Privacy Preserving Substring Search Protocol with Polylogarithmic Communication Cost

Nicholas Mainardi
Politecnico di Milano – DEIB
Milano, Italy
nicholas.mainardi@polimi.it

Alessandro Barenghi
Politecnico di Milano – DEIB
Milano, Italy
alessandro.barenghi@polimi.it

Gerardo Pelosi
Politecnico di Milano – DEIB
Milano, Italy
gerardo.pelosi@polimi.it

ABSTRACT

The problem of efficiently searching into outsourced encrypted data, while providing strong privacy guarantees, is a challenging problem arising from the separation of data ownership and data management typical of cloud-based applications. Several cryptographic solutions allowing a client to look-up occurrences of a substring of choice in an outsourced document collection have been publicly presented. Nonetheless, practical application requirements in terms of privacy, security and efficiency actively push for new and improved solutions. We present a privacy-preserving substring search protocol exhibiting a sub-linear communication cost, with a limited computational effort on the server side. The proposed protocol provides search pattern and access pattern privacy, while its extension to a multi-user setting shows significant savings in terms of outsourced storage w.r.t. a baseline solution where the whole dataset is replicated. The performance figures of an optimized implementation of our protocol, searching into a remotely stored genomic dataset, validate the practicality of the approach exhibiting a data transfer of less than 200 kiB to execute a query over a document of 40 MiB, with execution times on client and server in the range of a few seconds and a few minutes, respectively.

CCS CONCEPTS

• **Security and privacy → Privacy-preserving protocols**; **Management and querying of encrypted data**; *Security protocols.*

ACSAC '19, December 9–13, 2019, San Juan, PR, USA

ACM ISBN 978-1-4503-7628-0/19/12…$15.00
https://doi.org/10.1145/3359789.3359842

KEYWORDS

Secure substring search, Cryptography, Homomorphic encryption, Privacy-preserving protocol

ACM Reference Format:
Nicholas Mainardi, Alessandro Barenghi, and Gerardo Pelosi. 2019. Privacy Preserving Substring Search Protocol with Polylogarithmic Communication Cost. In *2019 Annual Computer Security Applications Conference (ACSAC '19), December 9–13, 2019, San Juan, PR, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3359789.3359842

1 INTRODUCTION

Current trends and innovations in the information technology scenario have prompted users and organizations to store a growing amount of sensitive data in a third party located cloud, beyond their direct control. In such a scenario, companies rely on the cloud for data storage and management, profiting from low storage costs and high availability, while end-users enjoy ubiquitous availability of data, which can also be accessed via mobile devices. However, such benefits come with a loss of control of the data itself and the concrete possibility of privacy and security information leakages.

In this paper, we consider the popular cloud computing model composed by three entities: the data owner, the cloud server and the users authorized to access the remotely stored data. The data owner stores the data on the cloud server and authorizes the users to issue specific queries on the outsourced data. To protect the data, the data owner encrypts the data before outsourcing them and shares the decryption keys with the authorized users only. However, data encryption is a major hindrance to perform data access operations, such as searching for a given pattern, with the same efficiency provided by the ones acting on data stored and maintained on premise. Therefore, there is a pressing need for effective solutions enabling a set of querying functionalities on encrypted data, possibly by multiple users, preserving the confidentiality of the searched information even against the service (storage) provider itself.

Table 1: Comparison of existing privacy-preserving substring search protocols with our protocol. In the table, n denotes the size of the document collection, m the length of the queried substring q, and o_q the number of occurrences of q found
† the asymptotic cost in [16] hides a large constant factor C, e.g., $C \geq 16 \times 10^6$, for providing 80-bit security parameters

Protocol	Communication Cost	Server Cost	Search & Access Pattern Privacy	Data Owner Off-line	Extension to multi-user	Adversary
[30]	$O(n)$	$O(nm)$	✓	✓	×	Semi-honest
[25]	$O((m+o_q)\sqrt{n})$	$O((m+o_q)n)$	✓	✓	×	Semi-honest
[16]	$O(C(m+o_q)\log(n))^\dagger$	$O(C(m+o_q)n\log n)^\dagger$	✓	✓	×	Semi-honest
[10]	$O(m+o_q)\log(n)$	$\Omega(\frac{n}{m}+o_q)\log(n)$	×	×	✓	Malicious
[22]	$\Omega(m\log^5(n)+o_q\log^2(n))$	$\Omega(m\log^5(n)+o_q\log^2(n))$	✓	✓	×	Semi-honest
Ours PPSS	$O((m+o_q)\log^2(n))$	$O(m+o_q)n$	✓	✓	✓	Semi-honest

Problem Statement. A data owner outsources a set of documents $\mathbf{D} = \{D_1, \ldots, D_z\}$, $z \geq 1$, encrypted with a cryptographic primitive of choice, where each document is a sequence of symbols (*string*) over an alphabet Σ with length $\texttt{len}(D_i)$ and $n = \sum_{i=1}^{z} \texttt{len}(D_i)$. In addition, the data owner builds an indexing data structure to enable the search for any substring $q \in \Sigma^*$, $\texttt{len}(q) = m \geq 1$, over $\mathbf{D}$. A query for a substring q will yield, for each document D_i, $1 \leq i \leq z$, the set of positions, S_i, where an *occurrence* of q appears. Along with the collection of documents $\mathbf{D}$, the data owner stores on the remote storage a *privacy-preserving representation* of the aforementioned indexing data structure allowing authorized clients to use the substring search functionality with the cooperation of the service provider. The main challenge in this scenario is reducing the information learnt by an adversary (including the service provider) to the knowledge of the size of the outsourced document collection, the size of the substring, the one of the indexing data structure and the total number of occurrences matching the query at hand. We remark that the private retrieval of the matching documents from the remote storage is out of scope in the problem addressed by this paper, as this functionality can be achieved by hinging upon existing cryptographic primitives such as Oblivious RAMs (ORAMs) [27] or Private Information Retrieval (PIR) protocols [19].

Adversary and Security Model. In a real-world deployment of a privacy-preserving substring search solution, the notion of *semi-honest* adversary fits well entities that trustworthy follows the protocol specification, although being curious about any other additional information that may be inferred with a polynomial computation effort about the confidential data as well as the *access pattern*s or the *search pattern*s on the remote data storage. Informally, a search pattern refers to the understanding of how similar distinct queries are (e.g., if they share a common prefix or only some non consecutive symbols), while access pattern refers to the understanding of the positions of replicas of the queried substring in $\mathbf{D}$.

Prior Art Approaches. The seminal work on searching over data in an encrypted state [26] (a.k.a. *searchable encryption* schemes) as well as many of the subsequent improvements in terms of computational and communication resources [3, 8], relies on pre-registering searchable keywords, and does not allow free form searching over the encrypted data. Such a limitation is overcome by *substring searchable encryption* schemes [6, 15, 18, 28]. These schemes exhibit computation/communication complexities linear or quadratic in the length of the searched substring, with different server side storage savings and assumptions on the adversary capabilities. While these solutions coped with the problem of substring search, the works in [6, 18, 28] do not provide protection of search and access pattern, while the information leakage shown in [15] is not explicitly framed as a search or access pattern one. The importance of protecting both the search and access pattern is demonstrated by [5, 24], where the authors describe the recovery of either a significant portion of the documents in the collection $\mathbf{D}$ or the content of the queried substrings by combining search and access pattern leakages with public information related to the application domain itself.

Contributions. Substring searchable encryption schemes [6, 15, 18, 28] employ symmetric-key or order-preserving cryptographic primitives, obtaining good performance figures in terms of required bandwidth, computational power, and storage demands on both clients and servers. However, they do not take into account the information leakage coming from the observation of both search and access patterns.

The higher security guarantees resulting from the inclusion of such leakages in the security model comes along with the usage of cryptographic primitives with higher computational complexity. We refer to substring search schemes preserving search and access pattern confidentiality as *privacy-preserving substring search* (PPSS) protocols.

In the following, we describe the first multi-user PPSS protocol secure against semi- honest adversaries, with an $O(m \log^2 n)$ communication cost between client and service provider. We combine the working principles of the Burrows Wheeler Transform (BWT) [4] (as a method to perform a substring search) with a single server private information retrieval (PIR) protocol; specifically, we choose the PIR proposed by Lipmaa in [19], which is based on the generalized Pailler homomorphic encryption scheme [9], because of its limited communication cost. The solution exhibits an $O(m \log^4 n)$ computational cost and requires $O(\log n)$ memory on the client side, while the computational and storage demands on the service provider side amount to $O(mn)$ and $O(n)$, respectively. In a multi-user scenario, our PPSS protocol allows distinct and simultaneous queries on the same document collection, run by multiple clients without any interaction with the data owner and among themselves. Our multi-user approach avoids to replicate the outsourced document collection for each authorized client, limiting the additional memory required by each query to $O(\log^2 n)$ cells.

2 RELATED WORK

In [30], the authors describe a PPSS protocol to establish if a given substring is present in the outsourced document collection with an $O(n)$ communication cost and an impractical $O(n)$ amount of *cryptographic pairing* computations required at the client side for each query. Shimizu *et. al.* in [25] described how to use the Burrows Wheeler Transform (BWT) [4] and Pailler's additive homomorphic encryption (AHE) scheme [23] to effectively retrieve the occurrences of a substring. The main drawback of the scheme lies in the significant communication cost: each query needs to send $O(m\sqrt{n})$ ciphertexts from client to server. Such a cost was reduced by Ishimaki *et. al.* [16] to $O(m \log(n))$, at the price of employing a fully homomorphic encryption (FHE) scheme [13], making their solution unpractical. Indeed, FHE schemes generally require ciphertexts bigger than the ones exhibited by Pailler AHE scheme, introducing a significant constant factor in the communication cost. Moreover, the computational cost for the server is $O(mn \log(n))$, which also hides a large constant overhead (about 10^6) required to compute on FHE ciphertexts.

A multi-user protocol, preserving only the search pattern confidentiality and with communication cost linear in the size of the searched substring is described in [10]. The main drawbacks of this solution are the need for the client to interact with both the data owner and the server to perform a query, and the constraint that only substrings of a fixed length, which must be decided when the privacy preserving indexing data structure to be outsourced is built, can be queried, in turn limiting the impact of the solution.

