Sequence analysis

Efficient privacy-preserving string search and an application in genomics

Kana Shimizu¹,², *, Koji Nuida³,⁴ and Gunnar Rätsch², *

¹Biotechnology Research Institute for Drug Discovery, National Institute of Advanced Industrial Science and Technology, Tokyo 135-0064, Japan, ²Computational Biology, Memorial Sloan Kettering Cancer Center, New York, NY 1275 York, USA, ³Information Technology Research Institute, National Institute of Advanced Industrial Science and Technology, Tokyo 135-0064, Japan and ⁴Japan Science and Technology Agency (JST) PRESTO Researcher, Tokyo, Japan

*To whom correspondence should be addressed.
Associate Editor: Janet Kelso
Received on April 19, 2015; revised on November 27, 2015; accepted on January 19, 2016

Abstract

Motivation: Personal genomes carry inherent privacy risks and protecting privacy poses major social and technological challenges. We consider the case where a user searches for genetic information (e.g. an allele) on a server that stores a large genomic database and aims to receive allele-associated information. The user would like to keep the query and result private and the server the database.

Approach: We propose a novel approach that combines efficient string data structures such as the Burrows–Wheeler transform with cryptographic techniques based on additive homomorphic encryption. We assume that the sequence data is searchable in efficient iterative query operations over a large indexed dictionary, for instance, from large genome collections and employing the (positional) Burrows–Wheeler transform. We use a technique called oblivious transfer that is based on additive homomorphic encryption to conceal the sequence query and the genomic region of interest in positional queries.

Results: We designed and implemented an efficient algorithm for searching sequences of SNPs in large genome databases. During search, the user can only identify the longest match while the server does not learn which sequence of SNPs the user queried. In an experiment based on 2184 aligned haploid genomes from the 1000 Genomes Project, our algorithm was able to perform typical queries within $\approx 4.6$ s and $\approx 10.8$ s for client and server side, respectively, on laptop computers. The presented algorithm is at least one order of magnitude faster than an exhaustive baseline algorithm.


Contacts: shimizu-kana@aist.go.jp or Gunnar.Ratsch@ratschlab.org

Supplementary information: Supplementary data are available at Bioinformatics online.

1 Introduction

String search is a fundamental task in the field of genome informatics, for which a large variety of techniques have been developed (see, for instance, Altschul et al., 1990; Kent, 2002; Li and Homer, 2010). Traditionally, those techniques have been optimized for accuracy and computational efficiency, however a recent boom of personal genome sequencing and analyses has spotlighted a new criteria, namely, privacy protection. As reported in many studies, a genome is considered
to be one of the most critical pieces of information for an individual’s privacy. In fact, it is largely different from any other personal information because it works as an identifier of an individual while it possesses the information that has strong correlation with the phenotype of the individual (Erlich and Narayanan, 2014; Roche and Annas, 2001). Therefore, in principle, privacy protection is an inevitable problem when handling personal genomes. As a practice, the most popular approach is protecting genomes physically; genomic sequences have been kept at few collaborator sites, and only a limited number of researchers are allowed to access them. This conservative approach severely limits the great potential of existing genomic resources. In order to mitigate the stagnation caused by privacy issues, it appears crucial to develop practical methods that enable searching and mining genomic databases in a privacy-preserving manner.

So far, several groups have tackled related problems. Jha et al. (2008) developed secure multi-party computation protocols for computing edit distance. Blanton and Aliassgari (2010) proposed a protocol to search DNA string against a DNA profile represented by finite automata. Bruekers et al. (2008) proposed a protocol to detect a match between two short DNA sequences for the purpose of genetic test. Baldi et al. (2011) also aimed for genetic test to develop a method for computing set intersection cardinality. Freedman et al. (2005) proposed a protocol for searching pre-defined keywords from databases. Nagamura et al. (2012) proposed a substring search protocol for public databases while keeping user’s query private. Ayday et al. (2013) developed a system by using several cryptographic techniques to find a subset of short reads which includes a fixed-length query string at specific position. He et al. (2014) proposed an algorithm for finding relatives by secure identity-by-descent matches.

We propose a general approach which utilizes an efficient iteratively queriable data structure together with cryptographic techniques. Among many variations of such data structures, the Burrows–Wheelie Transform (BWT; Langmead et al., 2009; Li and Durbin, 2009; Li et al., 2009) and related techniques such as the positional BWT (PBWT; Durbin, 2014) have dramatically improved the speed of genomic database analyses. Those data structures commonly have an indexed dictionary called a rank dictionary. By referring to the rank dictionary in iterative operations, one can efficiently search the database. For the case of BWT, a match between query and database is reported as a left-open, right-closed interval search the database. For the case of BWT, a match between query string and database is reported as a left-open, right-closed interval search the database. For the case of BWT, a match between query and database is reported as a left-open, right-closed interval search the database. For the case of BWT, a match between query and database is reported as a left-open, right-closed interval search the database.

Let us describe the information flow between a user and a server for such problems. In this work, we perform searches on the (positional) Burrows–Wheelie Transform of a genomic database of length N. (P)BWT stores string information very efficiently and still allows computations (this is a property of Succinct Data Structures, see Jacobson, 1988).

