PrivGenDB: Efficient and privacy-preserving query executions over encrypted SNP-Phenotype database

Sara Jafarbeiki a,b,c, Amin Sakzad a, Shabnam Kasra Kermanshahi c, Raj Gaire b, Ron Steinfeld a, Shangqi Lai a,b, Gad Abraham d, Chandra Thapa b

a Faculty of Information Technology, Monash University, Clayton Campus, Melbourne, Australia
b CSIRO DATA61, Australia
c The School of Computing Technologies, RMIT University, Melbourne, Australia
d The Systems Genomics Laboratory, Baker Institute, Melbourne, Australia

A B S T R A C T

Privacy and security issues limit the query executions over genomics datasets, notably single nucleotide polymorphisms (SNPs), raised by the sensitivity of this type of data. Therefore, it is important to ensure that executing queries on these datasets do not reveal sensitive information, such as the identity of the individuals and their genetic traits, to a data server. In this paper, we propose and present a novel model, we call PrivGenDB, to ensure the confidentiality of SNP-phenotype data while executing queries. The confidentiality in PrivGenDB is enabled by its system architecture and the search functionality provided by searchable symmetric encryption (SSE). To the best of our knowledge, PrivGenDB construction is the first SSE-based approach ensuring the confidentiality of SNP-phenotype data as the current SSE-based approaches for genomic data are limited only to substring search and range queries on a sequence of genomic data. Besides, a new data encoding mechanism is proposed and incorporated in the PrivGenDB model. This enables PrivGenDB to handle the dataset containing both genotype and phenotype and also support storing and managing other metadata, like gender and ethnicity, privately. Furthermore, different queries, namely Count, Boolean, Negation and k’-out-of-k match queries used for genomic data analysis, are supported and executed by PrivGenDB. The execution of these queries on genomic data in PrivGenDB is efficient and scalable for biomedical research and services. These are demonstrated by our analytical and empirical analysis presented in this paper. Specifically, our empirical studies on a dataset with 5000 entries (records) containing 1000 SNPs demonstrate that a count/Boolean query and a k’-out-of-k match query over 40 SNPs take approximately 4.3 s and 86.4 μs, respectively, outperforming the existing schemes.

1. Introduction

Advances in genotyping technology have enabled the genotyping of large human cohorts and biobanks (e.g., UK Biobank [1,2], NIH All of Us [3], and the Million Veteran Program [4]). Consequently, there are large human genomics datasets. These datasets are critical for advancing our understanding of human health, disease and their genetic components, for improving treatment practices, and for better delivery of healthcare. Specific examples include diagnosing a genetic predisposition to disease and precision medicine (e.g., predicting the outcome of a specific drug or treatment) [5,6]. In addition to these large datasets, hospitals and research institutions often collect similar data (on a smaller scale) and store them in local databases with limited resources, while the overall volume of data can be huge.

In genetics, single-nucleotide polymorphisms (SNPs) are considered as the most prevalent type of genetic variations. Many genetic analyses involve tasks identifying the associations between SNPs and characteristics or illnesses. These associations in the genome are crucial for clinicians to provide personal and effective medical care to their patients. Besides, in genetic association studies, analysts determine which genes are involved in the development of a certain illness in humans. Such connections are identified by leveraging statistical approaches such as linkage disequilibrium, the Cochran–Armitage test...
Informatics in Medicine Unlocked 31 (2022) 100988

2

a Symmetric Encryption; Hybrid: Software & Hardware, PH: Phenotype.

Existing query execution schemes on genomics data comparison.

S. Jafarbeiki et al.

encrypted data [9,18–24]. These solutions provide some or all of

sensitive health data like genomics data.

solutions are more attractive for privacy-preservation and security of
de-identification approaches struggle to defend against re-identification
to manage sensitive information via de-identification [26]. However,
Accountability Act of 1996 (HIPAA) [25] for instructions on how

an essential factor in the development of electronic frameworks for

individuals/entities is a key challenge [15]. Moreover, data privacy remains
and maintenance while enabling appropriate access to authorised indi-

sensitivity, and the requirements of computation, secure management
and the information residing on the servers may be exploited by an

However, cloud-based servers can be vulnerable to cyber breaches,
cloud storage and computations are performed by cloud computing.

capability, and (iii) disclosure of family relationships [10]. Besides,
with individual characteristics and certain disorders, (ii) identification
an individual. The dataset can reveal information such as (i) connection

whether the number of matches exceeds the query’s specified threshold.

This time is for just 173 records and 1000 SNPs with query consisting of 25 SNPs.

Efficiency is compared in terms of query execution time, over 5000 entries (records) and 1000 SNPs, with query consisting of 10 SNPs for all schemes except [19,21,23] that are
specified [24,25,26].

Notations: PE: Paillier Encryption; AES: Advanced Encryption Standard; GC: Garbled Circuit; SCP: Secure Cryptographic Coprocessor; SGX: Software Guard Extensions; SSE: Searchable
Symmetric Encryption; Hybrid: Software & Hardware, PH: Phenotype.

$^a$Phenotype contains the specific disease/trait, not only positive/negative indications indicating if a record has a certain phenotype or not; M: Metadata; $k'$: $k'$-out-of-$k$ match; C:

Count; N: Negation; B: Boolean.

$^b$Security guarantee is met with some leakage.

$^c$Efficiency is compared in terms of query execution time, over 5000 entries (records) and 1000 SNPs, with query consisting of 10 SNPs for all schemes except [19,21,23] that are
specified.

$^d$Time for 100,000 SNPs and less than 173 records.

for trend (CATT), the Pearson goodness-of-fit test or a chi-squared
test [7,8]. However, knowing how many entries in the database match
the genomics/disease information provided in the query predicates is
one of the requirements for computing these statistical tests. Therefore,
some useful queries, such as count query for counting the specific records, help genome-based researchers with their studies, for example,
SNP-disease association study using linkage disequilibrium [9]. Other
useful queries in genomic data are Boolean query, $k'$-out-of-$k$ match, and negation predicates. Boolean query is executed to know a specific information about the patients, like their IDs, based on their SNPs, and $k'$-out-of-$k$ match query is executed to know if different SNPs with specific genotype information are included in the patient’s profile or whether the number of matches exceeds the query’s specified threshold.

Despite the usefulness of the genomics data and its analysis (through query), maintaining its security and privacy at the highest level are essential. This is because genomics data has sensitive information about an individual. The dataset can reveal information such as (i) connection with individual characteristics and certain disorders, (ii) identification capability, and (iii) disclosure of family relationships [10]. Besides, due the huge volume of the genomics dataset, it is usually stored in cloud storage and computations are performed by cloud computing. However, cloud-based servers can be vulnerable to cyber breaches, and the information residing on the servers may be exploited by an adversary capable of violating security [11–14].

Overall, considering the huge volume of the genetic dataset, its high sensitivity, and the requirements of computation, secure management and maintenance while enabling appropriate access to authorised individuals/entities is a key challenge [15]. Moreover, data privacy remains an essential factor in the development of electronic frameworks for storing/managing genomics data [16,17].

Traditional strategies for protecting the privacy of health data in-
clude de-identification. Refer to the Health Insurance Portability and
Accountability Act of 1996 (HIPAA) [25] for instructions on how
to manage sensitive information via de-identification [26]. However,
de-identification are often not sufficient for data privacy [27]; most
de-identification approaches struggle to defend against re-identification attacks, and lead to utility loss [28,29]. In this regard, cryptographic solutions are more attractive for privacy-preservation and security of sensitive health data like genomics data.