Finally, the suffix-array based solutions proposed by Moataz *et. al.* in [22] guarantee the confidentiality of the content of both the substring and the outsourced data, as well as the privacy of the access pattern and the search pattern observed by the server. The access pattern to the outsourced indexing data structures is concealed by employing an ORAM data structure [27] – which is specifically designed to obliviously access a remote data storage without leaking search and access patterns. The asymptotic complexities of the protocol showed in [22] mainly depends on the size of each document being negligible w.r.t. the total number of them (denoted as z). Indeed, it exhibits $O(m \log^3(z))$ communication and computation complexities, assuming that the size of each document is $O(\log^2(z))$. If the size of each document is not negligible w.r.t. their total number, the computational and communication cost of the solution increase proportionally to the size n of the document collection, by (at least) a factor $\log^2(n)$. We show in Table 1 a concise comparison between our PPSS protocol and the state-of-the-art solutions we have just described.

3 PRELIMINARIES

In the following, we describe the basic algorithms and cryptographic primitives employed in this work, detailing their features and pointing out the properties needed to define our privacy-preserving substring search (PPSS) protocol.

3.1 Substring Search with BWT

The Burrows-Wheeler Transform (BWT) [4] was designed to compute a transformation of a given text (string), s, to make it more compressible by run-length encoding methods. It computes an invertible permutation of the string at hand, $L = \texttt{BWT}(s)$, that can be efficiently compressed if letters of the alphabet Σ have repetitions in the string s, regardless of their position. The BWT computation has a time complexity that is linear in the string length n.

Besides its usefulness as a preprocessing for compression, the BWT enables a very efficient substring search algorithm when combined with the so-called *suffix array*, i.e., the array of starting positions of all sorted suffixes of a string [11]. The substring search algorithm has a linear time complexity in the length of the substring to be searched for, and requires only a limited storage overhead. Consider a string s with length n defined over an alphabet $\Sigma \cup \{\$\}$, where the end-of-string delimiter \$ precedes any character in Σ, for any order relation of choice (e.g., the alphabetical one). We denote with an increasing numerical subscript the occurrences of the same character in s (e.g., a_1, a_2 will denote the first and second occurrence of a in s) and define as *index* of a substring in s the position of its leading character in the original string, counting from 1 onwards.

String	Index
a1 l1 f1 a2 l2 f2 a3 $	1
l1 f1 a2 l2 f2 a3 $ a1	2
f1 a2 l2 f2 a3 $ a1 l1	3
a2 l2 f2 a3 $ a1 l1 f1	4
l2 f2 a3 $ a1 l1 f1 a2	5
f2 a3 $ a1 l1 f1 a2 l2	6
a3 $ a1 l1 f1 a2 l2 f2	7
$ a1 l1 f1 a2 l2 f2 a3	8

sorting →

F	String	L	SA
$	a1 l1 f1 a2 l2 f2 a3	8	
a3	$ a1 l1 f1 a2 l2 f2	7	
a2	l2 f2 a3 $ a1 l1 f1	4	
a1	l1 f1 a2 l2 f2 a3 $	1	
f2	a3 $ a1 l1 f1 a2 l2	6	
f1	a2 l2 f2 a3 $ a1 l1	3	
l2	f2 a3 $ a1 l1 f1 a2	5	
l1	f1 a2 l2 f2 a3 $ a1	2	

Figure 1: Burrows Wheeler Transform L and Suffix Array SA of the string `alfalfa`

As shown in Figure 1, taking as an example s = `alfalfa$`, first the BWT computation mandates to build a list of $n+1$ strings obtained performing a cyclic shift of s by all the amounts in $\{0, 1, \ldots, n\}$. Each of these $n+1$ strings contain the suffixes of s, represented by the portion of the shifted string preceding the string delimiter \$, whose indexes are also computed and stored. The list of $n+1$ shifted strings is then sorted lexicographically, and the BWT of s, $L = \texttt{BWT}(s)$, is derived concatenating the trailing characters of each string in the sorted list. The suffix array, SA, associated to L is built by storing the indexes of the cyclic shifts of s in the sequence defined by the sorting step.

Given L and SA, the inverse BWT transform, allows to reconstruct the original string $s = \texttt{BWT}^{-1}(L)$ and also to lookup for the occurrences of a given substring. Note that, the string F, i.e., the concatenation of the leading characters of the sorted list of suffixes employed to compute the BWT (in blue in Figure 1) can also be obtained concatenating $s[SA[j]]$ for all $1 \leq j \leq n+1$, i.e., $F[j] = s[SA[j]]$. We outline some useful properties of the strings L and F in the following statement:

Theorem 3.1. *Consider a string s , with length $n+1$, over the alphabet $\Sigma \cup \{\$\}$ and \$ as trailing character. Denote the BWT of s as $L = \texttt{BWT}(s)$, its suffix array as SA and as F the string $F[j] = s[SA[j]]$ with $1 \leq j \leq n+1$. Denoting the position of a character $c \in \Sigma$ in F and L as $pos_F(c)$ and $pos_L(c)$, respectively, the following properties hold:*

(1) *Characters in the same position in L and F are consecutive in the original string s: $\forall c \in s(pos_L(c) = pos_F(succ_s(c)))$.*

(2) *All the occurrences of the same character appear in the same order in both F and L, i.e., for each pair of occurrences $\langle c_1, c_2 \rangle$ of the same character: $pos_F(c_1) < pos_F(c_2) \Leftrightarrow pos_L(c_1) < pos_L(c_2)$.*

(3) *Consider two occurrences of the same character c in L, denoted by c_1, c_2, where $pos_L(c_1) < pos_L(c_2)$. If no occurrence c_3 of c such that $pos_L(c_1) < pos_L(c_3) < pos_L(c_2)$ exists, then $pos_F(c_2) = pos_F(c_1) + 1$.*

Proof. (1) follows directly from BWT construction, as the characters $F[i]$, $L[i]$, $1 \leq i \leq n+1$, are consecutive characters in one of the cyclic shifts of the original string. Concerning (2), we observe that since F is constructed by concatenating the first characters of the sorted cyclic shifts of s, then $pos_F(c_1) < pos_F(c_2) \Leftrightarrow pos_F(succ_s(c_1)) < pos_F(succ_s(c_2))$. Due to (1), $pos_L(c) = pos_F(succ_s(c))$, thus $pos_F(succ_s(c_1)) < pos_F(succ_s(c_2)) \Leftrightarrow pos_L(c_1) < pos_L(c_2)$, which proves (2). Finally (3) is proven by contradiction. Assume there is no c_3 such that $pos_L(c_1) < pos_L(c_3) < pos_L(c_2)$ with $pos_F(c_2) - pos_F(c_1) \neq 1$. As F contains a sorted sequence of characters in s, having $pos_F(c_2) > pos_F(c_1)+1$ implies the existence of a further occurrence, c_3, between the two, $pos_F(c_2) > pos_F(c_3) > pos_F(c_1)$. Property (2) implies $pos_L(c_2) > pos_L(c_3) > pos_L(c_1)$, contradicting the hypothesis. □

Relying on the previous theorem, Algorithm 1 computes the number of occurrences of a substring q with length m in a string s with n characters, taking as input three data structures and the substring to be searched.

The first data structure replaces L, the BWT of s, with a $(|\Sigma|+1) \times (n+1)$ integer matrix M indexed by a character c in $\Sigma \cup \{\$\}$ and an integer i, storing in each cell $M[c][i]$ the number of occurrences of c in the string $L[1], \ldots, L[i]$. The second data structure is a dictionary `Rank` of size $|\Sigma|+1$, with pairs $\langle c, l \rangle$, where $c \in \Sigma$, and l, $0 \leq l \leq n+1$, is the number of characters in s alphabetically smaller than c. The third one is the suffix array SA of s.

The substring search procedure looks for the characters in q starting from the last one, i.e., $q[m]$, moving backwards towards $q[1]$. In the algorithm, a run of equal characters in F is tracked by $\alpha+1$ and β which denote the positions of the first and the last of them in F. Starting from $q[m]$, and the corresponding values for α and β (lines 1–2), the algorithm looks for all the occurrences of $q[m-1]$ followed by $q[m]$ in s (lines 4–6) to update $\alpha+1$ and β with the first and last positions in F of the leading character of the substring $q[m-1, m]$. In particular, all the repetitions of $q[m-1]$ among the predecessors of $q[m]$ in $s = \texttt{BWT}^{-1}(L)$ coincide with the repetitions of $q[m-1]$ in $L[\alpha+1, \ldots, \beta]$ (property (1) in Thm. 3.1). Denote the first and last repetition of $q[m-1]$ in $L[\alpha+1, \ldots, \beta]$ as $q[m-1]_{\texttt{first}}$ and $q[m-1]_{\texttt{last}}$. Note that, thanks to property (3) in Thm. 3.1, the repetitions of $q[m-1]$ in the unsorted string $L[\alpha+1, \ldots, \beta]$ correspond to the subsequence of consecutive characters in F with positions between $\alpha+1 = pos_F(q[m-1]_{\texttt{first}})$ and $\beta = pos_F(q[m-1]_{\texttt{last}})$.

The value $pos_F(q[m-1]_{\texttt{first}})$ can be obtained adding to the position of the leading character in F (i.e., 1) the number $r = \texttt{Rank}(q[m-1])$ of characters in s smaller than $q[m-1]$ (i.e., the number of characters preceding any repetition of $q[m-1]$ in F), and the number of repetitions of $q[m-1]$ with smaller positions in F than $q[m-1]_{\texttt{first}}$. As by property (2) in Thm. 3.1, the latter quantity equals $M[q[m-1]][\alpha]$ thus, line 5 in Alg. 1 correctly updates α.

Algorithm 1: Substring search

Input: M, matrix representation of the BWT L of a given n-character string s; $M[c][i]$ stores the number of occurrences of the character $c \in \Sigma$ in the string $L[1], \ldots, L[i]$, $1 \leq i \leq n$.
Rank, dictionary of size $|\Sigma|+1$, of pairs $\langle c, l \rangle$, with $c \in \Sigma$, $l = \mathsf{Rank}(c)$, $0 \leq l \leq n+1$ number of chars in s smaller than c.
SA, suffix array with length $n+1$ of the string s;
q, a substring with length $1 \leq m \leq n$.