To search for a query string q over the alphabet Σ, one iteratively operates on intervals that can later be used to identify the matching genomic regions based on the (P)BWT. A substring match is represented by an interval [f, g]. The number of matches is given by the length of the interval g−f. It is known that the (k + 1)th interval [f_{k+1}, g_{k+1}] corresponding to a (k + 1)-mer match can be updated from the kth interval [f_k, g_k] and the (k + 1)th letter of the query q.

We will provide more details on how to update f and g in Section 3.3. To understand the key ideas, it is sufficient to understand that the updates can be written in the form of:

\[ f_{k+1} = \nu_c[f_k] \quad \text{and} \quad g_{k+1} = \nu_c[g_k], \]

where \( c = q[k + 1] \) and \( \nu_c \in \mathbb{N} \) is a large, static lookup table. Hence, the iterative algorithm of updating \( f_k, g_k \) by using the query \( q \), can be written as a recursive algorithm:

\[ f_{k+1} = \nu_c[q[k+1]]\nu_c[q[k+1]]\nu_c[q[k+1]]\nu_c[q[k+1]]\nu_c[q[k+1]]\nu_c[q[k+1]]\nu_c[q[k+1]]\nu_c[q[k+1]] \]

This can be done analogously for \( g_{k+1} \). In this work we will refer to data structures that can be queried in the recursive way described above as recursive search data structures. Figure 1 illustrates the information flow of a search on the recursive search data structure.

2.3 Oblivious transfer for privacy-preserving search

In a search on the recursive search data structures, the user needs to conceal not only a query string \( q \) but also \( f \) and \( g \) because \( f \) is \( f_{1} \)-th element of \( v_{q[k+1]} \), and \( q[k+1] \) is inferred from those two values. Analogously, \( q[k+1] \) is also inferred from \( g \) and \( g_{k+1} \). The server needs to minimize output because the user reconstructs a part of \( v \) from the server’s output. In this study, we achieve such security requirements by a cryptographic technique called oblivious transfer.
are random values to the user. Only the server can recover $f_k$ and $g_k$ in encrypted form (i.e. the server does not see $f_k$ and $g_k$ when recovering them), and thus the user can recursively access $v_i[f_k]$ and $v_i[g_k]$ correctly. The details of this approach are given in Section 3.2.

In this work, we designed an algorithm based on these techniques that can be used for privacy-preserving search in large genotype databases.

Note that there are still privacy risks for the server, though returning only a unique match minimizes the information leakage from the server. For example, assume there is a database storing a genomic study of drug addicts that implements the PBWT-sec, and a person (Bob) participated in the study. If someone obtains a DNA sample from Bob and queries the databases, he/she will reveal that Bob is a drug addict. As described in the above case, there is always a limitation for protecting the server’s privacy as long as the server returns the search results, and there is associated information such as phenotypes (Shringarpure and Bustamante, 2015). We emphasize that this issue is common for any database search application and is not specific to our proposed method.

### 3 Methods

#### 3.1 Additively homomorphic encryption

Our main cryptographic tool in this paper is an additively-homomorphic public-key encryption scheme (KeyGen; Enc; Dec), which enables us to perform additive operations on encrypted values. Here, the algorithm KeyGen generates a public key pk and a secret key sk; Enc(m) denotes a ciphertext obtained by encrypting message m under the given pk; and Dec(c) denotes the decryption result of ciphertext c under the given sk. The scheme also has the following additively-homomorphic properties:

- Given two ciphertexts Enc($m_1$) and Enc($m_2$) of integer messages $m_1$ and $m_2$, Enc($m_1 + m_2$) can be computed without knowing $m_1$, $m_2$ and the secret key (denoted by Enc($m_1$) ⊕ Enc($m_2$)).
- Given a ciphertext Enc($m$) of a message $m$ and an integer $e$, Enc($e \cdot m$) can be computed without knowing $m$ and the secret key (denoted by $e \otimes$ Enc($m$)). In particular, Enc($-m$) can be computed in this manner.

This scheme should have semantic security; that is, a cipher text leaks no information about the original message (Goldwasser and Micali, 1984). For example, we can use either the Paillier cryptosystem (Paillier, 1999) or the ‘lifted’ version of the ElGamal cryptosystem (ElGamal, 1985); now the second operation $\otimes$ can be realized by repeating the first operation $\oplus$.

Figure 3 illustrates an outline of performing an additive operation on a user’s value $m_1$ and a server’s value $m_2$ by the additively homomorphic encryption. In the first step, the user generates two keys: a secret key and a public key, and the user sends the public key to the server. In the second step, the user encrypts $m_1$ by the public key and sends a ciphertext Enc($m_1$) to the server. In the third step, the server
3.2 Recursive oblivious transfer by random rotations

To protect the privacy of the database, we propose a technique for recursively querying a data structure without obtaining information about intermediate results. Let us define the recursive oblivious transfer problem as follows:

**Model 1.** A user has a private value \( 0 \leq x_i < N \) and a server has a private vector \( v \) of length \( N \). Let us denote \( x_{i+1} = v[x^i] \) and the user is allowed to access the server \( i - 1 \) times. After the calculation, the user learns only \( x_i \) and the server learns nothing about \( x_1, \ldots, x_i \).