Several existing cryptographic solutions, such as Paillier encryption, Garbled circuit, and Trusted execution environments, have been proposed for encrypting genomics data and running queries on this encrypted data [9,18–24]. These solutions provide some or all of the following privacy patterns that can be defined when outsourcing genomics data and conducting different queries against it: Data Privacy (DP), Query Privacy (QP), and Output Privacy (OP) [20] (see definitions in Section 4). The main security goal here is to protect the shared genomics data, the queries, and the unencrypted results of the executed queries by the analysts/clinicians (as users) from the Data Server (where the data is stored, e.g., the cloud server). The studies related to the information gain and leakages at a client from the query results are out of scope. Table 1 depicts an overview of these cryptographic solutions. Moreover, it includes their cryptographic primitive, solution type, types of the dataset used, supported queries, and privacy considerations taken into account by these schemes for overall comparison.

Besides security goal, our functionality goal is to provide a solution for several different queries executions on the SNP-phenotype database. Existing schemes do not execute a set of valuable queries (count, Boolean for retrieving data, $k'$-out-of-$k$ match, negation) altogether on encrypted genomics data. Moreover, they are less efficient to benefit cutting-edge biomedical applications (refer to Table 1). Furthermore, they provide limited support for storing and managing dataset containing additional information (like gender and ethnicity) apart from the genomics data and phenotypes privately. Some of these solutions are only software-based and asymmetric, while others are hybrid, meaning they exploit both software and hardware. Most importantly, none of the previous schemes are software-based and symmetric. A symmetric software-based scheme offers low computation complexity and thus fast queries executions. This is because it uses one shorter key unlike in the asymmetric case where multiple long keys are used, and no overhead due to the use of extra hardware like in hardware based schemes where frequent switching between application and enclaves are required. In this regard, searchable symmetric encryption, which is a popular symmetric software-based scheme, is considered in our studies.

Searchable encryption provides search functionality over encrypted data and is a cryptographic technique. It has two key research aspects: searchable symmetric encryption (SSE) and public-key encryption with keyword search (PEKS) [30]. SSE has obtained a lot of attention from the researchers and evolved well, resulting in more expressive searches and increased efficiency [31–36]. SSE has been used in genomics data analysis, but it is limited only to substring search [37], or range query [38] on encrypted sequences of genomics data. The proposed method in [39] does not consider a middle entity and let users submit specific types of queries by interacting with the server. So, there is no

Table 1

Existing query execution schemes on genomics data comparison.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Method</th>
<th>Primitive</th>
<th>Solution</th>
<th>Functionality</th>
<th>Privacy</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>[9] PE</td>
<td>Asymmetric</td>
<td>Software</td>
<td>SNP</td>
<td>C</td>
<td>Data</td>
<td>Query</td>
</tr>
<tr>
<td>[18] PE</td>
<td>Asymmetric</td>
<td>Software</td>
<td>SNP, PH</td>
<td>C, top $k$</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>[19] PE</td>
<td>Asymmetric</td>
<td>Software</td>
<td>SNP</td>
<td>C, $k'$</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>[20] PE,GC</td>
<td>Asymmetric</td>
<td>Software</td>
<td>SNP, PH$^*$</td>
<td>C</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>[21] PE,SGX</td>
<td>Asymmetric</td>
<td>Hybrid</td>
<td>SNP, PH</td>
<td>C</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>[22] SCP,AES</td>
<td>Symmetric</td>
<td>Hybrid</td>
<td>SNP, PH</td>
<td>C</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>[23] PE,GC</td>
<td>Asymmetric</td>
<td>Software</td>
<td>SNP, PH</td>
<td>C</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>[24] PrivGenDB</td>
<td>SSE</td>
<td>Symmetric</td>
<td>Software</td>
<td>SNP, PH, M</td>
<td>C, N, $k'B$</td>
<td>●</td>
</tr>
</tbody>
</table>

Time for 100,000 SNPs and less than 173 records.
other controlling and checking on the queries and results. This means users are limited to submit specific types of queries, but nothing more than that is being controlled before accessing the server and submitting the queries. Due to SSE’s efficient search functionality over encrypted data via indexing, it can provide confidentiality to the data, and to the best of our knowledge, this is the first research that considered SSE for maintaining the confidentiality of SNP-phenotype data and controlling the access to the server, the queries, and the results.

1.1. Our contributions

The main contributions of this paper are as follows:

• PrivGenDB Model and Construction:
  
  We propose a new model, we call PrivGenDB, for providing privacy of outsourced SNP-phenotype data stored on the cloud. PrivGenDB provides data confidentiality and query privacy. It also provides output privacy for the sensitive information retrieved from the server (see Table 1-Privacy for comparing provided privacy requirements in different schemes). The privacy-preservation in PrivGenDB is enabled by the proposed system architecture and the search functionality provided by searchable symmetric encryption (SSE). Inspired by Oblivious Cross Tag (OXT) SSE scheme introduced in [40], we construct PrivGenDB, the first secure and efficient SSE-based scheme for SNP-phenotype data privacy as the current SSE-based approaches for genomic data are limited only to substring search and range queries on a sequence of genomic data. Besides, we present a novel encoding method (GeEncode-described in Section 3.3) for genomic data, and construct an inverted index based on that encoding.

• PrivGenDB Functionality:
  
  The proposed encoding method (GeEncode), and hence the designed PrivGenDB, enable handling datasets containing genotype and phenotype (the exact disease) and other information like gender and ethnicity (see Table 1-Queries). Furthermore, different queries on genomic data (see Table 1-Queries), such as count query, Boolean query, $k'$-out-of-$k$ match query and negation predicates (see Section 2.1.2 for detailed information about these queries) are supported by PrivGenDB.

• PrivGenDB Efficiency evaluation:
  
  Our proposed PrivGenDB is the first model that provides the privacy of data, query and output (details in Section 4) for SNP-phenotype database without relying on hardware (e.g., using SGX) and is based on symmetric cryptographic primitives only that offers low computation complexity and thus fast query executions. We evaluate, both asymptotically and numerically, the computational costs (related to the initialisation and search phases), communication costs, and storage size of our PrivGenDB scheme for executing the above-mentioned queries and compare these against the state-of-the-art schemes (see Table 4 and Section 6). We have conducted several experiments and the results show the effectiveness of our SSE-based PrivGenDB in comparison with previous works, which were mainly based on the Pailler Encryption algorithm. For example, for a count query on a dataset of 5000 records and 1000 SNPs, it takes approximately 130.8 s in [20], while PrivGenDB improves the query execution time to 1.3 s only.

2. Preliminaries

2.1. Biology background

This section provides background on genomic data, phenotype, and some common queries against this type of dataset.

2.1.1. Genomic data and phenotype

The human genome consists of DNA (deoxyribonucleic acid) with two long complementary polymer chains of four bases called nucleotides, C (Cytosine), A (Adenine), T (Thymine), and G (Guanine). Single nucleotide polymorphism (SNP) is a variation in the genome where more than one base (A, T, C, G) is observed in a population. The majority of SNPs are biallelic, that just two potential variations (alleles) detected. An individual’s genotype is the set of particular alleles that they carry. SNPs make up a significant portion of genetic variation underpinning several human features like height and illness propensity (also known as phenotype). Table 2 depicts one sort of representing data in a database of genotypes (of SNPs) for several individuals [20,22], as well as the individuals’ phenotype.