Output: R_q, set of positions in s with the leading character of every repetition of q.

1 $c \leftarrow q[m]$
2 $\alpha \leftarrow \mathsf{Rank}(c),\ \beta \leftarrow \alpha + M[c][n+1]$
3 **for** $i \leftarrow m-1$ ***downto*** 1 **do**
4 $\quad c \leftarrow q[i],\ r \leftarrow \mathsf{Rank}(c)$
5 $\quad \alpha \leftarrow r + M[c][\alpha]$
6 $\quad \beta \leftarrow r + M[c][\beta]$
7 $R_q \leftarrow \emptyset$
8 **for** $i \leftarrow \alpha + 1$ ***to*** β **do**
9 $\quad R_q \leftarrow R_q \cup \{SA[i]\}$
10 **return** R_q

Analogously, $pos_F(q[m{-}1]_{\mathtt{last}})$ can be obtained by adding to the position of the leading character in F (i.e., 1) the number $r = \mathsf{Rank}(q[m{-}1])$ of characters in s smaller than $q[m{-}1]$ (i.e., the number of characters preceding any repetition of $q[m{-}1]$ in F), and the number of repetitions of $q[m{-}1]$ with smaller positions over F than $q[m{-}1]_{\mathtt{last}}$. By property (2) in Thm. 3.1, the latter quantity equals $M[q[m{-}1]][\beta] - 1$ as the count given by $M[q[m{-}1]][\beta]$ includes also $q[m{-}1]_{\mathtt{last}}$. Thus, Alg. 1 at line 6 correctly updates β.

Note that, in case $q[m{-}1]$ is not in $L[\alpha+1, \ldots, \beta]$, then $M[q[m-1]][\alpha] = M[q[m-1]][\beta]$ thus, α and β are correctly updated to the same value.

At the end of the first iteration of the loop, $\beta{-}\alpha$ amounts to the number of repetitions of the substring $q[m{-}1, m]$ in s. In the next iteration the values $\alpha{+}1$, β are updated with the positions in F of the first and last repetition of the leading character of $q[m{-}2, \ldots, m]$. The algorithm proceeds in such a way to compute during the last iteration the values of $\alpha{+}1$ and β referring to the first and last positions in F of the leading character of the whole substring $q[1, \ldots, m]$ thus obtaining the number of occurrences of q, denoted as o_q, i.e., $o_q{=}\beta{-}\alpha$. Then, exploiting the fact that $F[i]{=}s[SA[i]]$, $1 \leq i \leq n+1$, the set R_q of integers in $SA[j]$ with $\alpha{+}1 \leq j \leq \beta$, includes the position of the leading character of each repetition of q in s. In Alg. 1 lines 7–9 computes R_q following the mentioned observation.

In Alg. 1, the time and space complexities to find the number of repetitions of a substring q with length m amounts, respectively, to $4m{-}1$ memory accesses, i.e., $O(m)$, and $O(|\Sigma|n)$. The computation of the set of positions of the leading characters of repetitions of q in s increases the time complexity up to $O(m + o_q)$.

Substring Search over a Collection of Documents. The problem of finding the repetitions of a substring q with length m over a set of $z \geq 1$ documents $\mathbf{D} = \{D_1, \ldots, D_z\}$, can be solved considering a string s obtained as the ordered concatenation of all documents, each terminated by an end-of-string character i.e.: $s = D_1\$D_2\$ \ldots D_z\$$, and returning a set of pairs $\langle \mathtt{doc}, \mathtt{off} \rangle$, where doc is the identifier of the document where the repetition of q is found, and off is the position of the said replica into the document. Therefore, it is easy to adapt Alg. 1 also to this multi-document scenario. Specifically, Alg. 1 takes as input a matrix M derived from the BWT of s, the dictionary Rank over the alphabet Σ and an augmented suffix array SA storing for each cell $SA[j]$, with $1 \leq j \leq n+1$ and $n = \mathtt{len}(s)$, a pair of values $\langle \mathtt{doc}, \mathtt{off} \rangle$.

Alg. 1 correctly computes the solution by recognizing all the repetitions of q in $D_1, D_2, \ldots, D_z$ separately. Indeed, the interleaving of the end-of-string delimiters with the sequence of documents during the construction of s guarantees that no substring matching across two adjacent documents is considered. Thus, the application of Alg. 1 with a properly prepared input returns a result equivalent to running it separately over each document.

3.2 Cryptographic Building Blocks

Definition 3.2 (Additive Homomorphic Encryption). An additive homomorphic encryption (AHE) scheme is a tuple of four polynomial time algorithms (KeyGen, E, D, Add):

- $(pk, sk, evk) \leftarrow \mathsf{KeyGen}(1^\lambda)$ is a probabilistic algorithm which, given the security parameter λ, generates a public key pk, a secret key sk and a public evaluation key evk used to perform the homomorphic operation.
- $c \leftarrow \mathsf{E}(pk, m)$, denoted also as $\mathsf{E}_{pk}(c)$, is a probabilistic algorithm which, given the public key pk and a plaintext value $m \in \mathcal{M}$, where $\mathcal{M}$ denotes the plaintext space of the scheme, encrypts the message to a ciphertext $c \in C$, where C denotes the ciphertext space.
- $m \leftarrow \mathsf{D}(sk, c)$, denoted also as $\mathsf{D}_{sk}(c)$, is a deterministic algorithm which, given the secret key sk and a ciphertext $c \in C$, recovers the plaintext value $m \in \mathcal{M}$.
- $c_{add} \leftarrow \mathsf{Add}(evk, c_1, c_2)$, the homomorphic-addition primitive, is a deterministic algorithm which, given the evaluation key evk and two ciphertexts $c_1, c_2 \in C$, computes the homomorphic addition of the two ciphertexts, which is a ciphertext $c_{add} \in C$.

For every key (pk, sk, evk) generated by the KeyGen algorithm, the encryption, decryption and homomorphic addition algorithms satisfy the following correctness properties.

Decryption Correctness: $\forall m \in \mathcal{M}(\mathsf{D}_{sk}(\mathsf{E}_{pk}(m)) = m)$

Addition Correctness: $\forall m_1, m_2 \in \mathcal{M}(\mathsf{D}_{sk}(\mathtt{Add}(evk, \mathsf{E}_{pk}(m_1), \mathsf{E}_{pk}(m_2))) = m_1 + m_2)$, where $m_1 + m_2$ represents the addition in the plaintext space $\mathcal{M}$.

An AHE scheme allows to perform another operation HybridMul, which we call *hybrid homomorphic multiplication*, as follows: given a generic ciphertext $c = \mathsf{E}_{pk}(m) \in C$ and an integer $h \geq 1$, HybridMul computes a ciphertext c_{hmul} that is an encryption of $m \cdot h$. Formally:

$$\forall m \in \mathcal{M}, h \geq 1(\mathsf{D}_{sk}(\mathtt{HybridMul}(evk, h, \mathsf{E}_{pk}(m))) = m \cdot h)$$

This operation can be efficiently implemented via a double-and-add strategy which employs $O(\log h)$ homomorphic additions.

Definition 3.3 (Flexible Length Additive Homomorphic Encryption). An AHE scheme is defined as a flexible length additive homomorphic encryption (FLAHE) scheme if it is augmented with an additional parameter $l \geq 1$, called *length*, which specializes the definition of the plaintext and ciphertext spaces, and of the encryption, decryption and homomorphic addition operations, such that:

$$\forall l_1, l_2 \in \mathbb{N}(l_1 < l_2 \Rightarrow C^{l_1} \subset \mathcal{M}^{l_2})$$

where the superscript l_1 (resp. l_2) is employed to specify the plaintext and ciphertext spaces for length l_1 (resp. l_2). Therefore, the expression $C^{l_1} \subset \mathcal{M}^{l_2}$ indicate that ciphertexts in C^{l_1} are valid plaintexts for ciphertexts in C^{l_2} (i.e., a ciphertext in C^{l_1} is a valid output of the decryption algorithm fed with an element of C^{l_2}).

Paillier FLAHE Scheme. Proposed in 1999 [23], it is a public key AHE scheme based on the *Composite Residuosity Class Problem*, which is polynomially reducible to the *Integer Factoring Problem*. The plaintext space of this scheme is $\mathcal{M} = \mathbb{Z}_N$, with N computed as the product of two large primes, while the ciphertext space is $C = \mathbb{Z}^*_{N^2} \subset \mathbb{Z}_{N^2}$, i.e., the subset of all and only elements of $\mathbb{Z}_{N^2}$ with a multiplicative inverse modulo N^2. The key generation algorithm computes the public pk and private key sk, with the public evaluation key $evk = pk$. The Paillier scheme is semantically secure, which intuitively means that it is computationally unfeasible to determine if two ciphertexts encrypt the same plaintext or not. Given the ciphertexts $c_1, c_2 \in \mathbb{Z}_N$, the homomorphic addition is defined as: $\forall m_1, m_2 \in \mathbb{Z}_N(\mathsf{D}_{sk}(\mathsf{E}_{pk}(m_1) \cdot \mathsf{E}_{pk}(m_2) \bmod N^2) = m_1 + m_2 \bmod N)$.

Therefore, the result of an hybrid homomorphic multiplication HybridMul is obtained as an exponentiation of a ciphertext c to an integer. It can also be conceived as the encryption of the product of two plaintexts:

$$\forall m_1, m_2 \in \mathbb{Z}_N(\mathsf{D}_{sk}(\mathsf{E}_{pk}(m_1)^{m_2} \bmod N^2) = m_1 \cdot m_2 \bmod N)$$

By combining the homomorphic addition and the HybridMul operation, the Paillier scheme allows to perform a *dot product* between a cell-wise encrypted array, denoted as $\langle A \rangle$, and an unencrypted one B, as:

$$\mathsf{D}_{sk}\left(\prod_{i=1}^{n}(\langle A \rangle[i])^{B[i]} \bmod N^2\right) = \sum_{i=1}^{n} A[i] \cdot B[i] \bmod N$$

An FLAHE variant is described in [9] where the plaintext and ciphertext spaces are specialized on the size of their elements as follows: $\mathcal{M}^l = \mathbb{Z}_{N^l}$, and $C^l = \mathbb{Z}^*_{N^{l+1}}$.