Here we explain our idea by extending a simple linear communication size \( OT \) where the user aims to know the \( r \)th element of the server’s vector \( v \).

Figure 4 illustrates the oblivious transfer algorithm based on additive homomorphic encryption. In the initialization step, the user generates a public key and a secret key and sends the public key to the server. The server sends a ciphertext \( \text{Enc}(v) = (\text{Enc}(v_0), \ldots, \text{Enc}(v_{N-1})) \)

The server computes

\[
\hat{c} = \bigoplus_{i=0}^{N-1} (v_i \otimes \text{Enc}(q_i))
\]

and sends \( \hat{c} \) to the user.

The user computes \( \text{Dec}(\hat{c}) \) and obtains \( \hat{v}[t] \) by using the secret key because \( t = t \rightarrow \hat{v}[t] \otimes \text{Enc}(q_i) = \text{Enc}(v_i) \) and \( t \neq t \rightarrow \hat{v}[t] \otimes \text{Enc}(q_i) = \text{Enc}(0) \).

Now we consider the case that the server does not leak \( v[t] \), but allows the user to access \( v[\hat{v}[t]] \). Our idea is that the server generates a random value \( r \in \{0, 1, \ldots, N - 1\} \) and returns the ciphertext

\[
\hat{c} = \bigoplus_{i=0}^{N-1} ((v_i + r) \mod N) \otimes \text{Enc}(q_i) = \text{Enc}((v_i + r) \mod N)
\]

where \( (a + b) \mod N \) denotes addition in a number field modulo \( N \). The user decrypts \( \hat{c} \) to know a randomized result \( (v_i + r) \mod N \), and performs the next query:

\[
\check{q} = (\check{q}_0 = 0, \ldots, \check{q}_{N-1}) = (1, \ldots, 1) \mod N - 1
\]

Note that \( \check{q} \) is the \( r \)-rotated permutation of the ‘true’ query:

\[
\check{q'} = (\check{q}_0 = 0, \ldots, \check{q}_{N-1} = 1, \ldots, 1) \mod N - 1
\]

Therefore, denote \( \text{Perm}(q, r) \) as the permutation of a vector \( q \) such that \( \text{r} \)th element moves to \( \left((i - r) \mod N\right)\)th position, the server...
can correctly recover 'true' query $q'$ in its encrypted form by the following permutation: $\text{Enc}(q') = \text{Perm}(\text{Enc}(q), r)$. In this way, the server correctly computes an encrypted $v[i]_{th}$ element by

$$\text{Enc}(v[i]) = \bigoplus_{n=1}^{N-1} (v[i] \otimes \text{Enc}(q'_n)).$$

without learning any information about the user’s query.

By recursively applying these calculations, the user can obtain $x_{k+1}$ according to Model 1. The complete algorithm implementing this idea is given in Algorithm 1. It uses a function $\text{ROT}$ for rotating the server’s results to conceal intermediate query results in order to protect the database.

**Algorithm 1. Recursive oblivious transfer**

```plaintext
1: function PrepQuery($t, N$)
2: $q = (q_0 = 0, \ldots, q_t = 1, \ldots, q_{N-1} = 0)$
3: $\text{Enc}(q) = (\text{Enc}(q_0), \ldots, \text{Enc}(q_{N-1}))$
4: return $\text{Enc}(q)$
5: end function
6: $\text{function ROT(Enc}(q), v, r, r', N)$
7: $\text{Enc}(q') = \text{Perm(Enc}(q), r')$
8: $\hat{c} = \bigoplus_{n=1}^{N-1} ((v[i] + r)_{\text{mod}N} \otimes \text{Enc}(q'_n))$
9: return $\hat{c}$
10: end function
11: $\text{function ROT(Enc}(q), v, r, r', N)$
12: $v$ is a server’s private vector of length $N$.
13: $x_t$ is a user’s private value.
14: $\ell$ is known to both user and server.
15: User’s initialization: $t \leftarrow x_t$
16: Server’s initialization: $r' \leftarrow 0$
17: Common initialization: $i \leftarrow 1$
18: while $i < \ell$ do
19:   The user computes: $\text{Enc}(q) \leftarrow \text{PrepQuery}(t, N)$
20:   if $i == (\ell - 1)$ then
21:      Server sets: $r = 0$
22:      else
23:      Server generates random value $r$
24:      end if
25:   Server computes: $\hat{c} \leftarrow \text{ROT(Enc}(q), v, r, r', N)$
26:   Server sets: $r' \leftarrow r$
27:   Server sends $\hat{c}$ to user
28:   User computes: $t \leftarrow \text{Dec}(\hat{c})$
29: while end
30: User obtains $x_{k+1}$.
31: return $x_{k+1}$.
```

### 3.3 PBWT-sec: Privacy-preserving search on genotype databases

In this section, we introduce a practical genotype database search based on recursive oblivious transfer and PBWT. We only introduce the algorithm to search for the longest match starting from 0th column, however, variations are possible and would allow for a variety of different search types (see also Durbin, 2014).