2.1.2. Queries on genomic data

While biobanks contain individuals’ genomic data, different types of queries need to be performed on the data to help researchers analyse and correlate diseases and genome variations such as SNPs. Here are some query types over genomic data:

Count query. This determines the number of entries that satisfy the query terms in the database and helps in computing several statistical measurements for genetic association studies. A count query example on the dataset represented in Table 2 is as follows:

\[ \text{Count the entries with sequences that diagnoses } = \text{Cancer B AND SNP}_2 = \text{CC AND SNP}_4 = \text{AG} \]

The query size is defined as the total number of SNPs indicated in the query predicate. In the above-mentioned example, the query size is 2, and the answer to the mentioned query is 2 (i.e., records 2 and 5).\(^1\)

Boolean query. A query from clinicians can be about identifying patients who have specific genotypes so that they can contact them to provide personalised medication or inform them about their predisposition for drug allergy or vaccine. This could be done by obtaining the IDs their genotypes satisfy the query keywords and then utilising the IDs to retrieve additional information requested by the clinician.\(^2\) A Boolean query example on the dataset represented in Table 2 is as follows:

\[ \text{Retrieve IDs from sequences where diagnoses } = \text{Cancer B AND SNP}_2 = \text{CC OR SNP}_4 = \text{AG} \]

The answer to the above query is a list of records’ IDs that match the predicates and is: (2,5,7).

\(k'\)-Out-of-$k$ match query. Sometimes the answer to this query can be yes/no when the query symbols are being checked for a particular record. In this case, we can check if the SNPs for a particular record have the query predicates’ characteristics or not by some threshold that is defined. In our scenario, presented in algorithms in Section 3.3, the IDs related to records that have this threshold matching can also be retrieved to examine their additional information, such as illnesses, treatments or medications. Examples of a threshold query on the dataset represented in Table 2 are as follows:

\[ \text{Does record number } 7 \text{ have at least } k' = 2 \text{ matches out of } k = 3: \]

\[ \text{ID } = 7 \text{ AND SNP}_2 = \text{CC AND SNP}_4 = \text{CT AND SNP}_4 = \text{AG}. \text{ If this ID satisfies the query predicates, the answer will be ID, which means yes. Otherwise, it will return 0, which means no. Answer: yes.} \]

---

\(^1\) For count query: researchers can count the number of records with a specific genotype(s) for a given SNP(s) while having a particular disease. Furthermore, some other categorisation can also be added to the query, e.g., if they are male/female or even in a certain ethnicity or nationality group.

\(^2\) Of course, access control has to be considered since this type of information can be released to the clinicians, not the analysts (note that we do not discuss access control mechanisms in this paper; however, they have been extensively studied in the literature).
In the above-mentioned queries, negation keywords could also be added as predicates. For instance,

**Count the entries with sequences that diagnoses = Cancer B AND SNP₂ ≠ CC AND SNP₄ = AG.** Answer: 1 (Record 7 satisfies the query predicates).

### 2.2. Cryptographic background

This section reviews the cryptographic primitives we opted for in our proposed model.

#### 2.2.1. Searchable symmetric encryption (SSE)

One significant contribution in the field of Searchable Encryption is the Oblivious Cross Tag (OXT) technique proposed by Cash et al. [40], which is important due to its functionality as it supports conjunctive queries and Boolean queries. OXT provides a solution for searches over multiple keywords in the form of general Boolean queries. A special data structure called TSet is used for single keyword searches. Another dataset named XSet is used for multiple keyword searches to verify if the identified documents for the first term also satisfy the rest of the terms. Although the search query type is a Boolean kind rather than a single term, the OXT technique provides sublinear search time. This is important due to its functionality as it supports conjunctive and is used to verify the membership of an element in a set of collections of equivalent bit strings indexed by the entities of \( \mathcal{W} \), in which for each \( w \in \mathcal{W} \), \( T[w] \) is a list that contains the strings \( (s_1, \ldots, s_t) \) or of strings. \( T[w] \) includes one tuple for each DB document that matches \( w \), i.e., \( T_w = |DB(w)| \). It outputs a pair \( \text{TSet}, K_T \).

#### 2.2.2. Pseudorandom functions (PRFs)

In our framework, we use PRFs as the deterministic encryption to hide the search queries and tokens. It has also been used in the generation of TSet for the confidentiality of indexes and keywords. A PRF [41] is a set of efficiently computable functions, where no efficient algorithm (polynomial-time algorithm) can differentiate between a randomly selected function from the PRF family, as well as a random oracle (that is, a function whose outputs are entirely random), with a substantial advantage. Pseudorandom functions are essential tools in the development of cryptographic primitives, defined as follows:

Let \( X \) and \( Y \) be sets, \( F : \{0,1\}^s \times X \to Y \) be a function, \( s \)-S be the act of allocating a random element of \( S \) to \( s \), \( \text{Fun}(X, Y) \) define the collection of all the functions from \( X \) to \( Y \), \( \lambda \) be the security parameter, and \( \text{neg}(\lambda) \) denote a negligible function. We call \( F \) a pseudorandom function if for all efficient adversaries \( \mathcal{A} \),

\[
\text{Adv}^\text{PRT}_\mathcal{A}(\lambda) = \text{Pr}[FA^\perp(\lambda^t) = 1] - \text{Pr}[FA^\perp(\lambda^t) = 0] \leq \text{neg}(\lambda),
\]

where the probability is over the randomness of \( A, \lambda \sim \{0,1\}^\lambda \), and \( f \) is a function.

#### 2.2.3. Bloom filter

A Bloom filter BF [42] is a kind of data structure which is probabilistic and is used to verify the membership of an element in a set of \( N \) elements. The point is to select \( k \) different hash functions, \( \{H_i\}_{i \in \Sigma} \). The Bloom filter consists of an \( m \)-bit binary vector, all bits set to 0. The bits at positions \( \{H_i(n)\}_{i \in \Sigma} \) are modified to 1 for each element \( n \) in the set to set up BF for the set. We search if \( b \) has 1’s in all the positions \( \{H_i(q)\}_{i \in \Sigma} \) to verify membership of \( q \) in the set and if so, we infer \( q \) has a high likelihood of being in the set. Otherwise, \( q \) is not in the set with probability 1. There are possible false-positive matches when \( q \) is not in the set, and still the membership check returns 1. The false-positive probability of \( q \) over a uniformly random selection of \( \{H_i\}_{i \in \Sigma} \) is

\[
P_F \leq (1 - e^{-kN/m})^k.
\]

We use the Bloom filter to keep a part of encrypted database in a tiny portion of the server’s RAM.

### 3. Methodology

In this section, first, the used notations are presented, and then a general architecture of the PrivGenDB system model and threat models are illustrated. Finally, a detailed explanation of PrivGenDB methodology is presented along with the PrivGenDB algorithms.

#### 3.1. Notations

Table 3 lists the notations that are frequently used in this paper.
receives the correct keys from the query. We finally assume that \( v \) is honest (honest-but-curious adversary). That means \( v \) does not collude with \( h \). It follows that \( v \) is a trusted entity checking the legitimacy of users (Analysts or Clinicians) can be unauthorized, so they will be authorized through protocol. We assume the trustee to be a trusted entity. Users (Analysts or Clinicians) can be unauthorized, so they will be authorized through protocol. We assume the trustee to be a trusted entity.