Given two lengths l_1, l_2, with $l_1 < l_2$, the hybrid homomorphic multiplication HybridMul between a ciphertext in $\mathbb{Z}^*_{N^{l_2+1}}$ and one in $\mathbb{Z}^*_{N^{l_1+1}}$, equals the encryption of the product between the plaintext value in $\mathbb{Z}_{N^{l_2}}$ (enciphered by the first operand) and the latter ciphertext (being $\mathbb{Z}^*_{N^{l_1+1}} \subset \mathbb{Z}_{N^{l_2}}$). Indeed, $\forall m_1 \in \mathbb{Z}_{N^{l_1}}, m_2 \in \mathbb{Z}_{N^{l_2}}$:

$$\mathsf{D}^{l_2}_{sk}\left(\mathsf{E}^{l_2}_{pk}(m_2)^{\mathsf{E}^{l_1}_{pk}(m_1)} \bmod N^{l_2+1}\right) = m_2 \cdot \mathsf{E}^{l_1}_{pk}(m_1) \bmod N^{l_2}$$

where the superscript l_1 (resp. l_2) denotes that the encryption and decryption operations are performed for plaintext and ciphertext spaces $\mathcal{M}^{l_1}$ and C^{l_1} (resp. $\mathcal{M}^{l_2}$ and C^{l_2}). This homomorphic operation is at core of the *Private Information Retrieval* (PIR) protocol introduced by Lipmaa in [19], which, in turn, is an important building block of our PPSS protocol.

3.3 Lipmaa's PIR Protocol

Given an array A with n elements, each encoded with ω bits, stored on a remote server, a PIR protocol allows a client to retrieve the element in the h-th cell, $0 \leq h \leq n-1$, with the server being able to determine which element was selected with probability at most $\frac{1}{n}$.

A draft description of the PIR in [19] assumes that both the client and the server read the positions of the cells of the array in positional notation with radix $b \geq 2$, i.e., an index h is represented by the sequence of $t = \lceil \log_b(n) \rceil$ digits in $\{0, \ldots, b-1\}$ such that $h = \sum_{i=0}^{t-1} h_i b^i$. The request of the array element at position h is performed in t communication rounds. First, the client asks the server to select all the cells having the least significant digit of the b-radix expansion of their positions equal to h_0 to compose a new array A_{h_0} concatenating the selected cells in increasing order of their original position, i.e., $A_{h_0}[j] = A[j \cdot b + h_0]$, $0 \leq j \leq \lceil \frac{n}{b} \rceil - 1$. In the next round, the client asks to select the cells in A_{h_0} having the least significant digit of the b-radix expansion of their positions equal to h_1, constructing an array A_{h_1} as $A_{h_1}[j] = A_{h_0}[j \cdot b + h_1] = A[j \cdot b^2 + h_1 \cdot b + h_0]$, $0 \leq j \leq \lceil \frac{n}{b^2} \rceil - 1$. The next rounds continue employing the subsequent digits of h with the same logic until, in the last round (i.e., the t-th one), a single cell (the h-th one) is identified by the server.

In the proper, fully private, PIR protocol [19], the client initially generates a public/private Paillier FLAHE keypair (pk, sk) with a public modulus $N \geq 2^{\omega}$, and shares pk with the server. The protocol is defined by three procedures: `PIR-trapdoor` and `PIR-retrieve`, executed at client side, and `PIR-search`, executed at server side.

PIR-Trapdoor procedure. The `PIR-trapdoor` procedure takes as input the public key pk, an integer $b \geq 2$, and the remote array index h referring to the item that must be retrieved. The output value is an "obfuscated" version of h, denoted as $\langle h \rangle$. The first step of the trapdoor computation considers the value h as the sequence of $t = \lceil \log_b(h) \rceil$ digits in b-radix positional representation. Each digit h_i with $0 \leq i \leq t-1$, is encoded as a bit-string $\texttt{hdigit}_i$, with length b, constructed as $\texttt{hdigit}_i[x]=1$ if $x=h_i$, 0 otherwise, $x \in \{0, \ldots, b-1\}$. Then, each bit $\texttt{hdigit}_i[x]$, $x \in \{0, \ldots, b-1\}$ of the string $\texttt{hdigit}_i$ is considered as a plaintext in $\mathbb{Z}_{N^l}$, $l=i+1$ and is encrypted into a ciphertext in $\mathbb{Z}^*_{N^{l+1}}$. Thus, the bit-wise encryption of the b-bit string $\texttt{hdigit}_i$ is given as the concatenation of b ciphertexts in $\mathbb{Z}^*_{N^{l+1}}$. The "obfuscated" version of h, $\langle h \rangle$, is returned as the concatenation of the bit-wise encryptions of each b-bit string in the sequence $\texttt{hdigit}_0, \texttt{hdigit}_1, \ldots, \texttt{hdigit}_{t-1}$, with total size $b \log_b^2(n) \log(N)$ bits. The computational cost of the `PIR-trapdoor` procedure amounts to $O(b \log^3(N) \log_b^4(n))$ bit operations, assuming the use of modular multiplication quadratic in the size of the operands.

PIR-search procedure. The `PIR-search` procedure, run at server side, takes as input the obfuscated value of h, $\langle h \rangle$ and the value of the radix b from the client, as well as the array A of items to be accessed, and returns a ciphertext that will be decrypted by the client as the content of $A[h]$. The search steps executed at server side follows the t-iterations over the array A reported in the draft description of the PIR protocol. In particular, in the first iteration, the server computes an encrypted array $\langle A_{h_0} \rangle$ with $\lceil \frac{n}{b} \rceil$ items, where each entry $\langle A_{h_0} \rangle[j]$, $0 \leq j \leq \lceil \frac{n}{b} \rceil - 1$ is a ciphertext in $\mathbb{Z}^*_{N^2}$ encrypting the item $A[j \cdot b + h_0]$ (i.e., $\mathtt{D}_{sk}(\langle A_{h_0} \rangle[j]) = A[j \cdot b + h_0]$). To this end, each item $\langle A_{h_0} \rangle[j]$ is computed as the *homomorphic dot product* between the sub-array $A[j \cdot b, \ldots, j \cdot b + b - 1]$, whose entries are plaintexts in $\mathbb{Z}_N$, and the bit-wise encryption of the b-bit string $\texttt{hdigit}_0$, whose b ciphertexts are in $\mathbb{Z}^*_{N^2}$. In the second iteration, the server constructs an array $\langle A_{h_1} \rangle$ with $\lceil \frac{n}{b^2} \rceil$ items, where the $\langle A_{h_1} \rangle[j]$ item $0 \leq j \leq \lceil \frac{n}{b^2} \rceil - 1$ is computed as the *homomorphic dot product* between the sub-array $\langle A_{h_0} \rangle[j \cdot b], \ldots, \langle A_{h_0} \rangle[j \cdot b + b - 1]$, whose entries are ciphertexts in $\mathbb{Z}^*_{N^2}$, and the bit-wise encryption of the b-bit string $\texttt{hdigit}_1$, whose b ciphertexts are in $\mathbb{Z}^*_{N^3}$. Specifically, this dot-product is computed by combining the homomorphic addition and `HybridMul` of the FLAHE Paillier scheme in the same fashion showed for the AHE Paillier scheme, i.e., $\langle A_{h_1} \rangle[j] = \Pi_{z=0}^{b-1} \texttt{hdigit}_1[z]^{\langle A_{h_0} \rangle[j \cdot b + z]} \bmod N^3$. The result of this dot-product is a ciphertext in $\mathbb{Z}^*_{N^3}$ which encrypts the item $\langle A_{h_0} \rangle[j \cdot b + h_1]$. As the latter element is a ciphertext itself, then $\langle A_{h_1} \rangle[j]$ is a *double-layered* ciphertext, that is the item $A[j \cdot b^2 + h_1 \cdot b + h_0]$ could be obtained by decrypting twice the ciphertext $\langle A_{h_1} \rangle$: i.e., $A[j \cdot b^2 + h_1 \cdot b + h_0] = \mathtt{D}_{sk}(\mathtt{D}^2_{sk}(\langle A_{h_1} \rangle[j]))$. After $t = \lceil \log_b(n) \rceil$ iterations, the server computes a single *t-layered* ciphertext $\langle A_{h_{t-1}} \rangle$ and sends it back to the client, who in turn must decrypt it t times to derive the target value $A[h]$. The computational cost of the `PIR-search` procedure amounts to $O(\frac{n}{b} \log^3(N))$ bit operations to compute a ciphertext with $\lceil \log_b(n) \rceil \log(N)$ bits.

PIR-Retrieve Procedure. This procedure, run at client side, employs the secret key sk to decrypt the ciphertext $A_{h_{t-1}}$ computed by the `PIR-Search` procedure, obtaining the requested element $A[h]$.

Since $A_{h_{t-1}}$ is a *t-layered* ciphertext, then the client must remove all these t encryption layers by decrypting t times with decreasing length: i.e., $A[h] = \mathtt{D}_{sk}(\mathtt{D}^2_{sk}(\ldots \mathtt{D}^t_{sk}(\langle A_{h_{t-1}} \rangle)))$. The computational cost of the `PIR-Retrieve` amounts to $O(\log_b^5(n) \log^2(N))$ bit operations to derive the target value $A[h]$. Lastly, the communication cost of the described single-round PIR-protocol amounts to $O(\log(N) b \log_b^2(n))$ bits sent from client to server, and to $O(\log(N) \log_b(n))$ bits sent from server to client.

4 PROPOSED PPSS PROTOCOL

Definition 4.1 (Substring Search Functionality). Consider a collection of $z \geq 1$ documents $\mathbf{D} = \{D_1, \ldots, D_z\}$, each intended as a string of $\texttt{len}(D_i)$, $1 \leq i \leq z$, symbols of the alphabet Σ, stored on the server, and a query string $q \in \Sigma^m$, $m \geq 1$, provided by the client.

