_i) \leftarrow HC(\text{Dict}^{CBF}_i)\)
3: \(\Delta^H \leftarrow (Tr_i, \text{Dict}^H_i, PK_{i,1}, r_i)\)
4: return \(\Delta^H\)

The key part of the hierarchical clustering algorithm is setting the similarity metric, as it determines the quality of clustering. Instead of using traditional euclidean distance as the similarity metric, in which one of the dimensions may be relatively large and may overpower the other dimensions, we choose the cosine similarity. Thus, in our protocol, the similarity metric is calculated as \(\text{sim}(cbf_{ID_i}, cbf_{ID_j}) = \frac{cbf_{ID_i} \times cbf_{ID_j}}{|cbf_{ID_i}| \times |cbf_{ID_j}|}\), where \(cbf_{ID_i}\) and \(cbf_{ID_j}\) are two customized Bloom filters for patients \(ID_i\) and \(ID_j\), respectively. Also, \(|cbf_{ID_i}|\) and \(|cbf_{ID_j}|\) represent the lengths of \(cbf_{ID_i}\) and \(cbf_{ID_j}\). The inner product of Bloom filters \(cbf_{ID_i}\) and \(cbf_{ID_j}\) is equal to the sum of bitwise AND of \(cbf_{ID_i}\) and \(cbf_{ID_j}\) since each element of a Bloom filter is either 0 or 1. The efficiency of computing the similarity score is enhanced by the bitwise operation.

We use a similarity matrix to keep the pairwise similarity values between different Bloom filters representing genome sequences. Given the similarity matrix, the pairwise distances (i.e., 1-similarity value) can be easily calculated. Then, the multidimensional scaling (MDS) algorithm [34] is invoked to compute the relative positions of genome sequences. Based on the relative positions, the classic hierarchical clustering algorithm is applied. For this, we use the classic Ward variance minimization algorithm [35].

We describe the HierarchicalIndexGen algorithm below.
The input of the algorithm is \( \Delta \) consisting of \( \text{Dict}^{CBF}_i \), \( PK_{i,1} \), and \( r_i \). The CSP first extracts the dictionary \( \text{Dict}^{CBF}_i \) from \( \Delta \). Then, the hierarchical clustering algorithm is called with the input \( \text{Dict}^{CBF}_i \) . The outcome consists of a tree structure \( Tr_i \) and a new dictionary \( \text{Dict}^H_i \). Finally, the tree structure \( Tr_i \), dictionary \( \text{Dict}^H_i \), public key \( PK_{i,1} \), and \( r_i \) are gathered into a new tuple \( \Delta^H \).

In Fig. 8, we provide a toy example illustrating index structure and index construction process of hierarchical customized Bloom filters. We show 4 patients, 7 customized Bloom filters, and 4 standard Bloom filters. The customized Bloom filters, \( cbf_{ID_1}, cbf_{ID_2}, cbf_{ID_3}, cbf_{ID_4} \) are constructed based on the standard Bloom filters (indirectly from patients' genomic data) by the hospital and sent to the CSP. The remaining customized Bloom filters, \( cbf_{ID_5}, cbf_{ID_6}, cbf_{ID_7} \) are generated by the CSP according to the tree structure, where \( cbf_{ID_1} = cbf_{ID_1} \text{AND} cbf_{ID_1} \), \( cbf_{ID_6} = cbf_{ID_6} \text{AND} cbf_{ID_5} \), and \( cbf_{ID_7} = cbf_{ID_7} \text{AND} cbf_{ID_6} \) (represents bitwise OR operation).

5.2.5 Index Merging (at the CSP) - IndexMerge

The index merging algorithm is invoked by the CSP once a high number of indices are received from different hospitals. The goal is to reconstruct an efficient index to replace all the stored indices without any loss in terms of utility and privacy.

Algorithm 4. IndexMerge

Input: a set of indices \( I_{ind} \)
Output: a merged index \( \phi \)
1: \( \gamma \rightarrow \phi \)
2: for all \( ind \in I_{ind} \)
3: \( (Tr_i, \text{Dict}^{CBF}_i, PK_{i,1}, r_i) \rightarrow ind \)
4: \( \gamma \rightarrow \gamma \cup (PK_{i,1}, r_i) \)
5: for all leaf node \( n \in Tr_i \)
6: \( \text{Dict}_{temp}[n.ID \circ r_i] \leftrightarrow \text{Dict}^{CBF}[n.ID] \)
7: \( (Tr_M, \text{Dict}^{CBF}_M) \leftrightarrow HC(\text{Dict}_{temp}) \)
8: return \( \phi \)

We describe the IndexMerge algorithm in the following. The details of the algorithm are also given in Algorithm 4. The CSP first initializes a temporary dictionary \( \text{Dict}_{temp} \) and a set \( \gamma \). \( \text{Dict}_{temp} \) is applied to store the pair of pseudonym and corresponding customized Bloom filter. The set \( \gamma \) is used to store pairs including a public key and the corresponding random string. For each index, the CSP resolves it into a tree \( Tr_i \), a dictionary \( \text{Dict}^{CBF}_i \), a public key \( PK_{i,1} \), and a string \( r_i \). For all the indices are processed, the CSP runs hierarchical clustering algorithm HC over the \( \text{Dict}_{temp} \) and outputs the new tree structure \( Tr_M \) and dictionary \( \text{Dict}^{CBF}_M \). The output of the algorithm is a merged index consisting of \( \gamma, Tr_M, \) and \( \text{Dict}^{CBF}_M \).

An alternative faster approach for IndexMerge is to merge the roots of the hierarchical indices instead of merging all the leaves from scratch. In this way, in the new hierarchical index, each leaf becomes a root of the original hierarchical index. The advantage of this fast approach is the reduced time to build the new index. Using the faster approach, the time complexity of IndexMerge decreases from \( O(N \log N) \) to \( O(\log n) \), where \( N \) is the total number of leaves and \( n \) is the total number of roots. The disadvantage is that the new index does not precisely cluster all the similar nodes into a cluster across hospitals. This may result in increased search time since the time complexity of search is determined by the search path from the root of the index to all the matching leaves. We implement and evaluate the performance of this faster approach in Section 7.

5.3 Client Authorization

Upon a hospital \( i \) receives an authorization request from a client, the hospital makes a decision on whether to allow its genomic data to be accessed or not. If the hospital approves the request, a shared key is generated and sent to the CSP.
and a success message is sent back to the client along with the keys $K$ and $K_i$. Otherwise, a failure message is sent to the client. The shared key is used by the CSP to transform the client's query token into a searchable token over ASI. We describe the shared key generation algorithm in the following. The details of the algorithm are also given in Algorithm 5. The input of the algorithm includes two keys $K$ and $K_i$, and a set $S_{i,SNP}$ of SNPs. $K_i$ and $S_{i,SNP}$ are from hospital $i$, while $K$ is from client $c$. $S_{SNP}$ is the set of SNPs that the hospital allows the client to access. The hospital generates the first part of the shared key ($\delta_i$) by computing $g_2^{K_i/K_i}$. Then, for each SNP in $S_{i,SNP}$, the hospital executes the following two procedures. First, the hash function $H_1$ is invoked with the input SNP and the hash result is raised to the power $1/K_i$. Second, the previous outcome is added into a customized Bloom filter $cbf_i$. The final output is the shared key consisting of two parts: $\delta_i$ and $cbf_i$.