Our security goal is not to let the Data Server (D) learn about the shared genomic data and the unencrypted results of the executed query by the analysts/clinicians. The Data Server is assumed to be semi-honest (honest-but-curious adversary). That means D correctly follows the protocol and has no intention of maliciously behaving to produce the wrong result. However, D may try to gain more information than what is expected to be extracted during or after the execution of the protocol. We assume the trustee to be a trusted entity. Users (Analysts or Clinicians) can be unauthorized, so they will be authorized through the vetter, which herself is a trusted entity checking the legitimacy of the query. We finally assume that D does not collude with \( U' \), and \( V \) receives the correct keys from \( T \).

### 3.2. System model overview

The proposed system is composed of different entities, as shown in Fig. 1. Their functions are as follows:

- **Data owner \( O \):** A data owner is a person whose data is gathered. When a data owner arrives at a medical institution like a gene trustee, as a volunteer in a research or as a patient, her data is collected and recorded while her consent is given to the trustee to use her genome data for future research or treatment.

- **Data provider or Trustee \( T \):** A medical institution, such as gene trustees, acts as a data provider in PrivGenDB. \( T \) keeps a list of collected genome data with consent related to them. We assume that the data provider is trustworthy. The main responsibilities of the Trustee are: (i) data encoding and generation of an inverted index, (ii) encrypting the generated inverted index, (iii) managing the cryptographic keys.

- **Users – Analysts or Clinicians – \( U' \):** Users are either analysts, who want to get the count results for analysis/research purposes, or clinicians, who want to get information about the patients. They send their detailed requests to the vetter and wait for the result of the query execution. Different users are able to submit specific queries are (e.g., analysts can submit count queries, and clinicians can submit count/boolean/\( k' \)-out-of-\( k \) match searches).

- **Vetter \( V \):** This is a trusted role to check the access privileges and vet the request submitted to the system. Vetter \( V \) approves/denies the request, which is sent by the analyst/clinician. Furthermore, there is another vetting process when the result is being released.

- **Data Server \( D \):** The data server records genomic data, phenotype, and other information provided by the data provider. The \( D \) executes the encrypted queries on encrypted data and sends back the result.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID(_O)</td>
<td>Data Owner’s ID</td>
</tr>
<tr>
<td>EID(_O)</td>
<td>Encrypted Data Owner’s ID</td>
</tr>
<tr>
<td>( \Theta )</td>
<td>The set of keywords</td>
</tr>
<tr>
<td>( S )</td>
<td>The set of all SNP indices ( s )</td>
</tr>
<tr>
<td>( G_s )</td>
<td>The set of all genotypes of that specific SNP</td>
</tr>
<tr>
<td>( G_\rho )</td>
<td>The set of all genotypes related to other information, e.g., gender and ethnicity</td>
</tr>
<tr>
<td>( \rho )</td>
<td>The exact terms being asked in a query</td>
</tr>
<tr>
<td>( \psi )</td>
<td>The least frequent ( \rho ) among terms in the query</td>
</tr>
<tr>
<td>( O )</td>
<td>Data Owner’s IDs containing a particular ( s ), that is either phenotype or genotype or information like ethnicity</td>
</tr>
<tr>
<td>( \mu )</td>
<td>The set of all keywords in the database</td>
</tr>
<tr>
<td>(</td>
<td>\mu</td>
</tr>
<tr>
<td>( \mathbb{G}_D )</td>
<td>The set of Data Owner IDs containing a particular ( s ), that is either phenotype or genotype or information like ethnicity</td>
</tr>
</tbody>
</table>

### 3.3. PrivGenDB: Privacy-preserving query executions on genomic data

This section describes the architecture of PrivGenDB with the details of designed algorithms. PrivGenDB consists of four phases: \( \Pi_G = (\text{PrivGenDB.Initialisation}, \text{PrivGenDB.QuerySubmission}, \text{PrivGenDB.Search}, \text{PrivGenDB.ResGeneration}) \). The sequence diagram of our proposed model is depicted in Fig. 2 which depicts the protocol sequence.

PrivGenDB.Initialisation(\( \lambda \), \( G \)). This algorithm is managed by the Trustee \( Trustee \) and is illustrated in 1. By inputting the security parameter \( \lambda \) and the desired genotype-phenotype database \( G \) containing genotypes \( \Theta \) for each SNP, \( \rho \) phenotypes \( \in \mathbb{G}_s \), other information \( \in \mathbb{G}_d \), this algorithm generates the encrypted database \( EGD \).

This algorithm follows the sub-algorithms:

- **GenEncode(\( G \), \( D \)):** \( T \) generates the keywords \( s \in \mathbb{G}_s \) by concatenation of the SNP index to each \( \theta \), \( s = 1||\theta \). For example, based on Table 2, there would be three defined keywords for SNP\(_1\), those are: 1AG, 1GG, 1AA. When \( T \) receives the blood samples, sequences it and extracts the variations from it, creates a table like Table 2. Apart from the genotype for each SNP, \( T \) obtains additional information on the data owner, such as phenotype, gender, and ethnicity. The \( T \) also creates \( G_\rho \) for phenotype and \( G_\mu \) for other columns representing other information, like ethnicity and gender. For example, for Table 2, we have: \( |\mathbb{G}_\rho| = (1AG, 1AA, 1GG, 2AG, 2CT, 3AG, 3GG, 3TT, 4AG, 4AA, 4GG) \), \( |\mathbb{G}_\mu| = \{\text{Cancer A}, \text{Cancer B}, \text{Cancer C}\} \).

- **Binv(\( G \), \( D \)):** \( T \) runs this algorithm, that inputs the keywords in \( G \) and the database \( G \) to generate an inverted index IXN. This
Algorithm 1 PrivGenDB: Initialisation

PrivGenDB Initialisation(A, DB)
Input: A, DB
Output: EDB, K
The Trustee performs:
1. G = GeEncode(GDB)
2. INIX = BInv(G, GDB)
3. (EDB, K) = EGBN Setup(A, INIX, GDB)
4. The trustee outsources EGBN to the T.
5. The trustee sends K = (K_X, K_Y, K_Z) to the Yeter.

GeEncode(GDB)
Input: GDB
Output: G
1. \( \ell \) = the number of SNPs in GDB
2. for \( S = 1, \ldots, \ell \) do
3. \( G_{S} = \emptyset \)
4. for each distinct genotype \( g \) in column:"SNP," do
5. generate \( \ell = |g| \)
6. \( G_{g} = G_{S} \cup g \)
7. \( G_{g} = \emptyset \) and \( G_{S} = \emptyset \)
8. for each distinct keyword \( g \) in column:"Phenotype" do
9. \( G_{g} = G_{S} \cup g \)
10. for each distinct keyword \( g \) in column:"Gender, Ethnicity" do
11. \( G_{g} = G_{S} \cup g \)
12. return \( G = \{G_{1}\}_{1 \leq S} \cup G_{g} \).