The substring search functionality computes the number of occurrences of q in each document of $\mathbf{D}$, that is the set $O_{\mathbf{D},q} = \bigcup_{i=1}^{z} O_{D_i,q}$, where

$$O_{D_i,q} = \{\, 1 \leq j \leq \texttt{len}(D_i) - m + 1 \} \mid q = D_i[j], \ldots, D_i[j+m-1] \,\}$$

A privacy-preserving substring search (PPSS) protocol allows the server to provide the functionality specified in Definition 4.1 without learning the content of the document collection, $\mathbf{D}$, the value of the substring q and the positions in $O_{\mathbf{D},q}$, as well as guaranteeing search and access pattern privacy. To this end, the protocol needs to hide all these data by employing *privacy-preserving representations*. We will denote the privacy-preserving representation of a datum by enclosing it in square brackets (e.g., $[[\mathbf{D}]]$).

Definition 4.2 (PPSS Protocol). A PPSS protocol $\mathcal{P}$ for a set of $z \geq 1$ documents $\mathbf{D} = \{D_1, \ldots, D_z\}$ over an alphabet Σ, is a pair of polynomial-time algorithms $\mathcal{P} = (\texttt{Setup}, \texttt{Query})$.
The setup procedure: $([[\mathbf{D}]], aux_s) \leftarrow \texttt{Setup}(\mathbf{D}, 1^{\lambda})$, is a probabilistic algorithm, run by the client, taking as input the

security parameter λ and the document collection $\mathbf{D}$, and returning its privacy-preserving representation $[[\mathbf{D}]]$ together with an auxiliary pieces of information, aux_s which is kept secret by the client.

The query procedure: $R \leftarrow \texttt{Query}(q, aux_s, [[\mathbf{D}]])$, is a deterministic algorithm which is run interactively by the client and the server to compute the number of occurrences of the string $q \in \Sigma^m$ in each document of $\mathbf{D}$. The client obtains $R = O_{\mathbf{D},q} = \bigcup_{i=1}^{z} O_{D_i,q}$, where $O_{\mathbf{D},q}$ is as per Definition 4.1, while the server outputs nothing.

The `Query` procedure iterates $w \geq 1$ rounds, where each round corresponds to the execution of three algorithms:

- `Trapdoor`: $[[q]]_j \leftarrow \texttt{Trapdoor}(j, q, aux_s, res_0, \ldots, res_{i-1})$, is a probabilistic algorithm, run at client side, which employs aux_s and the results of previous rounds to build the privacy-preserving representation (a.k.a. trapdoor) $[[q]]_j$ of the queried substring q for the j-th round.
- `Search`: $[[res_j]] \leftarrow \texttt{Search}([[q]]_j, [[\mathbf{D}]])$, is a deterministic algorithm, run at server side, which employs $[[q]]_j$ and $[[\mathbf{D}]]$ to compute a privacy-preserving representation of the result for the j-th round, i.e., $[[res_j]]$.
- `Retrieve`: $res_j \leftarrow \texttt{Retrieve}([[res_j]], aux_s)$, is a deterministic algorithm, run at client side, which takes as inputs $[[res_j]]$ and aux_s and computes the result res_j.

Relying on the substring search algorithm based on the BWT transformation and reported in Algorithm 1 and the Lipmaa PIR protocol based on the FLAHE Paillier scheme, we now provide the operational description of the proposed PPSS protocol, reported in Algorithm 2 and Algorithm 3.

The document collection $\mathbf{D}$ employed for the searching operation is encrypted with a symmetric-key, and outsourced to the remote server. Along with the encrypted version of $\mathbf{D}$, the client computes the indexing structure $[[\mathbf{D}]]$ by employing the `Setup` procedure.

This procedure (see Alg. 2) takes as input the z documents in $\mathbf{D}$ to compute a single string s obtained concatenating the documents, interleaved with \$ (lines 2–3). The additional input λ is an integer number representing the computational security level employed to instantiate the underlying cryptographic primitives. Subsequently, the procedure computes the $(|\Sigma|+1)\times(n+1)$ matrix representation of $L=\texttt{BWT}(s)$, denoted as M in Algorithm 1, the corresponding $1 \times (n+1)$ suffix array, SA, and the `Rank` dictionary with size $|\Sigma|+1$, containing pairs (c, l), where $l=\texttt{Rank}(c)$, $0 \leq l \leq n+1$ is the number of characters in s alphabetically smaller than c. As the rows of M are indexed by characters in $\Sigma \cup \{\$\}$, a bijective function $\texttt{Order} : \Sigma \cup \{\$\} \mapsto \{0, 1, \ldots, |\Sigma|\}$, is employed to build a dictionary including pairs (c, o), where $c \in \Sigma \cup \{\$\}$ and $o=\texttt{Order}(c)$ is the unique numerical index corresponding to the character indexing a row of M. At lines 4–6, the integer matrix M is converted into a $(|\Sigma|+1)\cdot(n+1)$ array of integers, C, built as the concatenation of the rows of M in ascending order of the numerical index obtained via the `Order` function. We note that `Rank(c)` is summed to $M[c][j]$ at line 6 of Algorithm 2 to save the additions that should be executed later as per lines 5–6 of Algorithm 1.

As the data structures C and SA are sufficient to reconstruct s, and thus the document collection $\mathbf{D}$, they are cell-wise encrypted, obtaining arrays $\langle C \rangle$ and $\langle SA \rangle$, before being outsourced. To this end, any secure cipher $\mathcal{E}$ can be employed; we choose a symmetric block cipher for efficiency reasons. The algorithms referring to the mentioned cipher are denoted as ($\mathcal{E}$.`KeyGen`, $\mathcal{E}$.`Enc`, $\mathcal{E}$.`Dec`), where the `KeyGen` procedure yields a pair of public and private keys, i.e.: $pk_{\mathcal{E}}$, $sk_{\mathcal{E}}$ (line 7), where $pk_{\mathcal{E}} = sk_{\mathcal{E}}$ if $\mathcal{E}$ is a symmetric-key cipher.

Since an adversary with partial knowledge of the content of the document collection will infer the content of some of the cells in SA or C, the cell-wise encrypted arrays $\langle SA \rangle$ and $\langle C \rangle$ are randomly shuffled at lines 8–11 by employing two keyed Pseudo Random Permutations (PRPs) [2], denoted as π_{SA}, π_M, respectively, which are defined as follows:

$$\pi_{SA} : \{0,1\}^{\lambda} \times \{1, \ldots, n+1\} \mapsto \{1, \ldots, n+1\}$$

with $\pi_{SA}(k, i)=j$, $1 \leq i, j \leq n+1$; while,

$$\pi_C : \{0,1\}^{\lambda} \times \{1, \ldots, (|\Sigma|+1)\cdot(n+1)\} \mapsto \{1, \ldots, (|\Sigma|+1)\cdot(n+1)\}$$

with $\pi_C(k, i)=j$, $1 \leq i, j \leq (|\Sigma|+1)\cdot(n+1)$. The same key K generated at line 7 is employed for both the PRPs. At line 12, the secret information kept by the client aux_s is computed as the dictionary `Order`, the secret key of cipher $\mathcal{E}$ and key K employed in the PRPs. Finally, the `Setup` procedure in Algorithm 2 returns the secret data to be kept by the client, $aux_s=(\texttt{Order}, sk_{\mathcal{E}}, K)$, and the privacy-preserving representation $[[\mathbf{D}]]$ of the indexing structure of the document collection to be outsourced, as the pair of encrypted data structures $(\langle C \rangle, \langle SA \rangle)$.

The `Query` procedure takes as input the m-character string to be searched q, the secret parameters of the client $aux_s=(\texttt{Order}, sk_{\mathcal{E}}, K)$, and the privacy-preserving representation $[[\mathbf{D}]] = (\langle C \rangle, \langle SA \rangle)$.

The operations performed during the execution of the `Query` procedure are grouped in two phases. The first phase, labeled as `Qnum` (lines 2–11), corresponds to lines 1–6 in Algorithm 1, and allows to evaluate as $\beta - \alpha$ the total number of occurrences of q in the remotely stored documents. In particular, all memory look-ups performed on the matrix representation M of the BWT of the document collection in Algorithm 1 are realized accessing the cells of the array $\langle C \rangle$. As the cells of $\langle C \rangle$ are shuffled w.r.t. the cells over C, the client needs to compute the position of an entry $C[\alpha]$ ($C[\beta]$ resp.) in $\langle C \rangle$, by employing the keyed PRP as shown in line 5 (line 9 resp.). Realizing each access via the primitives of

Algorithm 2: `Setup` Procedure of our PPSS Protocol

Function `Setup`(**D**,λ):
Input: Document Collection **D** = $\{D_1, \ldots, D_z\}$, security parameter λ
Output: [[**D**]], privacy-preserving representations of the indexing structure of **D**; aux_s, secret auxiliary information employed by the client to perform search requests

1 **begin**
2 $s \leftarrow \mathtt{concat}(D_1, \$, D_2, \$, \ldots, D_z, \$)$
3 $n \leftarrow \sum_{i=1}^{z} \mathtt{len}(D_i) + 1$
/* Compute the suffix array SA, the matrix M and the Rank dictionary for string s (see Section 3.1) */
/* Compute the dictionary $\mathtt{Order} : \Sigma \cup \{\$\} \mapsto \{0, 1, \ldots, |\Sigma|\}$, containing pairs (c, o) where $c \in \Sigma \cup \{\$\}$, and o=Order(c) is a unique numerical index. */
4 **foreach** $c \in \Sigma \cup \{\$\}$ **do**
5 **for** $j \leftarrow 1$ **to** $n + 1$ **do**
6 $C[\mathtt{Order}(c) \cdot (n+1) + j] \leftarrow \mathtt{Rank}(c) + M[c][j]$
7 $K \xleftarrow{\mathcal{R}} \{0,1\}^{\lambda}, (pk_{\mathcal{E}}, sk_{\mathcal{E}}) \leftarrow \mathcal{E}.\mathtt{KeyGen}(\lambda)$
8 **for** $i \leftarrow 1$ **to** $n + 1$ **do**
9 $\langle SA \rangle[\pi_{SA}(K, i)] \leftarrow \mathcal{E}.\mathtt{Enc}(pk_{\mathcal{E}}, SA[i])$
10 **for** $i \leftarrow 1$ **to** $(n + 1) \cdot (|\Sigma| + 1)$ **do**
11 $\langle C \rangle[\pi_C(K, i)] \leftarrow \mathcal{E}.\mathtt{Enc}(pk_{\mathcal{E}}, C[i])$
12 $aux_s \leftarrow (\mathtt{Order}, sk_{\mathcal{E}}, K)$
13 [[**D**]] $\leftarrow (\langle C \rangle, \langle SA \rangle)$
14 **return** $(aux_s, [[\mathbf{D}]])$