**Algorithm 5. SharedKeyGen**

**Input:** a secret key $K_i$ of hospital $i$, a secret key $K_c$ of a client, a set $S_{i,SNP}$ of authorized SNPs

**Output:** shared key $(\delta_i, cbf_i)$

1. initialize a customized Bloom filter $cbf_i$
2. $\delta_i = g_2^{K_i/K_i}$
3. **for all** $(SNP.ID, SNP.val) \in S_{i,SNP}$
4. $v \leftarrow SNP.ID \oplus SNP.val$
5. $\ell_v \leftarrow H_1(v)^{1/K_i}$
6. $cbf_i.add(\ell_v)$
7. **return** $(\delta_i, cbf_i)$

---

### 5.4 Query Processing

As shown in Fig. 4, query processing includes: query generation (QueryGen and TokenGen), search (Search, SearchOverMergedIndex, TokenAdjust, ASISearch), and decryption (ASIDecrypt).

#### 5.4.1 Query Generation - QueryGen and TokenGen

Query generation is executed by a client. It consists of two algorithms, QueryGen and TokenGen. The outcome of the QueryGen is used to search over indices of target pseudonyms, while the outcome of the TokenGen is used to search over the encrypted ASIs.

**Algorithm 6. QueryGen**

**Input:** a secret key $K$ of PRF, a set $S_{c,SNP}$ of SNPs, threshold $\varepsilon_c$, top $k_c$, and signature $\sigma_i$

**Output:** query $Q$

1. initialize a Bloom filter $bf$
2. **for all** $(SNP.ID, SNP.val) \in S_{c,SNP}$
3. $bf.add(SNP.ID \oplus SNP.val)$
4. $E \leftarrow \phi$
5. **for all** integer $pos \in [1, |bf|]$ such that $bf[pos] = 1$
6. $\xi_i \leftarrow F(K, pos)$
7. $count \leftarrow count+1$
8. $E \leftarrow E \cup \xi_i$
9. **return** $Q \leftarrow (E, \varepsilon_c, k_c, \sigma_i)$

**QueryGen** algorithm is detailed in Algorithm 6. The input of the algorithm for a client $c$ consists of the secret key $K$, a set $S_{c,SNP}$ of SNPs, a threshold $\varepsilon_c$, representing the minimum cosine similarity for a successful search, and a parameter $k_c$ to specify the threshold for the maximum number of retrieved pseudonyms. Each pair of $SNP.ID$ and $SNP.val$ inside $S_{c,SNP}$ is concatenated and added into a Bloom filter $bf_c$. The procedure is same as the process in index generation (as in Section 5.2.2). For each non-zero element inside the $bf_c$, the position $pos$ is extracted and computed by calling $F$ using the secret key $K$ as the input. The output is added into a set $E$. The final form of the output is a tuple $(E, \varepsilon_c, k_c, \sigma_i)$, where signature $\sigma_i$ is generated by the hospital $i$ and sent to an approved client.

**Algorithm 7. TokenGen**

**Input:** a secret key $K_c$, a set $S_{c,SNP}$ of SNPs

**Output:** token $TK$

1. $TK \leftarrow \phi$
2. **for all** $(SNP.ID, SNP.val) \in S_{c,SNP}$
3. $v \leftarrow SNP.ID \oplus SNP.val$
4. $tk \leftarrow H_1(v)^{1/K_c}$
5. $TK \leftarrow TK \cup tk$
6. **return** $TK$

The details of the TokenGen algorithm are given in Algorithm 7. The input of the TokenGen algorithm includes a secret key $K_c$ and a set $S_{c,SNP}$ of SNPs. For each SNP inside $S_{c,SNP}$, a hash function $H_1$ is called and the hash result is raised to the power $1/K_c$ for encryption and future token adjustment. Each outcome is collected into a set $TK$. Finally, the algorithm outputs $TK$.

Eventually, the query sent to the CSP consists of the outputs of algorithms QueryGen and TokenGen.

#### 5.4.2 Search Over a Single Index - Search

The search algorithm is run by the CSP. For clarity of the presentation, we first consider a scenario in which the search algorithm runs over a single index (belonging to a single hospital). The search algorithm only uses the first part $Q$ of the query (encrypted input SNPs $E$) to traverse the index from the root to the leaves considering the minimum similarity threshold $\varepsilon_c$. The details of the search algorithm are as follows. They are also shown in Algorithm 8.

The CSP receives a query $Q$ from a client $c$ and an index $(\Delta_i, r_i)$ from a hospital $i$. The CSP first resolves $\Delta_i$ into a tree $T_{\Delta_i}$, a dictionary $Dict_{\Delta_i}$ (which consists of pairs of patient pseudonym and corresponding customized Bloom filter), and a public key $PK_{\Delta_i}$. Query $Q$ is resolved into a set $E$, a signature $\sigma_i$, and two threshold values $\varepsilon_c$ and $k_c$. Then, the CSP builds a dictionary $Dict$ with only $k_c$ entries. The dictionary $Dict$ stores pairs including (i) patient pseudonym and (ii) similarity score between corresponding patient’s customized Bloom filter and the queried genome’s customized Bloom filter.

Afterwards, the CSP verifies $\sigma_i$ by running the verification function $verify$ with the inputs $PK_{\Delta_i}$, $\sigma_i$, and $H(PK_{\Delta_i}, r_i)$. If the verification fails, the process is terminated. Otherwise, the CSP continues to execute the following procedures. For each element $\xi_i$ in the set $E$, the CSP runs $F$ with the inputs $\xi_i$ and $r_i$. The outcome $\xi'_i$ is added into a customized Bloom filter $cbf_{c,i}$. After completing the
above process, the CSP reads the root \((\text{root}_i)\) from \(Tr_i\) and pushes it into the queue \(qu\).

**Algorithm 8. Search**

**Input:** hierarchical index \(\Delta^H\), random string \(r_i\), and query 
\[ Q = (E, e_c, k_c, \sigma_c) \]

**Output:** a dictionary \(\text{Dict}\) with \(k_c\) entries

1: \((Tr_i, \text{Dict}_i^H, \text{PK}_{i,1}) \leftarrow \Delta^H\)
2: \(\text{Dict} = \phi\)
3: if \(\text{verify}(\text{PK}_{i,1}, \sigma_i, H_0(\text{PK}_{i,1}, r_i)) = \text{False}\) then
   4: return None
5: initialize a queue \(qu\) and a customized Bloom filter \(\text{cbf}_{e}\)
6: for all \(z_i \in E\) do
7: \(\zeta_2 \leftarrow F(r_i, z_i)\)
8: \(\text{cbf}_{e}.\text{add}(\zeta_2)\)
9: \(\text{root}_i \leftarrow Tr_i\)
10: \(qu.\text{push}(\text{root}_i)\)
11: while \(qu\) is not empty do
12: \(n \leftarrow qu.\text{pop}()\)
13: \(\text{sim} \leftarrow \frac{\text{Dict}_i^H[n.ID_1].\text{cbf}_{e}}{\text{Dict}_i^H[n.ID_1].||\text{cbf}_{e}|}\)
14: if \(\text{sim} \geq e_c\) then
15: if \(n\) is a leaf then
16: \(\text{Dict} \leftarrow \text{Insert}(\text{Dict}, n.ID_i, \text{sim})\)
17: else
18: if \(n\) has a left child then
19: \(\text{leftchild} \leftarrow \text{the left child of the } n\)
20: \(qu.\text{push}()\text{leftchild}\)
21: if \(n\) has a right child then
22: \(\text{rightchild} \leftarrow \text{the right child of the } n\)
23: \(qu.\text{push}()\text{rightchild}\)
24: return \(\text{Dict}\)

Following steps are recursively executed until the \(qu\) is empty. First, a node \((n)\) is popped out from the \(qu\). Second, the similarity score \(\text{sim}\) is computed by using the cosine similarity between \(\text{Dict}[n.ID]\) and \(\text{cbf}_{e}\). If \(\text{sim}\) is less than \(e_c\), then the next step is skipped and step one is invoked again. If \(\text{sim}\) is greater than or equal to \(e_c\), the property of \(n\) is checked. If \(n\) is a leaf, we call the insert function (\(\text{Insert}\)) with the inputs \(\text{Dict}_i, n.ID_i, \text{and sim}\). The details of \(\text{Insert}\) are given in Algorithm 9. The purpose of \(\text{Insert}\) function is to insert the pair \((ID_i, sim)\) into \(\text{Dict}\) if \(\text{Dict}\) is not full or there exits a pair that has smaller similarity score compared to the current node. If \(n\) is not a leaf and there exists a left child \((\text{leftchild})\), then \(\text{leftchild}\) is pushed into \(qu\). If its right child \((\text{rightchild})\) exits, then \(\text{rightchild}\) is pushed into \(qu\). After this iteration is completed, the CSP outputs the final result \(\text{Dict}\).