BInv(G, DB)
Input: G, DB
Output: INIX
1. Initialise INIX as an empty array that is indexed by keywords in G
2. for each keyword \( g \) in G do
3. \( \text{Output}(g, \text{ID}_g) \)
4. for each keyword \( g \) in G do
5. for all ID_g appended to g do
6. \( \text{DB}(g) = \text{ID}_g \)
7. INIX = \( (\_g, \text{DB}(g)) \)
8. return INIX

EGBN Setup(A, INIX)
Input: A, INIX
Output: EDB, K
1. Select key \( K_A \) for PRF F and keys \( K_X, K_Y, K_Z \) for PRF \( F \) using security parameter \( \lambda \), and \( G \) a group of prime order \( p \) and generator \( h \).
2. Pane DB as \( (\text{ID}_g, g) \) and let \( G = \emptyset \)
3. for \( g \) in G do
4. \( \text{Initiate inv} \left( g \right) \); and let \( K_A = F(K_A, g) \)
5. for \( \text{DB}(g) \in \text{GDB} \) do
6. Set a counter \( c = 1 \)
7. Compute \( \text{IDDB}(g) = F(g(K_A, ID_g), z) = F(g(K_A, \delta_g); \gamma) = \text{IDDB}(\gamma^{-1}) \), \( IDDB_g = E(K_A, IDDB) \), \( \delta_g \)
8. Set \( t_f = F(K_A, \delta_g \times \text{IDDB}) \) and let \( t_{g} = t_f \cup \delta_g \cup t_f \)
9. Append \( (g, \text{ID}_g) \) to \( c \) and let \( c = c + 1 \)
10. \( \text{INIX}(g) = \text{inv} \)
11. Set \( (\text{INIX}, K_A) = \text{TSetSetup}\text{(INIX)} \) and let EDB = \( (\text{INIX}, \text{DB}) \)
12. return EDB and \( K = (K_X, K_Y, K_Z) \).

![Fig. 2. PrivGenDB sequence diagram.](image)

PrivGenDB QuerySubmission(U, \( q(1), \ldots, q(n) \), K). This algorithm (depicted in 2), managed by the Y, inputs the query \( q \), the key set \( K \) and user \( U \) and generates the search token gToK by running below algorithms:

- **ReqV(U, q(1), \ldots, q(n))**: Y authorises the access of the \( U \) and also checks the query legitimacy. This algorithm outputs bit \( b = 1 \) if \( U \) is allowed to execute the submitted query \( q \) and \( b = 0 \) unless otherwise.

- **PrivGenDB** supports queries from researchers (who are interested in statistical analysis) and clinicians (who want to treat diseases of their patients or diagnose and prevent diseases by personalised medicine).

- Therefore, in this phase, \( Y \) checks the accessibility of researchers and clinicians and lets them execute their related queries.

- **QEncode(q(1), \ldots, q(n))**: Based on the predicates in the query \( q \), the \( Y \) generates the keywords for sending to \( D \). The genotype-related predicates will be encoded, and the new query will be produced as \( q(g_1, \ldots, g_n) \). Query encoding happens similar to the Initialisation phase. As an example, let the query be: count the entries that: SNP1 = CC and SNP2 = AG and diagnoses = Cancer.B. This is encoded to: \( g_1 = 2CC \) and \( g_2 = 4AG \)

- **TokerGen(g(1), \ldots, g(n), K)**: This algorithm takes the new query predicates and the key set as inputs, so it generates the token for the search, gToK. In the TSet.GetTag(K, g) from TSet in OXT [40] is invoked here. \( Y \) generates tokens as follows:

- The least frequent term, which in our database is the phenotype \( g_1 \), is used to produce \( r_g \).
- \( r_g \) is sent to \( D \) to let it process the queries based on their corresponding type.
- In the case of count/Boolean queries, one token is created for non-negated requested genotypes/phenotypes/other information, and one token is created for negated keywords.

\[4\] The main idea is to search for the least frequent term as the first keyword to lessen search complexity. Some steps in this process happen for all types of queries that PrivGenDB supports. D performs each query by receiving a unique integer \( r \) from \( Y \).

Different authentication/authorisation mechanisms exist, such as role-based access control that can be utilised in this model, but discussing them is outside the scope of this work.
Algorithm 2 PrivGenDB.QuerySubmission

```
PrivGenDB.QuerySubmission(U', q₁,...,qₖ, K)
Input: U', q₁,...,qₖ, K
Output: gToK, y
The Vetter performs:
1: b = ReqV(U', q₁,...,qₖ)
2: if b = 1 then
3: q₁,...,qₖ ← QEncode(q₁,...,qₖ)
4: (gToK, y) ← TokGen(q₁,...,qₖ, K)
5: Vetter sends gToK to the D
6: else reject the request and inform the U'
TokGen(q₁,...,qₖ, K)
Input: q₁,...,qₖ, K
Output: gToK, y
1: Computes rᵢ ← TSet.GetTag(Kᵢ, qᵢ).
2: V' sends rᵢ to D.
//Depending on the type of query, V' generates gToK:
```

Algorithm 3 PrivGenDB.Search

```
PrivGenDB.Search(gToK, EGDB)
Input: gToK, EGDB
Output: RSet, [ID'₁]
The search is carried out by D using the input ρ.
1: RSet ← \{\}
2: inv ← TSet.Retrieve(QEncode, rᵢ)
3: for i = 1,...,|inv| do
4: Retrieve(ID'Oᵢ, y) from the e-th tuple in inv
//Based on the type of the query, D performs search:
5: if y = 1 or 2 then
6: if Qencode(i, j) ∈ S₀ for all i = 2,...,n in Qencode and Qencode(i, j) ∈ S₀ for all i in Qencode
then
```

4. Security analysis

Our protocol’s security definition, $H_G$, is parameterised by a leakage function $L_D$ (information about the database and queries obtained by the data server $D$ through contact with a secure technique). This demonstrates how the perspective of $D$ can be simulated using simply the output of $L_D$ in an adaptive attack (queries and database picked by $D$). For an adversary $A$ and a simulator $Sim$, we define a real experiment $Real_{A,Sim}(λ)$ and an ideal experiment $Ideal_{A,Sim}(λ)$ for an adversary $A$ and a simulator $Sim$, respectively:

$Real_{A,Sim}(λ)$: $A(1^n)$ chooses GDB and a set of queries $q$. Then, the experiment performs $(K, EGDB) ← Initialisation(λ, GDB)$ and gives $EGDB$ to $A$. Then $A$ repeatedly chooses a query $q(λ)$. The experiment runs the algorithm $QEncode$ on input $q(λ)$ to get $(q)$ and $TokGen$ on inputs $(q), (g), (K)$, and returns search tokens to $A$. Eventually, $A$ returns a bit that experiment uses as its output.

$Ideal_{A,Sim}(λ)$: The game starts with a counter $i = 0$ and an empty list $q(1^n)$ chooses GDB and a query list $q$. The experiment runs $EGDB ← Sim(L_D(GDB))$ and gives $EGDB$ to $A$. Then, $A$ repeatedly chooses a search query $q$. To respond, the experiment records this query as $q(i)$, increments $i$ and gives the output of $Sim(L_D(GDB), q)$ to $A$, where $q$ consists of all previous queries as well as the latest query issued by $A$. Eventually, the experiment outputs the bit that $A$ returns.