Algorithm 3: `Query` Procedure of our PPSS Protocol

Function `Query`(q, aux_s, [[**D**]]):
Input: q, m-character string to be search; aux_s, secret auxiliary information employed by the client to perform search requests containing $(\mathtt{Order}, sk_{\mathcal{E}}, K)$; [[**D**]], remotely accessed privacy-preserving representations of the indexing structure of **D**, containing $(\langle C \rangle, \langle SA \rangle)$.
Output: R_q, set of positions of occurrences of q in **D**
Data: (pk, sk), public and private Paillier FLAHE keypair; b, radix employed to represent in positional notation an integer index in the Lipmaa PIR protocol

1 **begin**
2 $\alpha \leftarrow 0, \beta \leftarrow n+1$ // **start of the 1st phase:** `Qnum`
3 **for** $i \leftarrow m$ **downto** 1 **do**
4 $\alpha \leftarrow \alpha + \mathtt{Order}[q[i]] \cdot (n + 1)$
5 $\langle h \rangle \leftarrow \mathtt{PIR\text{-}Trapdoor}(pk, b, \pi_C(K, \alpha))$
6 $\mathtt{ctx} \leftarrow \mathtt{PIR\text{-}Search}(\langle h \rangle, b, \langle C \rangle)$
// ciphertext of $\langle C \rangle[\pi_C(K, \alpha)]$
7 $\alpha \leftarrow \mathcal{E}.\mathtt{Dec}(sk_{\mathcal{E}}, \mathtt{PIR\text{-}Retrieve}(sk, \mathtt{ctx}))$
8 $\beta \leftarrow \beta + \mathtt{Order}[q[i]] \cdot (n + 1)$
9 $\langle h \rangle \leftarrow \mathtt{PIR\text{-}Trapdoor}(pk, b, \pi_C(K, \beta))$
10 $\mathtt{ctx} \leftarrow \mathtt{PIR\text{-}Search}(\langle h \rangle, b, \langle C \rangle)$
// ciphertext of $\langle C \rangle[\pi_C(K, \beta)]$
11 $\beta \leftarrow \mathcal{E}.\mathtt{Dec}(sk_{\mathcal{E}}, \mathtt{PIR\text{-}Retrieve}(sk, \mathtt{ctx}))$
12 $R_q \leftarrow \emptyset$ // **start of the 2nd phase:** `Qocc`
13 **for** $i \leftarrow \alpha + 1$ **to** β **do**
14 $\langle h \rangle \leftarrow \mathtt{PIR\text{-}Trapdoor}(pk, b, \pi_{SA}(K, i))$
15 $\mathtt{ctx} \leftarrow \mathtt{PIR\text{-}Search}(\langle h \rangle, \langle SA \rangle)$
// ciphertext of $\langle SA \rangle[\pi_{SA}(K, i)]$
16 $R_q \leftarrow R_q \cup \mathcal{E}.\mathtt{Dec}(sk_{\mathcal{E}}, \mathtt{PIR\text{-}Retrieve}(sk, \mathtt{ctx}))$
17 **return** R_q

any PIR protocol allows to hide the position of the array cell requested by the client thus, providing search pattern privacy of the retrieved content. Indeed, without the PIR protocol the adversary, i.e., the server, would be able to infer the similarity between the strings searched in two separate queries due to deterministic access to the same positions of the array $\langle C \rangle$. In our proposal, the Lipmaa PIR protocol described in in Section 3.2 is adopted due to its efficiency in terms of communication complexity. Finally, as each cell $\langle C \rangle[h]$ stores an encrypted content, the client needs to further decrypt the material returned by the PIR-retrieve procedure, as shown in line 7 (line 11 resp.).

The second phase, labeled as `Qocc` (lines 12–16), corresponds to lines 7–9 in Algorithm 1, and allows to compute the set of positions, in the remotely stored documents, where the leading characters of the occurrences of q are found. Similarly to the previous phase, each memory look-up to the suffix array data structure in Algorithm 1 is realized by accessing privately the cells of the array $\langle SA \rangle$.

Informally, the security of our protocol is based on the security of the PIR protocol employed and on the semantic security of the encryption scheme used to encrypted the array $\langle C \rangle$ and the suffix array $\langle SA \rangle$, as the server observes only PIR queries on arrays encrypted with a semantically secure encryption scheme. The only information leaked to the server is the size of the array $\langle C \rangle$ and of the suffix array $\langle SA \rangle$, which are both proportional to the size of the document collection, while the length m of the substring q and the number of occurrences $|R_q|$ are leaked by the number of iterations required by the execution of the phases in Algorithm 3 labeled as `Qnum` and `Qocc`, respectively.

Concerning the computational and communication complexities of the `Setup` and `Query` procedures, we note that the former costs $O(n)$ bit operations, while storing [[**D**]] on the server requires $O(n)$ storage space. The cost of the

latter procedure is split between the client and the server obtaining, respectively, $O((m+|R_q|)\cdot b\log^3(N)\log_b^4(n))$ cost, where N is the modulus employed in the FLAHE Paillier keypair, and $O((m+|R_q|)\cdot\frac{n}{b}\log^3(N))$ cost. The amount of data exchanged between the client and the server amounts to $O((m+|R_q|)\cdot\log(N)b\log_b^2(n))$.

Multi-user Extension. Differently from many of the current PPSS protocols, our approach can be promptly and efficiently adapted to a multi-user scenario where a data-owner outsources the indexing data structure to a service provider, and multiple users equipped with their own Pailler FLAHE key-pair access the data structure running the PIR primitives simultaneously.

In such a setting, each user is guaranteed to perform its own substring search without leaking any information to both other users and the service provider itself. Indeed, the search and access pattern privacy of the queries of a user are guaranteed even in case of collusion between other users and the service provider.

From an operational point of view the data owner runs the `Setup` procedure shown in Algorithm 2, computing the pair of arrays $[[\mathbf{D}]]=(\langle C\rangle,\langle SA\rangle)$ to be outsourced and shares the secret auxiliary information $aux_s \leftarrow (\texttt{Order}, sk_{\mathcal{E}}, K)$ with the authorized users. Each authorized user in turn can independently run a modified version of the `Query` procedure shown in Algorithm 3 to find occurrences of a substring of her/his choice. The modifications to the `Query` procedure consists in replacing the use of the original Lipmaa `PIR-Search` primitive with the one reported in Algorithm 4, which aims to reduce the memory consumption of the Lipmaa `PIR-Search` procedure when multiple queries are simultaneously performed. Indeed, each run of the `PIR-Search` procedure in Lipmaa's protocol (Section 3.3) runs $t=\lceil\log_b(n)\rceil$ iterations, with the i-th iteration computing an array $A_{h_{i-1}}$ with $\lceil\frac{n}{b^i}\rceil$ elements. In particular, the first iteration computes an array A_{h_0} with $\lceil\frac{n}{b}\rceil$ entries, in turn requiring $O(n)$ memory to be allocated. Therefore, if u queries are performed simultaneously, the memory consumption of Lipmaa's protocol is $O(n+u\cdot n)$, providing poor scalability in case of multiple queries. To address this scalability issue, we propose to schedule differently the operations performed in the PIR protocol. Specifically, the `PIR-Search` procedure serializes the computation of the entire arrays $A_{h_0},\ldots,A_{h_{t-1}}$. Nonetheless, it is possible to compute the element $A_{h_1}[0]$ as soon as the b elements $A_{h_0}[0],\ldots,A_{h_0}[b-1]$ are computed, and, similarly, compute $A_{h_1}[1]$ as soon as the b elements $A_{h_0}[b],\ldots,A_{h_0}[2b-1]$ are computed. Considering a generic $A_{h_i}[j]$, $0\le i\le t-1$, $0\le j\le\lceil\frac{n}{b^{i+1}}\rceil$ element, we can compute it as soon as the b elements $A_{h_{i-1}}[b\cdot j],\ldots,A_{h_{i-1}}[b\cdot j+b-1]$ are available. This schedule of the operations is achieved by the recursive computation in Algorithm 4.

Algorithm 4: Optimized `PIR-Search` algorithm

Function `PIR-Search`($\langle h\rangle$, b, A):
  Input: $\langle h\rangle$, obfuscated value of the position h,
    represented as the concatenation of the bit-wise
    encryptions of each b-bit string in the sequence
    $\texttt{hdigit}_0,\ldots,\texttt{hdigit}_{t-1}$, with $t=\lceil\log_b(n)\rceil$
    (see Section 3.3);
    $b\ge 2$, radix chosen by the client to construct
    $\langle h\rangle$;
    A, remote array with n entries.
  Output: content of the cell $A[h]$
  return `RecursiveRet` ($\langle h\rangle$, A, $\lceil t\rceil$, 1, n, b)

Function `RecursiveRet`($\langle h\rangle$, A, l, `begin`, `end`, b):
  if `end` − `begin` = 0 **then**
    return A[`begin`]
  `size` ← $\left\lfloor\frac{\texttt{end}-\texttt{begin}}{b}\right\rfloor$, `acc` ← 1
  for i ← 1 ***to*** b **do**
    `el` ←
     `RecursiveRet`($\langle h\rangle, A, l-1,$ `begin`, `begin` + `size`)
    `begin` ← `begin` + `size` + 1
    `acc` ← (`acc` $\cdot\ \langle h\rangle[(l\cdot b+i]^{\texttt{el}})$ mod N^{l+1}
  return `acc`

The computational complexity of this algorithm is clearly equivalent to the naive iterative implementation, as the same operations are performed. Nevertheless, it exhibits a sub-linear memory consumption per query. Indeed, the maximum depth of recursion is $O(\log(n))$, which means that only the memory for the $O(\log(n))$ recursive calls is required. Each recursive call needs to store $O(l\log(N))$ bits due to the Paillier ciphertexts in $\mathbb{Z}^*_{N^{l+1}}$, hence the overall storage cost is: $\sum_{l=1}^{\lceil\log(n)\rceil}O(l\log(N))=O(\log^2(n)\log(N))$. In conclusion, when u queries are simultaneously performed, the server stores only $O(n+u\cdot\log^2(n))$ memory, with significant savings w.r.t. a naive approach.