**Algorithm 9. Insert**

**Input:** dictionary \(\text{Dict}\), pseudonym \(ID_i\), similarity score \(\text{sim}\)

**Output:** dictionary \(\text{Dict}\)

1: if \(\text{len}() < k_c\) then
2: \(\text{Dict}(ID_i) \leftarrow \text{sim}\)
3: return \(\text{Dict}\)
4: \(\text{find the pair (ID}_i, \text{sim}_i)\) such that
5: \(\text{sim}_i = \min(\text{ID}_i, \text{sim}_i)\) \(\text{Dict}\)
6: if \(\text{sim}_i < \text{sim}\) then
7: \(\text{delete the pair (ID}_i, \text{Dict}_i(ID_i))\) from \(\text{Dict}\)
8: \(\text{Dict}(ID_i) \leftarrow \text{sim}\)
9: return \(\text{Dict}\)

**5.4.3 Search Over a Merged Index - SearchOverMergedIndex**

Compared to searching over a single index, searching over a merged index mainly differs in two aspects. First, the CSP verifies all the signatures submitted by the client instead of a single one (to recognize which hospitals in the merged index authorize the search). Second, each random string attached to an authorized index is used to generate a customized Bloom filter based on the submitted query.

**Algorithm 10. SearchOverMergedIndex**

**Input:** Merged index \(\varphi\), query \(Q = (E, \sigma_c, e_c, k_c)\)

**Output:** dictionary \(\text{Dict}\) with \(k_c\) entries

1: \((T, Tr_M, \text{Dict}^{CBF}_M) \leftarrow \varphi\)
2: initialize dictionary \(\text{Dict}\) with \(k_c\) entries
3: initialize a customized Bloom filter \(\text{cbf}_{e}\)
4: \(\text{sim} \leftarrow \phi\)
5: for all \((\text{PK}_{i,1}, r_i) \in \mathcal{Y}\) do
6: for all \(\sigma_i \in \sigma_c\) do
7: if \(\text{verify}(\text{PK}_{i,1}, \sigma_i, H(\text{PK}_{i,1}, r_i)) = \text{False}\) then
8: \(\text{sim} \leftarrow \text{sim} \cup r_i\)
9: for all \(z_i \in E\) do
10: \(\zeta_2 \leftarrow F(r_i, z_i)\)
11: \(\text{cbf}_{e}.\text{add}(\zeta_2)\)
12: if \(\text{cbf}_{e} \neq \phi\) then
13: \(\text{root} \leftarrow Tr_M\)
14: initialize a queue \(qu\)
15: \(qu.\text{push}(\text{root})\)
16: while \(qu\) is not empty do
17: \(n \leftarrow qu.\text{pop}()\)
18: \(\text{sim} \leftarrow \frac{\text{Dict}^{CBF}_i[n.ID_1].\text{cbf}_{e}}{\text{Dict}^{CBF}_i[n.ID_1].||\text{cbf}_{e}|}\)
19: if \(\text{sim} \geq e_c\) then
20: if \(n\) is a leaf then
21: \(\text{Dict} \leftarrow \text{InsertConditionally}(\text{Dict}, n.ID_i, \text{sim})\)
22: else
23: if \(n\) has a left child then
24: \(\text{leftchild} \leftarrow \text{the left child of the } n\)
25: \(qu.\text{push}()\text{leftchild}\)
26: if \(n\) has a right child then
27: \(\text{rightchild} \leftarrow \text{the right child of the } n\)
28: \(qu.\text{push}()\text{rightchild}\)
29: return \(\text{Dict}\)

We describe the \text{SearchOverMergedIndex} algorithm in the following. The details of the algorithm are also given in Algorithm 10. The input of the \text{SearchOverMergedIndex} algorithm is a merged index \(\varphi\) and a query \(Q\) from a client \(e\). The query \(Q\) includes a set \(E\) of encrypted SNPs, a set \(\sigma_c\) of signatures, and two thresholds \(e_c\) and \(k_c\). The merged index is resolved into a tree \(\mathcal{Y}\), a tree structure \(Tr_M\), and a dictionary \(\text{Dict}^{CBF}_M\). For each signature \(\sigma_i \in \sigma_c\), the CSP verifies whether there exists a pair \((\text{PK}_{i,1}, r_i) \in \mathcal{Y}\) that matches \(\text{verify}(\text{PK}_{i,1}, \sigma_i, H(\text{PK}_{i,1}, r_i)) = \text{True}\). For each pair \((\text{PK}_{i,1}, r_i)\) that matches the verification, the random string \(r_i\) is extracted and input into \(F\) with each \(z_i\) in \(E\). The output \(\zeta_2\) of \(F\) is added into the customized Bloom filter \(\text{cbf}_{e}\). If the customized Bloom filter is empty, it means no submitted signature is valid and the algorithm returns None. Otherwise, \(\text{cbf}_{e}\) is used to search over the tree \(Tr_M\). The process is similar to algorithm \text{Search} in Section 5.4.2. The only difference is that the \text{Insert} function is replaced by...
InsertConditionally. Compared with Insert, the difference is that in the InsertConditionally algorithm, the input pseudonym \(ID_i\) is the concatenation of a real patient pseudonym and a random string that is required to be inside the authorized set. The extra operation is to verify the legitimacy of the record and to guarantee that all the records stored in \(Dict\) are authorized (details of InsertConditionally are given in Algorithm 11. Finally, the SearchOverMergeIndex algorithm outputs \(Dict\).

**Algorithm 11. InsertConditionally**

**Input:** dictionary \(Dict\), pseudonym \(ID_i\), random string set \(\mathbb{R}\), similarity score \(sim\)

**Output:** dictionary \(Dict\)

1: \((ID^*_i, r^*_i) \rightarrow ID_i\)
2: if \(r^*_i \in \mathbb{R}\) then
3: \(Dict \leftarrow \text{Insert}(Dict, ID^*_i, sim)\)
4: return \(Dict\)

Note that the merge operation does not break the correctness of the search output. However, since our proposed merge algorithm does not guarantee to generate a balanced tree, the efficiency of the search operation may slightly degrade (since the search time is based on the length of the path from the root of the index to the matching leaves). In Section 7, we compare the performance of the search operation on (i) a balanced tree and (ii) a merged tree using our proposed algorithm.

### 5.4.4 Token Adjustment - TokenAdjust

The token received from the client is not directly applicable for search over the ASI ciphertext. The CSP needs to use a shared key to transform the received token into an executable token (see Algorithm 12 for details). The input of the TokenAdjust algorithm consists of a set of tokens (TK) and a shared key \((\delta, cbf_i)\) from a hospital \(i\). For each token \(tk\) in TK, the membership evaluation is conducted over the customized Bloom filter \(cbf_i\). If \(cbf_i(tk) \neq \text{False}\), the bilinear mapping algorithm \(e\) is called with inputs \(tk\) and \(\delta\). The result is collected into a set \(TK^*\). Otherwise, the current round of \(tk\) is skipped. Finally, the algorithm outputs \(TK^*\), which can be used to search the target ASIs.