To formulate the problem, let us consider a set $X$ of $M$ haplotype sequences $x_0$, $i = 1, \ldots, M$ over $N$ genomic positions indexed by $k = 1, \ldots, N$, and a query $q$ which is a user’s haplotype sequence over the same $N$ genomic positions. We denote $k$th allele of a sequence $x_i$ by $x_i(k)$. Given two indices $k_1$ and $k_2$, we say that there is a match between $q$ and $x_i$ from $k_1$ to $k_2$, if $q[k_1] \ldots q[k_2 - 1] = x_i[k_1] \ldots x_i[k_2 - 1]$. We say that the match is set-longest at $k_1$ if there is no match between $q$ and any sequence $x_i$ (possibly with $j = i$) from $k_1$ to $k_2 + 1$.

The goal is to find a set-longest match at a given position $t$ between $q$ and $X$ in a privacy-preserving manner. Here, we consider the case that the user’s private information is the query string and the position $t$ is not the user’s private information. We later introduce the case that the both the query string and $t$ are user’s private information. The formal description of the model is described as follows:

**Model 2.** The user is a private haplotype sequence holder, and the server is a holder of a set of private haplotype sequences. The user learns nothing but a set-longest match at a given position $t$ between the query and the database while the server learns nothing about the user’s query, $t$ is not a user’s private information and the server knows it.

Let us remember how to search the set-longest match in non-privacy-preserving manner. PBWT involves a matrix $P \in \mathbb{N}^{M \times N}$ that stores well-compressible information in an efficiently searchable form. It is created from the genotype matrix $X$ by algorithms described in Durbin (2014) such that $r$th column is $(i + 1)$th letters of sequences sorted by $i$ reverse prefix (i.e. sorted from $r$th letter to first letter). In order to compute the match starting from the first allele, $P$ has $0$th column $P_0 = (x_1[1], \ldots, x_M[1])^T$. By using rank dictionary operations on $P$ (see below), one can search a match between a query and $X$. When operating on $P$ one computes updates of intervals using the following two quantities (see Durbin, 2014 for more details): (i) The rank dictionary for sequence $S$ for letter $c \in \Sigma$ at position $t$:

$$\text{Rank}_c(S, t) = \left\{ j \mid S[j] = c, 1 \leq j \leq t \right\},$$

where $\Sigma$ is the alphabet of $S$. (ii) The table $\text{CF}$ counting occurrences of letters that are lexicographically smaller than $c \in \Sigma$ by

$$\text{CF}_c(S) = \sum_{j \in \text{Rank}_c(S, N)} \text{Rank}_c(S, j).$$

Based on these two quantities, we can compute the updates $(f_{k+1}, g_{k+1})$ using two simple operations

$$f_{k+1} = \text{CF}_c(P, k) + \text{Rank}_c(P, k, f_k),$$

$$g_{k+1} = \text{CF}_c(P, k) + \text{Rank}_c(P, g_k),$$

where we denoted the $k$th column vector by $P_k$. Let us define a look-up vector $v_c$ for the column $k$ where

$$v_c[i] = \begin{cases} \text{CF}_c(P, k) & (i = 0) \\ \text{CF}_c(P, k) + \text{Rank}_c(P, k, i) & (1 \leq i \leq M) \end{cases} \quad (1)$$

for $c \in \Sigma$. Then, updating an interval is equivalent to two look-ups in the vector $v_c$:

$$f_{k+1} = v_c[f_k] \quad \text{and} \quad g_{k+1} = v_c[g_k]. \quad (2)$$

Given a position $t$ and a PBWT $P$ of the database sequences, the first match is obtained as an interval $(f_1 = v_c[0], g_1 = v_c[M])$ where $c = q[1]$ and $v_c$ is a look-up vector for $(t - 1)$th column of $P$ (see the definition of $v_c$ in Equation (1)). The match is extended by one letter by an update of the interval. The update from the $k$th interval to $(k + 1)$th interval is conducted by specifying $c = q[k + 1]$, re-computing $v_c$ for $(k + 1)$th column of $P$ and referring $v_c[f_k]$ and $v_c[g_k]$ as $f_{k+1}$ and $g_{k+1}$ (see Equation (2)). The set-longest-match is found.
when \( f = g \). An outline of the search using PBWT is illustrated in Figure 5.

In order to achieve the security described in the model 2, for each update, the user has to specify \( c \) without leaking \( c \) to the server, and obtain only \( v_{c}(f) \) and \( v_{c}(g) \) without leaking \( f \) and \( g \). To satisfy the second requirement, the user accesses the server’s \( v_{c} \) through the function ROT, which allows the user to obtain a specific element in the specified vector. To achieve the first requirement, the server computes all possible intervals (i.e. computing \( (f, g) \) for all case of \( c = 0, \ldots, |Σ| - 1 \)). This allows the user to obtain the correct interval, however, the sever leaks extra information (i.e. intervals for \( c \neq q[k] \)). To avoid this, the user sends \( Enc(q[k]) \), and the server adds a conditional randomization factor \( r \times (q[k] - c) \) to \( f \) and \( g \) with different random value \( r \) for all \( c \in Σ \). Note that this factor becomes equal to 0 iff. \( q[k] = c \), and user only obtains the interval for \( c = q[k] \).