**Definition 1 (Leakage to Data Server).** Our scheme’s $L_D(GDB, q)$, where $GDB = \{(ID_O, g)\}$, $r$ represents the number of Data Owners and $q =$
Algorithm 4 PrivGenDB.ResGeneration

PrivGenDB.ResGeneration(RSet|Res, T,K)
Input: RSet|Res, T,K
Output: Res
The V performs:
1: if the output is a set RSet then
2: ISet ← Retrieve(RSet, T, K)
3: \( Res_v \leftarrow OutV(\text{ISet}) \)
4: otherwise Res_v ← OutV(RSet, T)
5: Vetter sends Res_v to the User

Retrieve(RSet, T, K)
Input: RSet, T, K
Output: ISet
1: ISet ← \{\}
2: Vetter sets \( K_v \leftarrow F(K_S, g_1) \)
3: for each \( ID_\gamma \in \text{RSet received} \) do
4: Compute \( ID_\alpha \leftarrow \text{Gen}(K_v, ID_\gamma) \)
5: if \( 1 \leq ID_\alpha \leq T \) then
6: return \( ID_\alpha \)
7: return \( ID_\alpha \)

\((g_1[i], g_2[i])\), where \( g_1 \) is the stem and \( g_2 \) is the xterms, is defined as \((N, \delta, \text{SP}, \text{RP}, \text{IP}, \text{XP}, \text{QT}, \text{NP})\). The first six leakages mentioned are like the OXT leakage in [40]. The other ones are minor leakages related to the fact that our SSE-based scheme supports different queries on the genomic dataset.

Proof. Let \( A \) be a semi-honest (honest-but-curious) data server that launches an adaptive attack against \( \Pi_G \), our protocol. Then, we are able to construct an algorithm \( B \) that violates the privacy of the server in OXT protocol [40] by executing \( A \) as a subroutine with non-negligible probability. Algorithm \( B \) runs GeEncode on the selected GDB by \( A \) and passes the output (G) to the OXT challenger. Then, the OXT challenger runs \( K(EBD) \leftrightarrow OXT.EDBSetUp(\text{G})) \) and \( \text{EBDSetUp} \) to the algorithm \( B \). Then, the algorithm \( B \) sends \( EDB = EDB \). The algorithm \( B \) sends \( EDB \) to an adversary \( A \). For each query \( q[i] \) issued by the adversary \( A \), algorithm \( B \) first runs GeEncode to generate the predicates for computing tokens. Then, algorithm \( B \) defines TokGen(K; \( q[i] \)) which uses the output of the Token Generation oracle of OXT \((q[i]) \) in TokGen has the encoded predicates). It classifies phrases as negated or non-negated and executes the OXT Token Generation algorithm twice to create gTok for content retrieval and Boolean searches. For \( k\)-out-of-\( k \) matches, it simply removes the \( k' \) from a query sent by \( A \). The remaining to the OXT TokenGeneration algorithm, and creates the gTok by utilising that output and includes the \( k' \) to transmit it to \( A \). Finally, the adversary \( A \) produces a bit \( h \), which is returned by the algorithm \( B \). Because the core structure of our scheme is inspired by OXT in [40], we can utilise the OXT oracle to reduce our proof of theorem to OXT protocol. As a result, if the security of OXT is preserved, our scheme’s security is ensured.

Simulator 1. It is only remaining to present an efficient simulator \( S_G \) that takes as input the leakage function \( L_c \) described in Definition 1 and outputs an EDB and simulates the tokens for the queries as follows. \( L_{\Pi_G}(\text{GDB, q}) = L_{\Pi_{OXT}}(\text{GDB, q'}) \), where \( q' = T(q) \), and \( T(q) \) maps the query vector \( q \) of \( \Pi_G \) given by attacker \( A \) to query vector \( T(q) \) for OXT used by attacker \( B \). If \( q \) is a Boolean or a count query, \( T(q) \) generates two sets for negated and non-negated encoded terms in \( q \); hence, \( q' \) is two separate queries to the simulator of OXT. For \( k\)-out-of-\( k \) match queries, \( T(q) \) is the encoded \( q \) without \( k' \). Such a simulator can be constructed for these queries by utilising \( S_{OXT} \), an OXT protocol simulator. Using all leakage components except NP, QT. the \( S_G \) can simulate the \( T_r, \text{Goken} \) by running \( S_{OXT} \). The extra leakage components \( NP, QT \) are exploited by algorithm \( B \) to construct gTok completely and send it to adversary \( A \). We only need to utilise the OXT simulator for \( A_{OXT} \) to build our scheme’s simulator for \( A_G \). We can construct a simulator \( S_G \) for EDBSetup and TokGen queries of our scheme by executing \( S_{OXT} \).

\[
\Pr \left( \text{Real}_{A}^{\Pi_G} = 1 \right) - \Pr \left( \text{Ideal}_{A,S_G}^{\Pi_G} = 1 \right) \leq \text{neg}\left(\lambda\right)
\]

\[
\Pr \left( \text{Real}_{B}^{\Pi_{OXT}} = 1 \right) - \Pr \left( \text{Ideal}_{B,S_{OXT}}^{\Pi_{OXT}} = 1 \right) \leq \text{neg}\left(\lambda\right)
\]

Therefore, since OXT is secure and its advantage is negligible, our protocol’s advantage is negligible.
• Data Privacy (DP): the genomic data stored in the cloud server should be protected, and no information about the data should be leaked. The confidentiality of the data should be preserved, even though the cloud server is compromised.
• Query Privacy (QP): entities such as the cloud server, data owners, and the compromised server should not learn anything beyond what is predicted about the query, which is executed by a researcher/clinician.
• Output Privacy (OP): the result of the query should not be revealed to other entities except the intended recipient.

The purposes behind requiring these privacy patterns are: (i) to protect the individuals’ data from data server or any other entity trying to get their information (DP), (ii) to preserve the clinicians’ patients’ records or analysts’ research objectives from the server (QP), and (iii) to provide privacy for individuals whose records are retrieved and privacy for analysts/clinicians’ objectives (OP).

Discussion: PrivGenDB provides these privacy patterns with some mentioned leakages. The attacks defined in [43] are categorised based on different categories (e.g., active/passive, minimum leakage needed for the attack to be applied). The active attacks cannot be applied to PrivGenDB since the middle entity, Vetter, controls all the accesses to the data server and does not write chosen documents to the server (there is no write permission for the Vetter). Amongst the passive attacks, just query recovery can be applied to PrivGenDB because the other ones need much more leakage than what PrivGenDB leaks. In particular, leakages I2, L3 and L4 defined in the body of [43] are not leaked by PrivGenDB. So, only query recovery attacks can be applied here, but with the assumption that the server has extensive prior knowledge about the dataset. In fact, for such attacks to be practical a minimum of approximately 80% of the dataset should be leaked. Reaching the 80% guess of the dataset is rather a strong assumption, and we do not reveal the information about the dataset to the data server (or cloud server) when we outsource our encrypted data.

It is worth mentioning that vetter plays an important role in providing output privacy in terms of what is revealed to the researchers, as the vetter checks all the outputs and vet the output to release it. In comparison to previous schemes, vetter entity has been proposed as the vetter checks all the outputs and vet the output to release it. Providing output privacy in terms of what is revealed to the researchers, a minimum of approximately 80% knowledge about the dataset. In fact, for such attacks to be practical a minimum of approximately 80% of the dataset should be leaked. Reaching the 80% guess of the dataset is rather a strong assumption, and we do not reveal the information about the dataset to the data server (or cloud server) when we outsource our encrypted data.