5 SECURITY ANALYSIS

In the previous sections we observed how our PPSS protocol ensures the confidentiality of the remotely stored string, of the searched substring, and of the results returned by each search query. Furthermore, it provide indistinguishability of the *search-pattern* followed by multiple queries as well as the *access-pattern* privacy of locating the occurrences of a given substring.

In the following, adopting the framework introduced by Curtmola in [8], we provide a formal definition of the information leakage coming from a PPSS and we formally specify the adversarial model as well as the security guarantees provided by our PPSS protocol.

Experiment $transcript \leftarrow \mathbf{Real}_{\mathcal{P},\mathcal{A}}(\lambda)$:
$(\mathbf{D}, \mathtt{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_{\mathbf{D}}(1^{\lambda})$, $([[\mathbf{D}]], aux_s) \leftarrow \mathcal{P}.\mathtt{Setup}(\mathbf{D}, 1^{\lambda})$
$\forall i \in \{1, \ldots, d\}$: $\mathtt{List\_q_i} \leftarrow \emptyset, \mathtt{List\_R_i} \leftarrow \emptyset$
  $(q_i, \mathtt{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_i\left([[\mathbf{D}]], \{\mathtt{List\_q_l}\}_{l=1}^{i-1}, \{\mathtt{List\_R_l}\}_{l=1}^{i-1}, \mathtt{st}_{\mathcal{A}}\right)$
  $\forall j \in \{1, \ldots, w\}$:
    $[[q_i]]_j \leftarrow \mathcal{P}.\mathtt{Trapdoor}(j, q_i, aux_s, res_1, \ldots, res_{j-1})$
    $([[res_j]], \mathtt{st}_{\mathcal{A}}) \leftarrow \mathcal{A}.\mathtt{Search}(\mathtt{st}_{\mathcal{A}}, [[q_i]]_j, [[\mathbf{D}]])$
    $res_j \leftarrow \mathcal{P}.\mathtt{Retrieve}([[res_j]], aux_s)$
  $\mathtt{List\_q_i} \leftarrow ([[q_i]]_1, \ldots, [[q_i]]_w)$
  $\mathtt{List\_R_i} \leftarrow ([[res_1]], \ldots, [[res_w]])$
$transcript \leftarrow \left\{[[\mathbf{D}]], \mathtt{st}_{\mathcal{A}}, \{\mathtt{List\_q_i}\}_{i=1}^{d}, \{\mathtt{List\_R_i}\}_{i=1}^{d}\right\}$

Experiment $transcript \leftarrow \mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$:
$(\mathbf{D}, \mathtt{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_{\mathbf{D}}(1^{\lambda})$, $([[\mathbf{D}]], \mathtt{st}_{\mathcal{S}}) \leftarrow \mathcal{S}_{\mathbf{D}}(\mathcal{L}_{\mathbf{D}}, 1^{\lambda})$
$\forall i \in \{1, \ldots, d\}$: $\mathtt{List\_q_i} \leftarrow \emptyset, \mathtt{List\_R_i} \leftarrow \emptyset$
  $(q_i, \mathtt{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_i\left([[\mathbf{D}]], \{\mathtt{List\_q_l}\}_{l=1}^{i-1}, \{\mathtt{List\_R_l}\}_{l=1}^{i-1}, \mathtt{st}_{\mathcal{A}}\right)$
  $\forall j \in \{1, \ldots, w\}$:
    $([[q_i]]_j, \mathtt{st}_{\mathcal{S}}) \leftarrow \mathcal{S}_{q_i}(j, \mathtt{st}_{\mathcal{S}}, \mathcal{L}_{\mathbf{D}}, \mathcal{L}_{q_1}, \ldots, \mathcal{L}_{q_{i-1}}, \mathcal{L}_{q_i})$
    $([[res_j]], \mathtt{st}_{\mathcal{A}}) \leftarrow \mathcal{A}.\mathtt{Search}(\mathtt{st}_{\mathcal{A}}, [[q_i]]_j, [[\mathbf{D}]])$
  $\mathtt{List\_q_i} \leftarrow ([[q_i]]_1, \ldots, [[q_i]]_w)$
  $\mathtt{List\_R_i} \leftarrow ([[res_1]], \ldots, [[res_w]])$
$transcript \leftarrow \left\{[[\mathbf{D}]], \mathtt{st}_{\mathcal{A}}, \{\mathtt{List\_q_i}\}_{i=1}^{d}, \{\mathtt{List\_R_i}\}_{i=1}^{d}\right\}$

Figure 2: Security game experiments

Definition 5.1 (Leakage of PPSS Protocol). Given a document collection $\mathbf{D}$, a string q, and a PPSS protocol $\mathcal{P} = (\mathtt{Setup}, \mathtt{Query})$ its leakage $\mathcal{L} = (\mathcal{L}_{\mathbf{D}}, \mathcal{L}_q)$ is defined as follows. $\mathcal{L}_{\mathbf{D}}$ denotes the information learnt by the adversary in the `Setup` phase, i.e., the information inferred by the adversary from the observation of the privacy-preserving representation $[[\mathbf{D}]]$. $\mathcal{L}_q$ denotes the information learnt by the adversary in the w iterations (rounds) executed during the `Query` phase of the protocol, i.e., information inferred from the result of the `Trapdoor` procedure and the execution of the `Search` procedure.

The security game stated in Definition 5.2 allows to prove that a semi-honest adversary does not learn anything but the leakage $\mathcal{L}$. To this end, this definition requires the existence of a simulator $\mathcal{S}$, taking as inputs only $\mathcal{L}_{\mathbf{D}}$ and $\mathcal{L}_q$, which is able to generate a transcript of the PPSS protocol for the adversary that is computationally indistinguishable from the one generated when a legitimate client interacts with the server during a real execution of the protocol.

Definition 5.2 (Security Game). Given a PPSS protocol $\mathcal{P}$ with security parameter λ, $d \geq 1$ queries and the leakage of $\mathcal{P}$ for all the queries $\mathcal{L} = (\mathcal{L}_{\mathbf{D}}, \mathcal{L}_{q_1}, \ldots, \mathcal{L}_{q_d})$, an adversary $\mathcal{A}$ consisting of $d+1$ probabilistic polynomial time algorithms $\mathcal{A} = (\mathcal{A}_{\mathbf{D}}, \mathcal{A}_1, \ldots \mathcal{A}_d)$, and a simulator $\mathcal{S}$, which is also a tuple of $d+1$ probabilistic polynomial time algorithms $\mathcal{S} = (\mathcal{S}_{\mathbf{D}}, \mathcal{S}_{q_1}, \ldots \mathcal{S}_{q_d})$, the two probabilistic experiments $\mathbf{Real}_{\mathcal{P},\mathcal{A}}(\lambda)$ and $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$ shown in Fig. 2 are considered. Denote as $\mathcal{D}(o)$ a probabilistic polynomial time algorithm taking as input a transcript of an experiment o and returning a boolean value indicating if the transcript belongs to the real or ideal experiment. The protocol $\mathcal{P}$, with leakage $\mathcal{L}$, is secure against every semi-honest probabilistic polynomial time adversary $\mathcal{A} = (\mathcal{A}_D, \ldots \mathcal{A}_d)$, if there exists a simulator $\mathcal{S} = (\mathcal{S}_D, \mathcal{S}_{q_1}, \ldots \mathcal{S}_{q_d})$ such that for every $\mathcal{D}$:

$$\mathtt{Pr}\left(\mathcal{D}(o)=1 | o \leftarrow \mathbf{Real}_{\mathcal{P},\mathcal{A}}(\lambda)\right) - \mathtt{Pr}\left(\mathcal{D}(o)=1 | o \leftarrow \mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)\right) \leq \epsilon(\lambda),$$

where $\epsilon(\cdot)$ is a negligible function.

In the experiments shown in Fig. 2, $\mathbf{D}$ is chosen by the adversarial algorithm $\mathcal{A}_D$ and the query q_i is adaptively chosen by the i-th adversarial algorithm $\mathcal{A}_i$, depending on the transcripts of the protocol in the previous queries. All the adversarial algorithms share a state, denoted as $\mathtt{st}_{\mathcal{A}}$, which is used to store possible information learnt by the adversary throughout the experiment.

The $\mathbf{Real}_{\mathcal{P},\mathcal{A}}$ experiment represents an actual execution of the protocol, where the client receives the document collection $\mathbf{D}$ and the d queries and it behaves as specified in the protocol; conversely, in the $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}$ experiment, the client is simulated by $\mathcal{S}$, which however employs only the leakage information $\mathcal{L} = (\mathcal{L}_D, \mathcal{L}_{q_1}, \ldots, \mathcal{L}_{q_d})$. In particular, the simulator $\mathcal{S}_D$ constructs a privacy-preserving representation $[[\mathbf{D}]]$ by exploiting only the knowledge of $\mathcal{L}_D$, while each simulator $\mathcal{S}_{q_i}$ constructs the trapdoor for each round of the i-th query by exploiting only the knowledge of the leakage $\mathcal{L}_D, \mathcal{L}_{q_j}, j = 1, \ldots, i$.