In order to identify the set-longest match, the user has to know if \( f \) = \( g \). The user cannot compute the identity of \( f \) and \( g \) directly from the server’s return, because ROT returns a value which is a random value to the user (but the true value is recovered in encrypted form only at the server side). Therefore, the server also sends an encrypted flag \( d \) which shows whether or not \( f \) = \( g \). Since \( f \) and \( g \) are represented as indices of \( q_{i} = Perm\{q_{i}, r^{(f)}\} \) and \( q_{j} = Perm\{q_{j}, r^{(g)}\} \) (see the functions PrepQuery and ROT), the server computes \( d \) by following:

\[
d = \bigoplus_{i=0}^{M} Enc(r_{i} \times (q_{i}[k] - q_{j}[k]))
\]

where \( r_{i} \) is a random value. \( Dec(d) \) is equal to 0 iff. \( q_{i} = q_{j} \). See Supplementary Algorithm S5 which defines a function isLongest. In addition to finding a set-longest match at \( t \), it is convenient to find a longest substring which matches to at least \( s \) sequences. This operation enables to avoid detecting unique haplotype and provides e-privacy result and is implemented by replacing the function: isLongest by another function: isLongestGT \( e \) which computes flags each of which shows if the interval matches to \( 0, \ldots, e - 1 \) respectively and returns shuffled flags, and the user knows the result by checking if there is a flag which is equal to zero. See Supplementary Algorithm S5 for more details.

The detailed algorithm of PBWT-sec is shown in Algorithm 2.

### 3.4 Concealing the search position

By the algorithm introduced above, the match position \( t \) needs to be provided to the server. Let us consider the case that \( t \) needs to be concealed (e.g. the used would not like to reveal which gene is analyzed). In practical genotypic database search, it is often sufficient for the user to hide \( t \) in a set of multiple columns. Therefore, here we assume the following security model.

**Model 3.** The user is a private haplotype sequence holder, and the server is a holder of a set of private haplotype sequences. The user has a vector of \( D \) positions \( T = (t_{1}, \ldots, t_{D}) \). The user learns nothing but a set-longest match at a given position \( t \in \{ t_{1}, \ldots, t_{D} \} \) between the query and the database while the server learns nothing about the user’s query string. The server knows \( T \) but cannot identify which element the user queries.

Conceptually, the user could query multiple positions at the same time to conceal the search position. In the extreme case the user would query all search positions to avoid leaking any information about \( t \). However, every answered query would leak more information from the database and querying would become computationally prohibitive. We therefore propose joint processing using OT that simultaneously uses multiple search positions. Let us define \( V_{L} \) as another look-up vector for a letter \( c \) as follows:

\[
V_{L}[o_{i} + j] = \begin{cases} 
    CF_{l}(P_{(i+k)}) + o_{i} & (i = 0) \\
    CF_{l}(P_{(i+k)}) + Rank_{l}(P_{(i+k)}) + o_{i} & (1 \leq j \leq D) \\
    0 & (0 \leq i \leq M)
\end{cases}
\]

where \( o_{i} = (j - 1)(M + 1) \) is an offset and \( k \) is an index which is initialized by -1 and incremented by 1 in each iteration of the recursive

---

**Algorithm 2.** The detailed description of PBWT-sec finding a set-longest match at position \( t \).

- **Public input:** Problem size \( M \& N \); alphabet \( Σ = \{0, 1, \ldots, |Σ| - 1\} \), start position \( t \in \{1, \ldots, N\} \)
- **Private input of user:** A query sequence \( S \) of length \( t \)
- **Private input of server:** PBWT matrix \( P \in \mathbb{R}^{M \times N} \)

0. (Key setup of cryptosystem) User generates key pair \((pk, sk)\) by key generation algorithm KeyGen for additive-homomorphic cryptosystem and sends public key \( pk \) to server.

1. (User initialization) Set initial interval \( [f = 0, g = M] \).

2. (Recursive search)

   **Initialize** query and position index: \( i \leftarrow 1; k \leftarrow t - 1 \)

   while \((i \leq t)\) do
   a. (Query entry) The user performs the following steps:
      - Prepares next query: \( Enc(q_{i}) \leftarrow PrepQuery(f, M + 1) \)
      - Enc(q_{i}) \leftarrow PrepQuery(g, M + 1)
      - Sends \( Enc(S[i]), Enc(q_{i}) \) to the server.