**Algorithm 12. TokenAdjust**

**Input:** a set TK of raw tokens, a shared key \((\delta, cbf_i)\)

**Output:** a set TK' of valid tokens

1: \(TK' \leftarrow \phi\)
2: for all \(tk \in TK\) such that \(cbf_i(tk) = \text{True}\)
3: \(tk' \leftarrow e(tk, \delta)\)
4: \(TK' \leftarrow TK' \cup tk'\)
5: return \(TK'\)

### 5.4.5 ASI Search - ASISearch

Given executable tokens (for search) and ciphertext, the CSP can proceed with ASI search. To identify an ASI, the tokens must match the access policy set on the ASI. The access policy is set in the granularity of SNPs. For example, ASIs related to the diagnosis and treatment of breast cancer are encrypted by considering the SNPs of BRCA gene as the required attributes. That is, these attributes are applied to construct a polynomial that outputs the value of a target parameter (secret key) when all the required attributes are satisfied.

**Algorithm 13. ASISearch**

**Input:** valid token TK', encrypted ASI data \(Diet^C\), a set \(\delta\) of pseudonyms, a master key \(MK_i\) of CPABE

**Output:** a set \(C_{C_1}\) of ASI ciphertext

1: \(C_{C_1} \leftarrow \phi\)
2: for all \(ID_i \in \delta\) do
3: for all \((C_3, C_2, \tau) \leftarrow \text{Diet}^C[|ID_i|]\) do
4: \(\vartheta \leftarrow \phi\)
5: for all \(tk \in TK'\) do
6: \(h \leftarrow H_2(t, tk)\)
7: \(\vartheta \leftarrow \vartheta \cup h\)
8: \(sk \leftarrow \text{CPABE.KeyGen}(MK_i, \vartheta)\)
9: if \(sk \neq \phi\) then
10: \(K_{\gamma_i} \leftarrow \text{CPABE.Decrypt}(sk, C_3)\)
11: if \(K_{\gamma_i} \neq \text{False}\) then
12: \(C_1 \leftarrow \text{AES.Decrypt}(K_{\gamma_i}, C_2)\)
13: \(C_{C_1} \leftarrow C_{C_1} \cup C_1\)
14: return \(C_{C_1}\)

The details of ASI search are also given in Algorithm 13. The input of the ASI search algorithm includes a set \(TK'\) of executable tokens, a set \(\delta\) of pseudonyms of patient records obtained from index search, a dictionary \(Diet^C\) containing pseudonym and ASI ciphertext pairs, and a master key \(MK_i\) of CPABE. For each pseudonym \(ID_i\) in \(\delta\), the entry \(\text{Diet}_{ASI}[|ID_i|]\) associates ciphertexts \(C_2, C_3\) and a random string \(\tau\). To decrypt ciphertext \(C_2\), the secret key \(sk\) of CPABE is generated as follows. For each token \(tk'\) in \(TK'\), the hash function \(H_2\) is called with the inputs \(\tau\) and \(tk'\). The result is gathered into a set \(\vartheta\). Then, the key generation algorithm \(\text{CPABE.KeyGen}\) is called with the master key \(MK_i\) of CPABE and \(\vartheta\). If the attribute set \(\vartheta\) does not match the access policy, the newly generated \(sk\) is null and following operations are skipped. Otherwise, the following procedures are executed to open the first layer of ciphertext \(C_2\). The decryption algorithm (\(\text{CPABE.Decrypt}\)) of CPABE is invoked with the inputs \(sk\) and \(C_3\). If the output symmetric key \(K_{\gamma_i}\) is not null, the decryption algorithm \(\text{AES.Decrypt}\) of AES is invoked with inputs \(K_{\gamma_i}\) and \(C_2\). The output \(C_1\) is collected into a set \(C_{C_1}\). Once all the elements inside \(\delta\) are accessed, the search algorithm outputs the set \(C_{C_1}\) that will be sent to the client.

### 5.4.6 ASI Decryption - ASIDecrypt

Upon receiving the search result, the client applies its secret key to decrypt the ciphertext of ASI. For each ciphertext of retrieved result \(C_{C_1}\), the decryption algorithm of AES is invoked to decrypt the ciphertext with the input secret key \(K_{\gamma_i}\) assigned by hospital \(i\). The plaintext ASI is gathered into a set \(S_{ASI}\). After all the ciphertext is decrypted, the algorithm outputs \(S_{ASI}\). The details of this operation are also given in Algorithm 14.
Algorithm 14. ASIDecrypt

Input: a secret key $K_i$ of symmetric encryption, a set $C_{C_i}$ of ASI ciphertext
Output: a set $A_{ASI}$ of plaintext of ASIs

1: $A_{ASI} = \emptyset$
2: for all $c \in C$ do
3:   $ASI \leftarrow AES.Decrypt(K_i, c)$
4:   $A_{ASI} \leftarrow A_{ASI} \cup ASI$
5: return $A_{ASI}$

6 PRIVACY ANALYSIS

In the following sections, we first provide a high level discussion about how our scheme achieves robustness in the presence of attacks discussed in Section 4.2 and then give formal privacy definition and proof.

6.1 Privacy Against the Ciphertext Analysis Attack

As discussed, we propose storing encrypted data and encrypted genome index at the CSP, and hence the CSP may try to infer sensitive information by analyzing the stored ciphertext. Encrypted data includes ciphertext for genome and ASI data. Since both genome and ASI data are encrypted using AES (which provides semantic security), their privacy is provided by the underlying cryptosystem. Additionally, encrypted index can also be analyzed by the CSP. Since the index is built through the hash function and PRF, the privacy of index depends on the strength of the building blocks of the proposed scheme.

6.2 Privacy Against the Query Analysis Attack

The input of the query is a set of pairs consisting of $SNP.ID$ and $SNP.val$. The first operation to generate the query is to map them into a Bloom filter by using hash functions. The second step is to use PRF to encrypt all the non-zero elements inside the Bloom filter. The privacy of the query relies on the one-wayness of the hash function and security of the PRF. Due to the property of hash function and PRF, the CSP is unable to recover the $SBP.ID$ and $SNP.val$ from the query.

6.3 Robustness Against the Illegitimate Access Attack

Access control consists of two phases. The first phase is to control the access to the genotype data. The second phase is to control the access to the ASI data. The first phase relies on two elements: the secret key $K$ assigned by the hospital for query generation and the signature from the hospital. Using these two elements, a valid query can be generated. In the second phase, access control is provided by a shared key $(\delta_i, cb_{f_j})$. The query is transformed to search over ASI if and only if a valid key is assigned by the hospital to the client and sent to the CSP. It is trivial to show that this access control mechanism provides robustness for the proposed scheme against the illegitimate access attack. In the following, we formally analyze the robustness of the proposed scheme against ciphertext analysis and query analysis attacks.

Following previous work [36], [37], [38], we consider the following as the allowed leaked information to the CSP (potential adversary) throughout the protocol: (i) size pattern, e.g., the number of patients in a hospital’s encrypted dataset, (ii) search pattern, e.g., the search path from the root to a leaf, (iii) access pattern, e.g., the frequency of record accesses, and (iv) query pattern, e.g., whether two queries include the same pattern. Additionally, we also assume cluster information of a dataset is obtainable by the CSP. This assumption does not increase extra leakage because the CSP is able to extract such information from access pattern leakage.

We observe that if there only exists a single semi-trusted server and the access pattern is not obfuscated, it is unavoidable to disclose search results to the server. Note that the previous work [10], [12], [13], did not adopt the single semi-honest model, and hence they do not have the disclosure of search results. We believe that it is reasonable to accept the disclosure of search result in the single semi-honest model as existing techniques of access pattern obfuscation, e.g., oblivious RAM (ORAM) [39], are still expensive and not practical for large datasets.