5.1. Initialisation computational cost

In the initialisation/setup phase of all the schemes in Table 4 except [20], the computational cost is in the order of the number of records (r) multiplied by the number of columns in the original database, x (which is the number of SNPs + 1 for phenotype + the number of other characteristics if they support, e.g., gender, ethnicity) [9, 18,19,22]. The tree structure proposed in [20] needs the encryption of all nodes in the setup phase. The number of nodes increases by the number of SNPs. As acknowledged by the author of [20], in our correspondence, if the number of records in the database increases, the number of nodes increases as well. We also assume that each SNP can have three possibilities of genotypes, and when the number of records increases, the possibility of having a new genotype for the SNPs increases. This assumption is completely dependent on the genotypes distribution in the dataset. The computational complexity of PrivGenDB in this phase is in the order of r.x. All methods in Table 4 are linear in rx, except [18] which does not encrypt all the data and [20] which depends on the number of nodes (increases by x and r). The proposed methods in [23,24] have the same computation/communication complexities’ behaviour as that of [20], since they proposed their scheme based on the index tree in [20], with some added functionality or improvements.

5.2. Search computational cost

The computational costs are split between the client/trusted-entity/ vetter and the server during the search process. In PrivGenDB, the vetter represents the client, as it is the entity that interacts with the server to perform the search against the database. As it is obvious from the Table 4, the computational cost of the search through server is in the order of the number of records in [9,18,19,22]. Based on the index tree proposed in [20], their scheme’s search cost depends mainly on the number of SNPs. However, increasing the number of records also increases the number of nodes in their index tree to cover all the genotype possibilities (as discussed in part A). PrivGenDB’s search complexity depends on the number of IDs that match the first predicate in the query (a). As the phenotype was selected as the first predicate, which is the least frequent term in our database, PrivGenDB’s search complexity is sublinear to the number of records in the database. This implies that, even by increasing the number of records, if the number of matched records to the first predicate (or phenotype) does not change, the search complexity remains unchanged.

5.3. Storage size

We now investigate the storage size of PrivGenDB and compare it to other schemes. PrivGenDB stores InvG and the Bloom filter of $S_{ID}$ in EGD, while [9,18,19,22] store the encrypted data and [20] stores the encrypted tree in the database. The size of $Inv_{G}$ equals $rx(e_{rk} + e_{r})$, which is the number of pairs ((i, $ID_{j}^i$) in the construction), multiplied by the size of each pair. The size of Bloom filter m is approximately $1.44 \cdot krx$ to attain a negligible probability of false positives. Other schemes’ storage size also depends on rx, except [20], which only supports the count query and stores the counts in the tree and depends on the number of nodes.

5.4. Interaction rounds and bandwidth

In PrivGenDB, the first interaction between the $V$ and the D happens when the $V$ sends the search tokens to the $D$. Based on the type of the submitted query and its size, the size of the result sent to the $V$ differs, which defines the next interaction. Therefore, when the $V$ generates the $e_{r}$, it sends it to the $D$. Then, depending on the number of retrieved IDs for the $e_{r}$, the $V$ generates the $Token$ and sends them to $D$, which checks a set membership against $S_{G}$. Communication overhead in [20] depends on the number of nodes that need to be accessed during the search to perform the query.
6. Experimental results

This section illustrates the implementation results of PrivGenDB with various number of records/SNPs/query-sizes and different types of queries. The source code is written in Java, and our machine for running it is an Intel(R) Core(TM) i7-8850H CPU @ 2.60 GHz processor with 32 GB RAM running Ubuntu Linux 18.04. In order to analyse the efficacy of our proposed method, we evaluate the following factors:

- **Initialisation Time:** $T$: to perform $\text{Blinv}$ and $\text{EGDBSetup}$. Search
- **Query Execution Time:** $V$ to execute $\text{TokGen}$ and $D$ to run $\text{PrivGenDB.Search}$
- **Storage Analysis:** Storage space needed to store the database.
- **Communication Overhead:** The data transmitted between the $V$ and the $D$ to perform the query.

To increase our implementation’s runtime speed, an in-memory key-value store Redis [44] is used to store the created $\text{Inv}_\text{G}$ for querying reasons. Furthermore, we use Alexandre Nikitin’s [45] Bloom filter because it is the fastest Bloom filter implementation for JVM. The Bloom filter’s false-positive rate is set at $10^{-6}$, allowing our solution to maintain the $\text{SC}_\text{G}$ in a small proportion of the server’s RAM. We also used the Java Pairing-Based Cryptography Library (JPBC) [46]. We use real-life and synthetic datasets for evaluation of PrivGenDB. The real-life data come from The Harvard Personal Genome Project (PGP) [47], where we collected information on 58 patients with their 2000 SNPs, as well as their phenotype, ethnicity, and gender. As a phenotype, these individuals have had brain, breast, thyroid, prostate, colon, melanoma, or kidney cancer, or they were recently Covid-19 positive/negative. Then, we created several synthetic datasets based on the above-mentioned real dataset to evaluate PrivGenDB with a different number of records (500 – 40,000) and different number of SNPs (500 – 2000). The number of all keywords $g$ in the database represented by $|G| = |G_\text{P}| + |G_\text{E}| + |G_\text{S}|$ is related to the number of SNPs, phenotypes, and other information (gender and ethnicity in our dataset). Since each SNP has three possibilities of genotypes, $|G|$ in our dataset is equal to three times the number of SNPs in the database plus the number of phenotypes (10 in our dataset) plus two (for gender, female, male) plus four (for ethnicity, black, white, Asian, American–Indian). $|G| = 3 \times |S| + 10 + 2 + 4 + 4$ in our database.

### 6.1. Initialisation time

Initialisation time includes the time needed to generate the inverted index and encrypt it. Genomic data encoding time is negligible. Fig. 3 illustrates the initialisation time of PrivGenDB when the number of SNPs increases or the number of stored records $(r)$ increases. The time consumption for the generation of the inverted index and encrypting it with 5000 records, 1000 SNPs, different phenotypes, gender and ethnicity is around 518 seconds in PrivGenDB, while it is 47 min and 25 min in [18,20], respectively. Note that we encrypt all the information in our database to support different queries and this phase is needed to be performed once only.

### 6.2. Storage analysis

Since PrivGenDB needs to support different queries and have the capability of retrieving the IDs, it encrypts all the related information and not just the count of different SNPs.

Table 5 lists the amount of spaces required to store the original data and encrypted data. The left column represents the number of records in the dataset ranging from 500 to 40,000. The original dataset of size 1.5 MB is encrypted to 113.4 MB, while it is 188 MB in [20]. The expansion in the encrypted tree size of [20] is due to encrypting the data using the Paillier Encryption. It is worth mentioning the growth ratio in the storage overhead (expansion factor) of PrivGenDB is approximately 80, whereas it is 180 in [20], and it is in the order of 1024 (key bit length) in [5,19].

### 6.3. Query execution time

We conducted a series of queries of varying sizes on the encrypted database to calculate query execution time. The queries we used were determined by randomly selecting 10, 20, 30, and 40 SNP sequences with other information like gender and ethnicity. PrivGenDB works with sublinear search capability, making it a proper solution for large

---

**Table 4**

<table>
<thead>
<tr>
<th>Reference</th>
<th>PrivGenDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>[9]</td>
<td>[22]</td>
</tr>
<tr>
<td>[18]</td>
<td>[19]</td>
</tr>
<tr>
<td>[20]</td>
<td>[24]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Comp.</th>
<th>Initialisation</th>
<th>$\text{Search (Data Server)}$</th>
<th>Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>$\pi(T_F + T_F + \ell T_F)$, $\pi(T_{Fa})$</td>
<td>$\pi(n + \text{AC})$ \text{find} $n$ positions</td>
<td>$\pi(\ell T_F + \ell T_F)$</td>
</tr>
<tr>
<td>$\ell$</td>
<td>$\text{Search ($\text{Client}$)}$</td>
<td>$\pi(n + \text{AC})$</td>
<td>$\pi(\ell T_F + \ell T_F)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sto.</th>
<th>Storage Size</th>
<th>$\pi(\ell T_F + \ell T_F + \ell)$</th>
<th>$\pi(\ell T_F + \ell T_F)$</th>
</tr>
</thead>
</table>

| Notations: $T$: Time needed to compute an exponentiation; $T_F$: Time needed to compute a PRF; $T_S$: Time needed to compute a hash; $T_G$: Time needed to compute a Bloom Filter (BF) size GC: Garbled circuit computation; SCP: Secure Cryptographic Coprocessor computation; $\ell$: Least requested SNP index; $|R|$: Total number of records that matched all the predicates in the query; $\ell$ (query): Length of the submitted query.