   b. (Search) The server performs the following steps:
      - Compute look-up tables for all \( c \in Σ \):
         \[
         \nu_{i}[j] = \begin{cases} 
            CF_{l}(P_{j}) & (j = 0) \\
            CF_{l}(P_{j}) + Rank_{l}(P_{j}, i) & (1 \leq j \leq M)
\end{cases}
\]
      - Obtain random values \( r^{(f)} \), \( r^{(g)} \)
      - Set \( r^{(f)} = r^{(g)} = 0 \) iff. \( i = 0 \)
      - Compute next possible intervals for all \( c \in Σ \):
         \[
         \begin{align*}
         e^{(f)}(j) &= \text{ROT}(Enc(q_{j}), \nu_{i}[j], r^{(f)}; M) \\
         e^{(g)}(j) &= \text{ROT}(Enc(q_{j}), \nu_{i}[j], r^{(g)}; M)
\end{align*}
\]
      - Randomize return values except for user’s target interval by computing the following for all \( c \in Σ \)
      - Generate temporary random values \( r_{o}, r_{l} \)
      - \( e^{(f)}(j) \leftarrow e^{(f)}(j) \oplus Enc(r_{o} \times (S[j] - c)) \)
      - \( e^{(g)}(j) \leftarrow e^{(g)}(j) \oplus Enc(r_{l} \times (S[j] - c)) \)
      - Compute an encrypted flag showing if match is longest
         \[
         d^{(f)} = \begin{cases} 
            0 & (i = 0) \\
            0 & (1 \leq j \leq D) \\
            1 & (0 \leq i \leq M)
\end{cases}
\]
      - Store random values \( r^{(f)} = r^{(g)} = r_{o} \)
      - Send \( d^{(f)} \), \( e^{(f)}(i) \) to the user
   c. (Decryption of encrypted flag and randomized interval)

   The user performs the following steps:
   if \((Dec(d) = 0)\)
      Sends decoy queries to server until \( i = t \)
      Reports result \( S[1, \ldots, i - 2] \)
   else
      Computes \( f \leftarrow Dec(e^{(f)}(0)) \), \( g \leftarrow Dec(e^{(g)}(0)) \),
   end if
   \( i \leftarrow i + 1, k \leftarrow k + 1 \)
end while
search. Note that \((V_v[t_0], \ldots, V_v[t_0 + M])\) corresponds to \(v_i\) for \(t_i\)th column. The algorithm for the Model 3 is designed by replacing the lookup tables \(v_i\) by \(V_v\) (see Step 2a, item 1 in Algorithm 2) and initializing \(f\) and \(g\) by \(o_v\) and \(o_v + M\), respectively, where \(t = t_k\) (see Step 1 in Algorithm 2). As a result the tables get \(D\) times larger which has an impact on computing requirements and data transfer size (see Section 3.7). We therefore suggest using this algorithm for small \(D\).

3.5 Reducing communication size

As we will describe in the Complexity analysis in the following section, the PBWT-sec algorithm using standard OT requires \(O(M|\Sigma|)\) in communication size per iteration in the best case, which makes the core algorithm less practical. We propose to use an algorithm for sub-linear-communication OT (SC-OT) proposed in Zhang et al. (2013). Using this approach we can reduce the communication size of PBWT-sec to \(O(\sqrt{M|\Sigma|})\) (best case). Here, we only outline the key ideas of SC-OT and its adaptation of PBWT-sec. In the SC-OT, the one encodes the position \(i\) by a two dimensional representation: \(t_0 = i/\sqrt{|\Sigma|}, t_1 = i \mod \sqrt{|\Sigma|}\), where \(\lfloor \cdot \rfloor\) denotes the ceil of the argument. The user sends \(Enc(t_0)\) and \(Enc(q)\) to the server, where

\[
Enc(q) = (Enc(q_0 = 0), \ldots, Enc(q_M = 1), \ldots, Enc(q_{\sqrt{|\Sigma|} - 1} = 0)).
\]

The server obtains random values \(r_k, k = 0, \ldots, \sqrt{|\Sigma|} - 1,\) and computes

\[
c_k = \bigoplus_{i=0}^{\sqrt{|\Sigma|} - 1} (v_i \times \sqrt{|\Sigma|} + i) \odot Enc(q_i) \odot (r_k \odot Enc(t_0 - k)),
\]

and sends \(c = (c_0, \ldots, c_{\sqrt{|\Sigma|} - 1})\) to the user. The user knows the result by the decryption: \(Dec(c_k)\). Note that \(Enc(t_0 - k) = Enc(0)\) iff. \(t_0 = k\), therefore the randomness of \(c_i\) becomes a random value when \(i \neq t_0\).

In order to apply bit-rotation technique naturally to SC-OT, the server needs to return \(v[i, i]\) in the same two dimension representation. The key idea here is that the server creates \(v_0\) and \(v_1\) where \(v_0[i] = v[i]/\sqrt{|\Sigma|}\) and \(v_1[i] = v[i] \mod \sqrt{|\Sigma|}, i = 0, \ldots, N - 1,\) and searches on both \(v_0\) and \(v_1\). Similar to the linear communication size function ROT, the removable random factors are added to server’s returns. More details on SC-OT is given in Section S1. The complete algorithm for privacy-preserving search based on SC-OT is given in Supplementary and Algorithm S2.