As will be discussed, our privacy goals are based on the strengths of the hash function, PRF, and AES. The allowed leakage does not violate our privacy goals since the revealed size pattern, search pattern, access pattern, and query pattern do not weaken the security of hash function, PRF, and AES. We provide the details of the leakage function in [33].

6.4 Privacy Definition

The privacy of the proposed scheme consists of two components. The first is the privacy of genomic data and the second is the privacy of ASI. Privacy of genomic data can be further split into index privacy and privacy of genome sequences. Genome sequences are encrypted using AES and they are not involved in the query processing. Thus, their privacy relies on the robustness of AES. Since AES encryption achieves semantic security (e.g., CBC and CTR modes [40]), the encrypted genome sequences are robust against chosen plaintext attacks. Similarly, the ASI information is protected by the AES encryption, which is also robust against chosen plaintext attacks.

Therefore, our privacy goal (for genomic data) is to protect SNP data from an adversary (CSP) that can analyze the values inside the customized Bloom filter (output of IndexGen), analyze encrypted genome data (output of SNPEncrypt), analyze encrypted query content (output of QueryGen), and conduct search operation (execute the Search algorithm). The CSP is also allowed to learn (i) the relative similarity between the records within a dataset (that is crucial for the CSP to run the heavy clustering algorithm), (ii) the number of records (it is unavoidable since all the data is stored on its platform, and (iii) the frequency of a query (the query is transformed by the CSP to search over different hospitals). Considering these, we formally represent this privacy goal for genomic data as the indistinguishability of two datasets $DB_{0,SNP}$ and $DB_{1,SNP}$ with the same index structure. In other words, we set the constraint on the selection of $DB_{0,SNP}$ and $DB_{1,SNP}$ to have exactly the same index structure. This also implies the same number of patients in $DB_{0,SNP}$ and $DB_{1,SNP}$. Under such a setting, we formulate a privacy game that captures the privacy of customized Bloom filter analysis, encrypted genome data analysis, query analysis, and search operation.

We formulate the privacy of genomic data as a game between a challenger and an adversary, which includes both

Authorized licensed use limited to: ULAKBIM UASL - Bilkent University. Downloaded on February 15,2023 at 07:48:43 UTC from IEEE Xplore. Restrictions apply.
privacy of index and genome sequence. First, the adversary selects two datasets $DB_{0,SNP}$ and $DB_{1,SNP}$ of SNPs. Considering what the adversary (e.g., CSP) is allowed to learn during the protocol (as discussed in Section 6.4), we set a constraint on the selection of $DB_{0,SNP}$ and $DB_{1,SNP}$ to have exactly the same index structure. This also implies the same number of patients in $DB_{0,SNP}$ and $DB_{1,SNP}$. Each item of dataset $DB_{i,SNP}$ $(i \in \{0, 1\})$ includes two components: patient pseudonym and corresponding SNPs of the patient. Then, the adversary sends the two datasets to the challenger. The adversary is allowed to send adaptive queries with constraint on the information leakage before making the final decision about which dataset is utilized by the challenger.

Let $\Pi_{SNP} = \{\text{Setup}, \text{IndexGen}, \text{QueryGen}, \text{IndexMerge}, \text{Search}, \text{SearchOverMergedIndex}\}$ be a set of algorithms of the proposed scheme that are related to genomic data privacy. For a probabilistic polynomial time (PPT) adversary $Adv$, the advantage function $ADV_{Adv}^{\Pi_{SNP}}$ is defined as follows. $ADV_{Adv}^{\Pi_{SNP}} = Pr(b^* = b) - \frac{1}{2}$, where $b$ and $b^*$ are defined in the following game which evaluates the probability of breaking the proposed scheme. We describe the key steps of the game between the challenger and the adversary below.

**Init.** The adversary $Adv$ submits two datasets $DB_{0,SNP}$ and $DB_{1,SNP}$ to the challenger with the same number of records and index structure.

**Setup.** The challenger generates the initial functions, parameters, and keys. Details can be found in Section 5.2.

**Phase 1.** The adversary is allowed to obtain the ciphertexts of genome, index, and query by adaptively submitting ciphertext requests, index requests, and query requests to the challenger.

**Ciphertext Request.** The adversary selects a dataset of genome sequence and submits it to the challenger to request its ciphertext. The selected dataset is not limited to $DB_{0,SNP}$ and $DB_{1,SNP}$.

**Index Request.** The adversary selects a dataset that is different from $DB_{0,SNP}$ and $DB_{1,SNP}$ and submits it to the challenger to request its index.

**Challenge.** The challenger randomly selects a bit $b$ from $\{0, 1\}$ and encrypts $DB_{b,SNP}$ to generate $C_{b,SNP}$. It also generates encrypted index $C_{ind_b}$ and sends $C_{b,SNP}$ and $C_{ind_b}$ to the adversary.

**Phase 2.** The adversary adaptively submits query request $Q$ in addition to the ciphertext and index requests described in Phase 1.

**Query Request.** The adversary selects target SNPs, sets two thresholds, attaches the signature of target index, and sends them to the challenger for asking the query with constraint $L(S_{b,SNP}, Q, C_{ind_b}) = L(S_{b,SNP}, Q, C_{ind_b})$.

**Guess.** The adversary $Adv$ outputs $b^*$ as a guess of $b$.

The privacy of the genomic data is preserved against selective chosen plaintext attack if in the above scheme $\Pi_{SNP}$ the adversary $Adv$ has negligible advantage. Explicitly, the advantage function $ADV_{Adv}^{\Pi_{SNP}}$ should be a negligible function in parameter $\lambda$.

### 6.5 Privacy Proof

We first describe the main conclusion about the privacy of genome data and then give the corresponding proof.

Theorem 1. Let $\Pi_{SNP} = \{\text{Setup}, \text{IndexGen}, \text{QueryGen}, \text{IndexMerge}, \text{Search}, \text{SearchOverMergedIndex}\}$ be a set of algorithms of the proposed scheme that are related to genomic data privacy. The scheme $\Pi_{SNP}$ is privacy-preserving if no PPT adversary can distinguish $DB_{0,SNP}$ from $DB_{1,SNP}$ with non-negligible advantage at the end of the defined game.

It is trivial to verify that the above theorem is consistent with the privacy definition of genomic data. This leads us to the following theorem.

Theorem 2. A PPT adversary cannot distinguish the view of $DB_{0,SNP}$ from the view of $DB_{1,SNP}$ in the defined privacy game if the applied AES encryption mode (e.g., CBC and CTR) is semantically-secure, and PRF is indistinguishable from a random function.

If above theorem holds, a PPT adversary is not able to learn extra information from ciphertext and query analysis attacks. The formal proof of theory 2 is presented below.

**Proof.** We prove Theorem 2 by assuming if there exists an adversary $Adv$ that can distinguish the two views of $DB_{0,SNP}$ and $DB_{1,SNP}$, then there exists a simulator $B$ that can break either the semantic security of AES encryption or the randomness of PRF. In the following, we follow the previously defined game with a simulator $B$ trying to break the proposed scheme.

**Init.** A simulator $B$ selects and submits two datasets of genome sequences $DB_{0,SNP}^*$ and $DB_{1,SNP}^*$ to the adversary $Adv$ and challenger with same number of records and index structure.

**Setup.** The challenger runs Setup to set initial parameters and functions.

**Phase 1.** The adversary adaptively submits one of the following requests to the simulator $B$.

**Ciphertexts Request.** $B$ directly submits the dataset from adversary to the challenger and sends the ciphertext from the challenger to the adversary. The dataset is not limited to $DB_{0,SNP}^*$ and $DB_{1,SNP}^*$.

**Index Request.** The simulator $B$ sends the newly generated ciphertext and index to the simulator $B$ and challenges with same number of records and index structure.

**Challenge.** The simulator $B$ directly sends the dataset to the simulator $B$ and sends the returned index from the challenger to the adversary $Adv$. The submitted dataset is not limited to $DB_{0,SNP}^*$ and $DB_{1,SNP}^*$.