$a$: The client performs computations to generate the query or help Data Server during the search process. This computation is the token generation phase in PrivGenDB.

$b$: Client is the entity that submits the query to the server and gets back the result. It has direct contact with the server. In previous works, it is either client or researcher or a trusted entity/proxy, and in our model, it is the vetter.

---

**Table 5**

<table>
<thead>
<tr>
<th>Database size (original and encrypted).</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Records(r)</td>
</tr>
<tr>
<td>#SNPs=500</td>
</tr>
<tr>
<td>Original</td>
</tr>
<tr>
<td>500</td>
</tr>
<tr>
<td>1500</td>
</tr>
<tr>
<td>5000</td>
</tr>
<tr>
<td>10,000</td>
</tr>
<tr>
<td>20,000</td>
</tr>
<tr>
<td>40,000</td>
</tr>
</tbody>
</table>
databases. Below, we discuss our results and show our model's scalability with respect to different characteristics of our database. Retrieve time is negligible.

6.3.1. Scalability of PrivGenDB in terms of the number of SNPs

We first experimented with the effect of the number of SNPs on the query execution time. As it is evident from Fig. 4, the time needed to perform a particular query on a dataset with 5000 records does not change when the number of stored SNPs for each record increases.

6.3.2. Scalability of PrivGenDB in terms of the number of records

The search complexity depends on the number of matched IDs to the first predicate (α) in the query, which we selected as phenotype. Therefore, even in a large database, the search complexity is in the order of the number of returned IDs for the phenotype. Fig. 5 illustrates the time taken for a query to be performed on a database when the number of records that match the phenotype differ. This experiment shows the token generation and search time depend only on α.

To explicitly show the scalability of PrivGenDB, we explored the query execution time on datasets with 500, 1000, and 1500 records, while they have the same number of IDs with particular cancer or Covid19+. Fig. 6.a represents different datasets with the same number of records for some phenotypes. Fig. 6.b shows that, for example, when the query’s first predicate is thyroid cancer in r = 1000 and r = 1,500 database, the search complexity is the same since both have 250 records with this disease. Moreover, the number of records with brain cancer is 50 in all three datasets, and it is obvious from Fig. 6.b that the token generation time and search time is the same in these three datasets, which indicates the search and token generation time complexities are independent of r and varies slightly with α.

6.3.3. Scalability of PrivGenDB in terms of k'-out-of-k match query for one patient

Our experiments show that the query execution time for conducting k'-out-of-k match query for a particular patient, when a clinician is interested, is mostly dependent on the time taken for the D to search for the ID in SΩ, which is the first predicate in this query. Token generation on Ω side and checking the genotypes on D side depends on the query size, but the whole process takes around 70 μs (see Fig. 7).

6.3.4. Performance comparison

Table 6 compares the query execution time of PrivGenDB with [9,18,20,22]. The times of [9,18,22] are reported directly from their original paper. The query execution time of our model depends mostly on the number of records matching the first predicate (α), which is the phenotype in our model, whereas it is linear to the number of records (r) for [9,18,22] and linear to the number of SNPs in [20]. In comparison to [20], which is the only other model that supports storing the phenotype in the database, PrivGenDB takes only 5.2 s to execute a query of size 50 on our 5000 database. It is worth mentioning that it may take less time if the number of IDs that match a phenotype is less than that in the experiment. Whereas, [20] takes around 130 seconds to execute the query on the encrypted tree structure. The query execution time in [19] is in the order of the number of records stored in the data server.

Table 6

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Phenotype</th>
<th>Query size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>[9]</td>
<td>no</td>
<td>1260 s</td>
</tr>
<tr>
<td>[22]</td>
<td>p/n</td>
<td>20 s</td>
</tr>
<tr>
<td>[18]</td>
<td>p/n</td>
<td>29 s</td>
</tr>
<tr>
<td>[20]</td>
<td>yes</td>
<td>130.8 s</td>
</tr>
<tr>
<td>[24]</td>
<td>yes</td>
<td>163.92 s</td>
</tr>
<tr>
<td>PrivGenDB</td>
<td>yes</td>
<td>1.3 s</td>
</tr>
</tbody>
</table>

p/n: positive/negative indicators showing if a record has or does not have a particular disease.

This row is related to the count and Boolean queries execution time.

This row is related to the k′-out-of-k match query execution time for a particular random patient on the database.

We acknowledge that running the experiments on the same machine as ours should improve the performance of [9,22].
Fig. 5. Query execution time on the dataset with 40,000 records (different number of records, α, match the phenotype in the query).

Fig. 6. Query execution time on different datasets with same number of records satisfy the phenotype in the query.

Fig. 7. k′-out-of-k match query execution time for one patient on dataset with 5000 records and 1000 SNPs.

6.4. Communication overhead

The amount of data transferred between the D and the V is depicted in Fig. 8. This is the data amount to execute the query with different query sizes while α changes. This overhead increases if the query size (n) or the number of records match the phenotype (α) increase, which defines the tokens generated by V and transmitted to the D for searching. Both the query execution time and communication cost generally depend on these two parameters, linearly. Query privacy is not provided in [19], and [18] does not encrypt the query. This overhead increases linearly if the number of nodes need to be accessed to perform a query increases in [20], and increases linearly by the number of records in [9,22].

7. Conclusion

In this paper, we proposed a new secure and efficient model, named PrivGenDB, for outsourcing Single Nucleotide Polymorphism (SNP)-Phenotype data to the cloud server. To the best of our knowledge, PrivGenDB is the first model that ensures the confidentiality
of shared SNP-phenotype data by employing searchable symmetric encryption (SSE) and makes the computation/query process efficient. PrivGenDB constructs an inverted index using a novel encoding mechanism for genotypes, phenotypes, gender, and ethnicity, then encrypts it. By employing SSE, the vetter can generate tokens, and the data server can execute different query operations. We have compared PrivGenDB to the state-of-the-art privacy-preserving query executions on SNP-Phenotype data both analytically and numerically. Our experimental results on real and synthetic data show that the proposed model outperforms the existing schemes in terms of query execution time. Finally, PrivGenDB is flexible to support consent, where data owners provide their consent and are allowed to revoke it later. This needs the system model to support deletions or deactivation of data for use. In other words, the system model would provide dynamicity and the interactions with the data owners would provide information for revocation. The privacy of the system needs to be provided. We consider adding consent and revocability features while still providing a low leakage profile as a promising research direction.

Declaration of competing interest

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

Acknowledgement

None to declare, except the authors that have their contributions to the paper.

References