3.6 An exhaustive baseline algorithm

There are a few related works in regard to finding a DNA substring match (Baldi et al., 2011; Cristofaro et al., 2013), however, the goal of PBWT-sec is to find the set-longest prefix match from a set of aligned sequences while those works aim to find a fixed-length approximate substring match between two sequences. Therefore, we will compare our algorithm with a baseline algorithm which can find the set-longest prefix match on the basis of exhaustive enumeration of \(k\)-mers. This baseline algorithm serves as a proxy for the other conceptually similar algorithms.

In order to identify the match, the user queries the server about the presence of a \(k\)-mer. Here, the server stores all \(k\)-mers, there are \(O(|\Sigma|^k)\) of them, and we use SC-OT. Such a strategy is efficient for short queries as \(|\Sigma|^k\) is not too large. However, the resource requirements will be dominated by queries for large \(k\) and the algorithm quickly gets intractable.

### Table 1. The summary of the time, communication and space complexities of PBWT-sec (CP) and an exhaustive method (EX) for the length of query and is the alphabet size

<table>
<thead>
<tr>
<th>Time</th>
<th>Communication</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>CP (user)</td>
<td>(O(t</td>
<td>\Sigma</td>
</tr>
<tr>
<td>CP (server)</td>
<td>(O(t</td>
<td>\Sigma</td>
</tr>
<tr>
<td>EX (user)</td>
<td>(O(D</td>
<td>\Sigma</td>
</tr>
<tr>
<td>EX (server)</td>
<td>(O(D</td>
<td>\Sigma</td>
</tr>
</tbody>
</table>

Both algorithms use SC-OT. \(M\) is the number of haplotype sequences (server side), \(D\) is the number of queried positions (including \(D - 1\) decoy positions to conceal the query position), \(t\) is the length of query and \(|\Sigma|\) is the alphabet size.

3.7 Complexity

Most of the computing and transfer on server and user side is related to the encryption/decryption and the computational cost of the search is negligible. While PBWT requires essentially \(O(1)\) to update the intervals per iteration, PBWT-sec needs to conceal the query and requires \(M|\Sigma|\) operations on the server, where \(M\) is the number of sequences in the database and \(|\Sigma|\) is the size of the alphabet. When multiple queries are performed at the same time, i.e. \(D > 1\), the effort increases linearly in \(D\), i.e. the server side compute effort is \(O(MD|\Sigma|)\) per iteration. When using SC-OT, the communication size and effort for the user is \(O(\sqrt{MD|\Sigma|})\) (see Section 3.5 and Supplementary Section S1 for details).

Table 1 summarizes the time, data transfer overhead and space complexities of the PBWT-sec, where the server’s PBWT is \(M \times N\) matrix consisting of a set of alphabet letters \(\Sigma\) and the user’s query length is \(t\) and the number of queries positions is \(D\) (including \(D - 1\) decoy positions; see Section 3.4 for details). For the purpose of comparison, we consider the method outlined in Section 3.6 that achieves the same security and utility as PBWT-sec. Since the complexity of the exhaustive approach is exponential to the query length, its performance deteriorates quickly for long matches. On the other hand, the time and data transfer overhead complexity of the PBWT-sec are linear and sub-linear to the query length, which enables the user to find a long match efficiently.

3.8 Security notion

In this paper, we assume the security model called Semi-honest model where both parties follow the protocol, but an adversarial one attempts to infer additional information about the other party’s secret input from the legally obtained information. The semantic security of the encryption scheme used in the protocol (see Section 3.1) implies immediately that the server cannot infer any information about the user’s query \(q\) during the protocol. Also, the user cannot infer any information about server’s return except for the result.

Another security model is called Malicious model where an adversarial party cheats even in the protocol (e.g. by inputting maliciously chosen invalid values) in order to illegally obtain additional information about the secret. Here we briefly describe one example of an illegal access based on the Malicious model. In our protocol, the user needs to create a bit vector \(q\) of \(N\) that includes a bit that is 1 and the rest of the \(N - 1\) bits are 0. If the malicious user creates a non-bit vector:

\[
q_k = \begin{cases} 
0 & (k \neq i \land k \neq j) \\
1 & (k = i) \\
x & (k = j)
\end{cases}
\]

where \(x\) is a large integer, the server returns \(c = Enc(v[i] + x \cdot v[j])\). When \(x\) is larger than any element of \(v\), the user can infer \(v[i]\) by
4 Experiments

In this section, we evaluate the performance of the proposed method on the datasets created from the chromosome 1 data from the 1000 Genomes Project phase 1 data release which consists of 2184 aligned haploid genomes (The 1000 Genomes Project Consortium, 2012). In our experiments and as in Durbin (2014), we used alleles having SNPs, but we did consider indel variants. We used all 2184 genomes of original data for all the experiments.

We implemented the proposed algorithm in C++ based on an open source C++ library of elliptic curve ElGamal encryption provided by AIST. Our implementation supports communication over the network. We used the standard parameters called secp192k1 (SECG curve over a 192-bit prime field), according to the recommendation by The Standards for Efficient Cryptography Group. For comparison, we also implemented an exhaustive baseline method (see Section 3.6) that achieves the same security and utility as PBWT-sec. In order to perform a fair comparison, both PBWT-sec and the exhaustive method used the same SC-OT module where computation of $c_0$ (Algorithm 1) is simply parallelized by OpenMP.