**Phase 2.** The adversary submits the following query request to simulator $B$ in addition to repeating Phase 1.

**Query Request.** Simulator $B$ uploads the submitted query request to the simulator $B$ and sends the result back to the adversary.

**Guess.** The adversary outputs its guess $b^*$ to the simulator $B$ and the simulator $B$ outputs the same guess.

According to the initial assumption, the adversary $Adv$ has significant advantage in breaking the defined experiment. In the proof, the described experiment strictly follows the defined experiment, and hence the simulator $B$ has significant advantage in guessing the
correct answer. Thus, the simulator B can distinguish the ciphertext $C_{0, \text{SNP}}^*$ and index $C_{\text{ind}0}$ from $C_{1, \text{SNP}}^*$ and index $C_{\text{ind}1}^*$ with significant advantage in the experiment. Since $C_{0, \text{SNP}}^*$ is the ciphertext obtained from AES encryption and index $C_{\text{ind}0}$ is the output of a PRF, the simulator B successfully breaks one of them if the simulator B cannot learn significant information from the ciphertext request, index request, and query request.

First, we analyse the ciphertext request. Since each ciphertext is generated as a result of AES encryption, the security is guaranteed by the robustness of AES. According to the assumption, the adopted AES achieves semantic security and the simulator B cannot learn significant information from the ciphertext. Second, the challenged datasets have the same index structure. Moreover, each index is randomized by a random string and hence the simulator B cannot correlate any two different indices. Third, the query request is constrained by the leakage function. The request will be released if and only if the request causes the same leakage of two challenged datasets. As the leakage is the same, the simulator B cannot learn significant information to distinguish the challenged ciphertext and index. Based on the above analysis, we can conclude that in the above experiment, the simulator B should break either the AES encryption or the randomness of PRF.

7 Evaluation

In this section, we evaluate the efficiency and scalability of the proposed scheme. Since the memory mapping technique [41], [42], [43] is widely applied in our implementation of the proposed scheme (e.g., small part of the index is stored in memory and the remaining is stored on the disk), a stable memory address is required to measure the memory usage for different scenarios. Due to this requirement, we first conducted experiments on a single machine to show the efficiency of the proposed scheme and to analyze the run-time under different scenarios. Due to the resource constraints of a single machine, we then turned to Amazon EC2 platform [44] for running large-scale experiments to show the scalability of the proposed scheme. In all experiments, the length of security parameter of RSA signature is set to 3096 bits. We use RSA only for the digital signature and it is only computed once per query by the data owner. Thus, it has a negligible effect on the overall performance. Index is built with a key ($K$) of size 256 bits and the genomic data is encrypted using AES with a key ($K_{\text{AES}}$) of size 256 bits. The asymmetric curve used in bilinear mapping is set to MNT224 and the symmetric curve applied in CPABE is set to SS512. The bilinear mapping is applied when the authorization protocol runs, while the CPABE is called for the ASI encryption. In our implementation, both MNT224 and SS512 provide 90 bits of security. The capacity of Bloom filter is set to $2^{21}$, and maximum false positive rate of the Bloom filter is set to 0.01. Also, in all experiments, we generated each query by setting the threshold for the minimum number of matching SNPs ($\epsilon$) to 90 percent of the total input SNPs and the threshold for the number of retrieved results ($k_c$) to 5. In addition, the number of SNPs in a query is equal to the number of SNPs per patient, if not specified otherwise. The client is assumed to be authorized to access all the hospitals’ data. Finally, we ran each experiment 10 times and report the average performance.

7.1 Experiments on a Single Local Machine

We ran the single machine experiments using a computer with Ubuntu system, i7 CPU, 32GB RAM, and 500GB hard disk.

7.1.1 Data Model

We used the rsnp tool [45] to obtain all the raw patient files from the publicly available OpenSNP dataset [46]. The whole dataset includes 3477 individuals and its plaintext size is 55GB. We first converted the raw patient files into VCF format using an open source software called personal-genome-analysis [47]. Eventually, we ended up with 2850 valid VCF files. For the affiliated sensitive information (ASI), we also used the OpenSNP dataset. In total, we collected 7388 ASIs and we randomly assigned them to the patients in varying numbers. The number of SNPs associated to an ASI varies from 20 to 2000.

7.1.2 Results

In our Bloom filter settings, the false positive rate is 1 percent for an input size of 2 million SNPs. When the input size increases to 3 million SNPs, the false positive rate increases to 6 percent. However, in our experiments, we did not observe such a high accuracy loss. The reason is that in the dataset few patient records had 3 million SNPs. Most patient records had around 2 million SNPs, so that the precision is at least 99 percent in all experiments.

We first evaluated the proposed mechanism when a hospital has 10, 100, 1000, and 2850 patients and each patient has 20 SNPs. The results are shown in Table 2. We observed that the time cost of QueryGen is constant, while the time costs of IndexGen and Search increases linearly with the increasing number of patients. Furthermore, the growth rate of the time cost of HierarchicalIndexGen is approximately equal to the square of the growth rate of the patient records. In addition, the memory and disk storage requirements increase linearly with the number of patients.

Then, we considered the scenario that includes a hospital with 2850 patients and each patient having 200, 2000, and 3350221 SNPs (whole sequence), respectively. The results are shown in Table 3. We observed that the time costs of IndexGen and QueryGen algorithms increase almost linearly with the increasing number of SNPs per patient. We also observed that the time costs of Search and HierarchicalIndexGen algorithms do not strongly correlate to the number of SNPs per patient; there is only slight increase in time cost when the number of SNPs increases dramatically. The index size is independent of number of SNPs. However, the size of the query increases linearly with the increasing number of used SNPs.

Next, we evaluated the index merging algorithm with 100, 200, 280, and 500 hospitals. Each hospital is assigned with 10 patients and each patient has 20 SNPs. The results are shown in Table 4. Notably, we observed that the time cost of Index-Merge algorithm increases quadratically with the increasing number of hospitals. In addition, we also evaluated the fast
approach for $IndexMerge$ (introduced in Section 5.2.5). Our results show that the fast approach is more than 290 times faster than the above method.

As discussed in Section 5.4.3, our proposed merge algorithm does not guarantee generation of a balanced tree. Thus, we also compared the performance of the search operation on (i) a balanced tree (that is constructed using plaintext data) and (ii) the merged tree using our proposed algorithm. We observed that the search time over the merged index increases by only around 17 percent (for 100 hospitals), 7 percent (for 200 hospitals), 5 percent (for 280 hospitals), and 1 percent (for 500 hospitals), respectively.

As discussed, the benefit of applying the $IndexMerge$ algorithm is to reduce the search time (i.e., to search over a merged index of multiple hospitals rather than searching over separate indices of individual hospitals). To justify this, we analyzed and compared the time costs of the $Search$ and $SearchOverMergedIndex$ algorithms. Table 4 shows that the time cost of $Search$ algorithm increases linearly with the increasing number of hospitals while the time cost of $SearchOverMergedIndex$ algorithm increases sublinearly. When the number of hospitals reaches 100, $SearchOverMergedIndex$ algorithm has significant advantage compared to the $Search$ algorithm.

We also compared the proposed mechanism with the state-of-the-art, including Wang et al.’s [10], Asharov et al.’s [12], and Schneider et al.’s [13] schemes. All of the three works adopted the same input parameters (number of SNPs and number of patient records) as the experiments conducted by Wang et al. [10], Asharov et al. [12], and Schneider et al. [13].

We did the comparison on a single hospital’s dataset (as compliant with the settings in [10] and [12]), in which the number of patients is 2850, the number of SNPs per patient is 3350221 (the whole sequence), and the query includes all 3350221 SNPs. We first implemented Wang et al.’s scheme [10] including (i) protocol one, which includes a bucketing technique to improve the secure computation of set difference size and (ii) protocol two, which replaces the square operation of protocol one with an estimation of normal distribution. We show the results in Table 5. We observed that our proposed scheme performs almost 63 times faster than the best case of protocol two of Wang et al.’s scheme. Moreover, our scheme provides higher accuracy than [10]. Next, we compared the proposed scheme with Asharov et al.’s [12] and Schneider et al.’s [13] schemes. We show the comparison result in Table 6. We observed that the running times of Asharov et al.’s and

<table>
<thead>
<tr>
<th>TABLE 2</th>
<th>Performance (in Terms of Time Cost in Seconds and Storage Cost) of the Proposed Scheme With Different Number of Patients in a Hospital’s Dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of patient records</td>
<td>10</td>
</tr>
<tr>
<td>IndexGen (s)</td>
<td>0.001</td>
</tr>
<tr>
<td>HierarchicalIndexGen (s):</td>
<td>0.24</td>
</tr>
<tr>
<td>QueryGen (s)</td>
<td>0.005</td>
</tr>
<tr>
<td>Search (s)</td>
<td>0.037</td>
</tr>
<tr>
<td>Index Size in RAM (B)</td>
<td>7.4K</td>
</tr>
<tr>
<td>Index Size in Disk (B)</td>
<td>45.5M</td>
</tr>
<tr>
<td>Query Size (B)</td>
<td>8.28K</td>
</tr>
</tbody>
</table>

In all scenarios, each patient has 20 SNPs.

<table>
<thead>
<tr>
<th>TABLE 3</th>
<th>Performance (in Terms of Time Cost in Seconds and Storage Cost) of the Proposed Scheme With Different Number of SNPs per Patient</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of SNPs per patient</td>
<td>200</td>
</tr>
<tr>
<td>IndexGen (s)</td>
<td>0.65</td>
</tr>
<tr>
<td>HierarchicalIndexGen (s):</td>
<td>19775</td>
</tr>
<tr>
<td>QueryGen (s)</td>
<td>0.005</td>
</tr>
<tr>
<td>Search (s)</td>
<td>10.93</td>
</tr>
<tr>
<td>Index Size in the RAM (B)</td>
<td>1005K</td>
</tr>
<tr>
<td>Index Size in the Disk (B)</td>
<td>13.3G</td>
</tr>
<tr>
<td>Query Size (B)</td>
<td>51.3K</td>
</tr>
</tbody>
</table>

In all scenarios, the hospital has 2850 patients.

<table>
<thead>
<tr>
<th>TABLE 4</th>
<th>Performance of Index Merging Algorithm (in Terms of Time Cost in Seconds) With Different Number of Hospitals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Hospitals</td>
<td>100</td>
</tr>
<tr>
<td>$IndexMerge$ (s)</td>
<td>2363</td>
</tr>
<tr>
<td>$SearchOverMergedIndex$ (s)</td>
<td>3.9</td>
</tr>
<tr>
<td>$Search$ (s)</td>
<td>5.8</td>
</tr>
</tbody>
</table>

Each hospital has 10 patients and each patient has 20 SNPs. “Search” processes the datasets of the hospitals one-by-one, while “SearchOverMergedIndex” processes over the merged index.

<table>
<thead>
<tr>
<th>TABLE 5</th>
<th>Comparison of the Proposed Mechanism With Wang et al.’s Scheme [10]</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k$</td>
<td>$l$</td>
</tr>
<tr>
<td>3</td>
<td>256</td>
</tr>
<tr>
<td>5</td>
<td>256</td>
</tr>
<tr>
<td>$k$</td>
<td>$l$</td>
</tr>
<tr>
<td>3</td>
<td>256</td>
</tr>
<tr>
<td>5</td>
<td>256</td>
</tr>
</tbody>
</table>

Protocol one in [10]:

- $k$ and $l$ are the number of iterations and adopted hash functions respectively.

Protocol two in [10]:

- $k$ and $l$ are the number of iterations and adopted hash functions respectively.

- Bloom filter capacity, error rate, Run-time (s), Accuracy
7.2 Experiments on Amazon EC2

To show the scalability of our scheme, we also conducted experiments on Amazon EC2 [44] and especially evaluated the QueryGen, Search, IndexMerge, and SearchOverMergedIndex algorithms.

7.2.1 Data Model

Using genomic data of 2850 patients (from OpenSNP dataset [46]), we extracted the statistics of the observed SNPs. Using these, we synthetically generated 10000 patients (synthetic data generation is also used in other similar works, such as [10] and [13]). In detail, we first assigned SNP IDs to 10000 patient records based on the extracted distribution. Then, following the extracted statistics, we assigned a SNP value for each SNP of each patient record. We then assigned the generated patients to 100 hospitals (each hospital has 100 patient records). In the following experiments, we first built the index for each hospital. The index is built based on all the SNPs (the maximum is 3350221) of each patient record.

7.2.2 Results

First, we evaluated the performance of the IndexGen and HierarchicalIndexGen algorithms. We observed that the time costs for IndexGen is 742.51 seconds and HierarchicalIndexGen is 896.93 seconds for building a hierarchical index. In Fig. 9a, we show the performance of the QueryGen algorithm for different number of SNPs in the query. We observed that the time cost of QueryGen increases slightly with increasing number of SNPs in the query. We then evaluated the time cost of the Search algorithm and observed (in Fig. 9a) that search time increases linearly with increasing number of SNPs in the query.

Next, in Fig. 9b, we show the performance of the IndexMerge algorithm. For the evaluation of the IndexMerge algorithm, we adopted the fast approach described in Section 5.2.5. We observed that the time cost of the IndexMerge algorithm increases superlinearly with increasing number of indices. Finally, in Fig. 9c, we show the time cost of the SearchOverMergedIndex algorithm while varying the number of merged indices and SNPs in the query. We

### TABLE 6

<table>
<thead>
<tr>
<th># of patients</th>
<th># of SNPs per patient</th>
<th>Run-time (s)</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>714</td>
<td>0.26</td>
<td>94.28%</td>
</tr>
<tr>
<td>100</td>
<td>1950</td>
<td>0.68</td>
<td>99.67%</td>
</tr>
<tr>
<td><strong>Schneider et al.’s scheme</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>1000</td>
<td>1.2</td>
<td>⊥</td>
</tr>
<tr>
<td>1000</td>
<td>75M</td>
<td>24480</td>
<td>⊥</td>
</tr>
<tr>
<td><strong>Proposed mechanism</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2850</td>
<td>20</td>
<td>10.036</td>
<td>≥ 99%</td>
</tr>
<tr>
<td>2850</td>
<td>2000</td>
<td>11</td>
<td>≥ 99%</td>
</tr>
<tr>
<td>2850</td>
<td>3350221</td>
<td>11.59</td>
<td>≥ 99%</td>
</tr>
</tbody>
</table>

⊥ means the accuracy is not reported.

Schneider et al.’s schemes are sensitive to the number of used SNPs in the query. The run-times of [12] and [13] increase linearly with the number of SNPs in the query while the run-time of our proposed scheme is almost constant. For example, when the number of SNPs in the query reaches to 3350221, the query times of both Asharov et al.’s and Schneider et al.’s schemes exceed 1100 seconds, which is more than 95 times slower than our proposed scheme.

Next, we evaluated the performance of ASI sharing focusing on a single ASI with 20, 200, and 2000 associated SNPs, respectively. We first assumed that the index search result only contains one pseudonym and each patient record only contains one ASI. The results are shown in the Table 7. We observed that the time costs of Encrypt, SharedKeyGen, TokenGen, TokenAdjust, and ASISearch algorithms increase linearly with the increasing number of associated SNPs with the ASI. On the other hand, the time cost of ASIDecrypt algorithm is almost constant with increasing number of SNPs.