In the first experiment, the user selected a true start position together with 49 decoys (see Section 3.4 for details), and both PBWT-sec and the exhaustive method were run with the same computational setting: the user used a single thread of a laptop computer equipped with an Intel Core(TM) i7 3.00 GHz CPU and 16 GB memory, and the server used more than eight threads of another laptop equipped with an Intel Core(TM) i7 2.60 GHz CPU (four cores with hyper-threading) and 8 GB memory. Those two computers communicated over the network.

Figures 6 and 7 show run time and data transfer overhead of PBWT-sec and of the exhaustive method. The observed run time and data transfer size of PBWT-sec is linear in the query length, while that of the exhaustive approach is exponential. For query lengths larger than 30 bit, the computation of the exhaustive method did not finish within 24 h. These results fit the theoretical complexity described in Section 3.7. We also evaluated performance of the runtime of PBWT-sec when the user selected 0, 4, 9, 14, and 49 additional decoy positions. The search with a typical query of length 25 SNP positions and no decoy required no more than 15.5 s including communication overhead (Table 2).

The user’s run time of PBWT-sec is relatively small, making it suitable for a practical case where computation power in a server side is generally stronger than that of user side. Since the memory usage of PBWT-sec does not depend on query length, it uses less
than 60 MB while that of the exhaustive method exponentially increases according to the query length and required 6 GB when the query length is 25 bit.

Although the exhaustive method is efficient for short queries, we consider that PBWT-sec is more practical when taking into account that the bit length of a unique substring for a human genome is greater than 31 bits. Moreover, since there are large linkage blocks, even queries with more than 100 bits would not always lead to unique matches in the 1000 genomes data. Hence, the exhaustive search strategy would either not always be able to return a unique match or would be very inefficient. The proposed iterative privacy-preserving technique is efficient also for long queries.

In the second experiment, we evaluated the performance of the run time of PBWT-sec on a compute node equipped with four CPU sockets (Intel Xeon 2.40 GHz CPU; total of 32 cores with hyper-threading). In this experiment, the user also selected 0, 4, 9, 14 and 49 additional decoy positions. For environmental reasons, we did not perform communication over the network and the data was transferred by file I/O which is also included in run time.

Although the current implementation is a prototype and there is room for improvement in terms of parallelization, the server’s run time was at an acceptable level in practical configurations (Table 3). We note, that with improvements in parallelization, the server run time may be reduced to 3–4 s.

### 5 Conclusion
In this paper, we have proposed a novel approach for searching genomic sequences in a privacy-preserving manner. Our approach combines an efficient data structure that can be used for recursive search and a novel approach for recursive oblivious transfer. It achieves high utility and has strong security features and requires acceptable compute and communication resources.

The developed novel algorithm can find the longest match between a query and a large set of aligned genomic sequences indexed by PBWT. We implemented the algorithm and tested on the dataset created from the 1000 Genomes Project data (The 1000 Genomes Project Consortium, 2012). Compared to an exhaustive baseline approach, our algorithm, named PBWT-sec, was orders of magnitude more efficient both in run time and data transfer overhead for practical query sizes. When the prototype program was run on laptop machines, the total run time including communication time over the network was 15.5 s for searching on 2184 genomes without concealing the query position. Searches with a concealed query position using a compute node took between 18.6 and 133 s depending on the level of privacy.

As the original data structure supports many useful search options such as wild card search and set maximal search, PBWT-sec could also support those options by using the same techniques used in the original structures in combination with cryptographic techniques, including OT. Moreover, the approach could be easily applied for BWT and has a potential to be applied for other recursively searchable data structures.

To the best of our knowledge, the proposed algorithm is the first that allows set-maximal search of genomic sequences in a privacy-preserving manner for user and database. We note that the implementation can still be improved and the overall run time can likely be reduced to not more than a few seconds per query. This would make it practical to use our approach in a genomic Beacon (see GA4GH’s Beacon Project) that would allow the privacy-preserving search for combinations of variants. It also appears practical to use our approach to enable search by a user that has access to his/her genomic sequence and would like to query the database, for instance, for information related to disease risk without sharing this information with anybody. Finally, the algorithm can also be used to facilitate sharing of genetic information across institutions and countries in order to identify large enough cohorts with a similar genetic backgrounds. This is in spirit of the mission of the Global Alliance for Genome and Health.

### Acknowledgements
We are thankful to Stephanie L. Hyland for proof-reading the manuscript. We would also like to acknowledge an encouraging discussion with Richard Durbin. We are grateful to the reviewers for their detailed comments that significantly improved the manuscript.

### Funding
We gratefully acknowledge funding from AIST (to K.S.), Memorial Sloan Kettering Cancer Center (to G.R.) and NIH (grant 1R01CA176785-01A1). This study was also supported by the Japan-Finland Cooperative Scientific Research Program of Japan Science and Technology Agency (JST; to K.S.).

### Conflict of Interest
None declared.

### References