Finally, we considered a more general case in which each patient has different number of ASIs, each having 20 associated SNPs. The result are shown in Table 8. We observed that with increasing number of ASIs, the time costs of SharedKeyGen, TokenGen, TokenAdjust, and ASIDecrypt algorithms are almost constant. Also, the time costs of Encrypt and ASISearch algorithms increase linearly with increasing number of ASIs.

### TABLE 7

<table>
<thead>
<tr>
<th>Number of SNPs</th>
<th>20</th>
<th>200</th>
<th>2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASIEncrypt (s)</td>
<td>2.95</td>
<td>28.87</td>
<td>293.4</td>
</tr>
<tr>
<td>SharedKeyGen (s)</td>
<td>0.092</td>
<td>0.86</td>
<td>8.55</td>
</tr>
<tr>
<td>TokenGen (s)</td>
<td>0.083</td>
<td>0.83</td>
<td>8.3</td>
</tr>
<tr>
<td>TokenAdjust (s)</td>
<td>0.487</td>
<td>4.86</td>
<td>48.6</td>
</tr>
<tr>
<td>ASISearch (s)</td>
<td>2.3</td>
<td>24.7</td>
<td>244.3</td>
</tr>
<tr>
<td>ASIDecrypt (ms)</td>
<td>0.066</td>
<td>0.07</td>
<td>0.068</td>
</tr>
</tbody>
</table>

### TABLE 8

<table>
<thead>
<tr>
<th>Number of ASIs</th>
<th>10</th>
<th>100</th>
<th>1000</th>
<th>7388</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASIEncrypt (s)</td>
<td>29.055</td>
<td>290.007</td>
<td>2885.7</td>
<td>21298.1</td>
</tr>
<tr>
<td>SharedKeyGen (s)</td>
<td>0.094</td>
<td>0.093</td>
<td>0.097</td>
<td>0.11</td>
</tr>
<tr>
<td>TokenGen (s)</td>
<td>0.084</td>
<td>0.084</td>
<td>0.084</td>
<td>0.084</td>
</tr>
<tr>
<td>TokenAdjust (s)</td>
<td>0.487</td>
<td>0.493</td>
<td>0.494</td>
<td>0.494</td>
</tr>
<tr>
<td>ASISearch (s)</td>
<td>2.506</td>
<td>3.74</td>
<td>14.39</td>
<td>89.3</td>
</tr>
<tr>
<td>ASIDecrypt (ms)</td>
<td>0.065</td>
<td>0.074</td>
<td>0.076</td>
<td>0.073</td>
</tr>
</tbody>
</table>

Each ASI has 20 associated SNPs.
observed that the number of SNPs has a limited impact on
the search efficiency compared to the number of indices.
Moreover, we observed that when the number of indices
reaches 50, the time cost of sequentially calling Search is
at least 170 seconds, while the maximum time cost of SearchOverMergedIndex is 166 seconds. This also supports the experi-
mental result obtained on the local machine, which shows
that 100 indices are enough to benefit from index merging.

8 CONCLUSION
In this paper, we have proposed a privacy-preserving and
efficient solution for the similar patient search problem
among several hospitals. To achieve this, we have proposed
a novel privacy-preserving index structure. To improve the
efficiency of the search operation, we have developed a hier-
archical index structure (to index each hospital’s dataset
with low memory requirement) and a novel privacy-pre-
serving index merging mechanism that generates a common
search index from individual indices of each hospital. We
have also considered the search for medical information
(e.g., diagnosis and treatment) that is associated with geno-
mic data of a patient. We have developed a scheme that
allows access to this information via a fine-grained access
control policy. Via simulations on real and synthetic geno-
omic data, we have shown the practicality and efficiency
of the proposed scheme. We believe that the proposed scheme
will further facilitate the use of genomic data in clinical set-
ings and pave the way for personalized medicine. In future
work, we will focus on supporting dynamic datasets and
we will extend our scheme to support batch search.

REFERENCES
https://cloud.google.com/life-sciences
dence for likelihood-based paternity inference in natural pop-
the future of health care: Toward predictive, preventative, and per-
sonalized medicine,” J. Proteome Res., vol. 3, no. 2, pp. 179–196,
2004.
1983.
“Identifying personal genomes by surname inference,” Science,
genomic data-sharing beacons,” Amer. J. Hum. Genet., vol. 97,
[8] H. Samet, Applications of spatial data structures. Boston, MA, USA:
Addison-Wesley, 1990.
[10] X. S. Wang, Y. Huang, Y. Zhao, H. Tang, X. Wang, and D. Bu,
“Efficient genome-wide, privacy-preserving similar patient query
based on private edit distance,” in Proc. 22nd ACM SIGSAC Conf.
ing search of similar patients in genomic data,” in Proc. Priv.
serving similar sequence queries on outsourced genomic data-
pp. 315–327.
“Countering gattaca: Efficient and secure testing of fully-
[16] M. Kantarcioğlu, W. Jiang, Y. Liu, and B. Malin, “A crypto-
graphic approach to securely share and query genomic
genome-wide association studies,” in Proc. IEEE Secur. Privacy
Workshops, 2015, pp. 3–6.
[18] Z. Jha, B. Peng, X. Wang, and H. Tang, “Large-scale privacy-
preserving mapping of human genomic sequences on hybrid
Hubaux, “Privacy-preserving processing of raw genomic data,” in Proc. Data Privacy Manage. Auton. Spontaneous Secur., 2013,
pp. 133–147.
for genomic computation,” in Proc. IEEE Symp. Secur. Privacy,
preserving error resilient DNA searching through oblivious
p. 519–528.
[22] Z. Huang et al., “A privacy-preserving solution for compressed
storage and selective retrieval of genomic data,” Genome Res.,

Authorized licensed use limited to: ULAKBIM UASL - Bilkent University. Downloaded on February 15, 2023 at 07:48:43 UTC from IEEE Xplore. Restrictions apply.


Xiaojie Zhu received the MS degree in computer science from the Chinese Academy of Sciences, Beijing, China, in 2015. He is currently working toward the University of Oslo, Oslo, Norway. His research interests include cloud security, data processing and privacy, and applied cryptography.

Erman Ayday (Member, IEEE) received the MS and PhD degrees from the Georgia Institute of Technology, in 2007 and 2011, respectively. He is currently an associate professor with Case Western Reserve University, Cleveland, OH, USA, and Bilkent University, Turkey. He was a postdoctoral researcher with EPFL, Lausanne, Switzerland. His research interests include privacy-enhancing technologies, which include big data and genomic privacy, wireless network security, and trust and reputation management.

Roman Vitenberg (Member, IEEE) is currently a professor with the Department of Informatics, University of Oslo. He has authored or co-authored more than 70 publications in peer-reviewed venues and has filed five patents. His research interests include distributed applications, middleware, and algorithms, including specification, design, analysis, implementation, performance evaluation, and software engineering. In particular, he has been working on large-scale communication, privacy and security, data storage, distributed event-based systems, fault-tolerant distributed computing, and more recently, blockchain. He is an associate editor for the EAI Transactions on Cloud Computing and a steering committee member for ACM DEBS. His papers were the recipient of the best paper awards at ACM/FIP/USENIX Middleware, ACM SAC, and ACM DEBS conferences.

Narasinha Raghavan Veeraragavan received the master’s degree from Vrije Universiteit, the Netherlands and the Phd degree from the University of Oslo. He is currently a postdoctoral researcher with the Department of Informatics, University of Oslo. He has several years of technical lead experience in global tech organizations and has also built a few privacy-aware distributed applications that serve hundreds of millions of users across the world. He has filed four patents, and authored or co-authored research papers in many peer-reviewed venues. His research interests include building and deploying dependable distributed systems with privacy and security taking lead roles.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.