Theorem 5.3. *Given a document collection $\mathbf{D}$ with $z \geq 1$ documents $\{D_1, \ldots, D_z\}$ and $d \geq 1$ substrings $q_1, \ldots, q_d$, our PPSS protocol is secure against a semi-honest adversary, as per Definition 5.2, with a leakage $\mathcal{L} = (\mathcal{L}_D, \mathcal{L}_{q_1}, \ldots, \mathcal{L}_{q_d})$, where $\mathcal{L}_D = (\sum_{i=1}^{z}(\mathtt{len}(D_i)+1), \omega)$, with ω denoting the size of ciphertexts computed by the semantically secure encryption scheme $\mathcal{E}$ employed to construct $[[\mathbf{D}]]$, and $\mathcal{L}_{q_i} = (\mathtt{len}(q_i), b_i, |O_{\mathbf{D},q_i}|), 1 \leq i \leq d$, where $O_{\mathbf{D},q_i}$ is defined as per Definition 4.1 and b_i is the radix chosen to execute the Lipmaa PIR protocol.*

Proof. See Appendix A. □

We remark that Theorem 5.3 guarantees search and access pattern privacy, as they are not enclosed in the leakage $\mathcal{L}$.

6 EXPERIMENTAL EVALUATION

We validated our PPSS protocol implementing a client-server architecture and running it on a dual Intel Xeon CPU E5-2620 clocked at 3 GHz, endowed with 128 GiB DDR4-2133, and

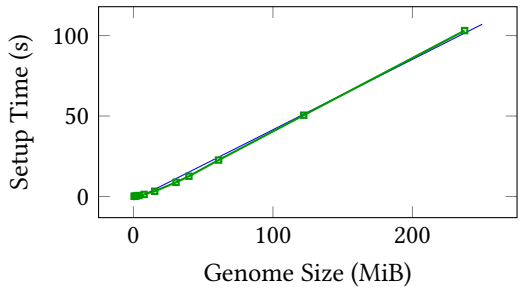

Figure 3: Execution time of the `Setup` procedure for genomes of increasing size. The blue line shows the fit between the experimental data and the linear model given by $SetupTime = 0.4369 * GenomeSize - 2.2$

64-bit Gentoo Linux17.0 OS. Our implementation provides a cryptographic security level of at least $\lambda = 80$ bits, relying on the multi-precision integer arithmetic GMP library [14] and a proper parametrization of the generalized Pailler algorithms provided by the LIBHCS library [29], to implement the PIR-related cryptographic operations. The AES-128 CounTeR (CTR) mode primitives of OPENSSL ver. 1.0.2r [17] are used for the cell-wise encryption/decryption of $[[\mathbf{D}]]=(\langle C\rangle, \langle SA\rangle)$. Our implementation, together with detailed instructions on how to reproduce the experimental campaign described in the following, as well as the data files employed for assessing functionalities and performance of the provided implementation, are publicly available online [20].

We chose as our case study a genomic dataset in the widely employed FASTA format [7] where an alphabet of five characters is employed to represent a DNA sequence, i.e.: $\Sigma = \{C, G, A, T, N\}$. Specifically, we considered a document containing approximately $40 \cdot 10^6$ nucleotides (characters) belonging to the 21-th human chromosome selected from the ENSEMBL publicly available data [12].

In the experiments, we considered documents with variable sizes replicating and truncating the mentioned dataset appropriately. We considered substring searches with a substring q having $m = 6$ characters, as it is the size of many *restriction enzyme sites* (transcribed as m-character strings), that are commonly employed in DNA-based paternity tests. Indeed, the test employs the distances between the occurrences of one of the mentioned substrings in the DNA fragments of two hosts to identify if the hosts are related [1].

In the actual implementation employed for the experimental campaign, we introduced some optimizations which allow to reduce the number of entries in the arrays $\langle C\rangle$ and $\langle SA\rangle$. First of all, we recall that $\langle C\rangle$ is the cell-wise encryption of the array C, which is obtained as described in lines 4 – 6 of Algorithm 2 from the matrix representation, M, of the BWT, L, of the document. Specifically, as any entry $M[c][i]$, with $c \in \Sigma \cup \{\$\}, i \in \{1, \ldots, n+1\}$ stores the number of occurrences of character c in the subarray $(L[1], \ldots, L[i])$, the array C has $(|\Sigma|+1) \cdot (n+1)$ entries storing $O(\log n)$ bits. To reduce the memory footprint of this array, we derive an *hybrid* representation between M and the BWT L: given a parameter R, referred to as *sample period*, we construct an array C_R with $\lceil \frac{n+1}{R} \rceil$ entries, where $C_R[j]$ is a tuple with $R+1$ elements, the first one being $M[:][j \cdot R]$, that is the $j \cdot R$-th column of M, and the other R ones are the characters of the BWT L at positions $\{j \cdot R, \ldots, j \cdot R + R - 1\}$ (i.e., $C_R[j] =$ `concatArrayWithCharacters`$(M[:][j \cdot R], L[j \cdot R], L[j \cdot R + 1], \ldots, L[j \cdot R + (R-1)])$). In this way, the array C_R has $\lceil \frac{n+1}{R} \rceil$ entries requiring (only) $O(|\Sigma| \cdot \log(n) + R \cdot \log(|\Sigma|)$ bits. The substring search procedure outlined in Algorithm 1 was modified accordingly to make use of C_R in place of M. Specifically, each access to $M[c][i], c \in \Sigma \cup \{\$\}, i \in \{1, \ldots, n+1\}$, is replaced by retrieving $M[c][\lfloor \frac{i}{R} \rfloor \cdot R]$ from the $\lfloor \frac{i}{R} \rfloor$-th entry of C_R and adding it to the number of occurrences of c in the first i mod R characters of the BWT L found in the $\lfloor \frac{i}{R} \rfloor$-th entry of C_R. We chose a sample period R which allows to encrypt each entry of C_R to an AES-128 CTR ciphertext within approximately $\log(N)$ bits, where N is the modulus employed in the LFAHE Paillier scheme. Furthermore, to reduce the number of entries of the array $\langle SA\rangle$, we encrypted in a single AES-128 CTR ciphertext of approximately $\log(N)$ bits as many entries as possible from the array SA. In this way, we reduced the original number of entries of the encrypted arrays $\langle C\rangle$ and $\langle SA\rangle$ by significant constant factors (resp. 1200 and 28), obtaining a comparable speed-up in the `Search` procedure.

In the first test, we focused on the `Setup` procedure of Algorithm 2, which builds the privacy-preserving representation $[[\mathbf{D}]]$ of the dataset. The execution time for this procedure for genomes of increasing size is reported in Figure 3. In this test we considered also the genomic data corresponding to the 1-st human chromosome, which is much bigger than the 21-th one employed in all other tests. The experimental results confirm the expected linear trend and they show practical performance for the `Setup` procedure: indeed, building the privacy-preserving representation of the 1-st human chromosome, which is as big as 238 MB, requires only 103 seconds.

In the subsequent tests, we profiled the performance of the `Query` procedure. We evaluated separately the two phases of the `Query` procedure, labeled as `Qnum` and `Qocc` in Algorithm 3, that compute the number of occurrences and the set of positions of the leading character of the occurrence of the substring, respectively. The performance figures related to the second phase refers to the retrieval of a single occurrence, as the costs of retrieving all of them is proportional to their number. We remark that the communication cost reported in

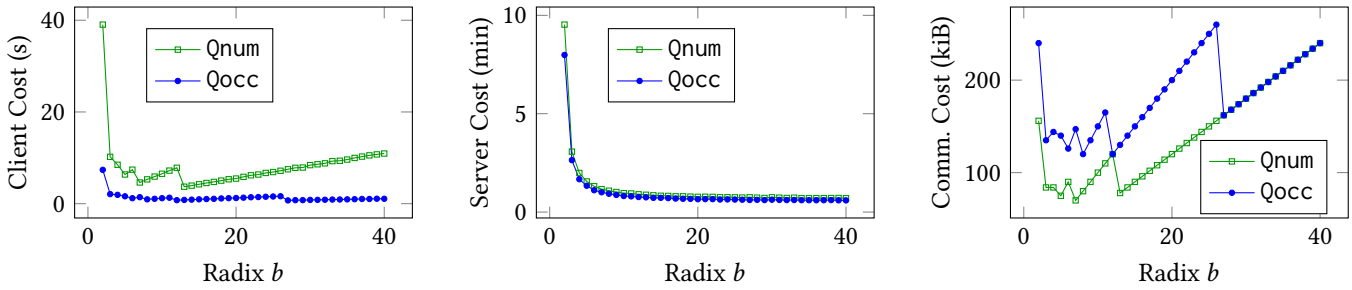

Figure 4: Performance of our PPSS protocol as a function of the radix b employed in the PIR algorithms. Private search of $q = CTGCAG$ in a genome with $500k$ nucleotides

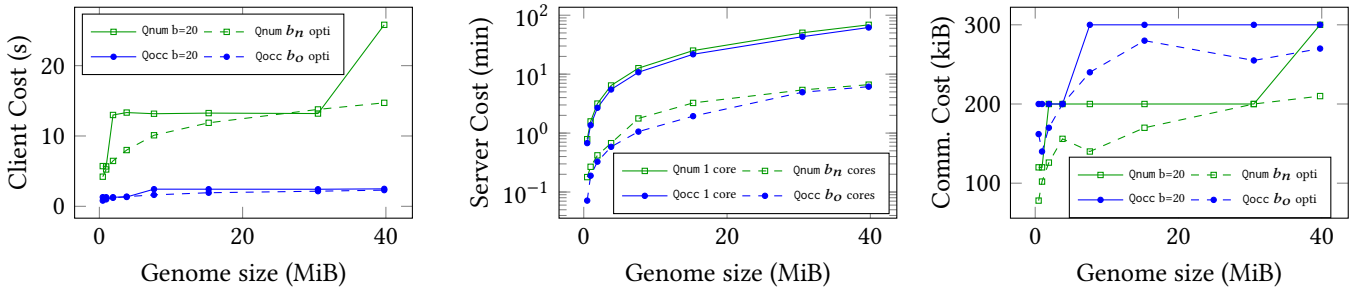

Figure 5: Performance of our PPSS protocol as a function of the genomic document size to find one occurrence of the substring $q = CTGCAG$. Considering each document size in increasing order, the optimal values of radixes b_n and b_o employed during the experiments are $\{13, 17, 21, 26, 14, 17, 20, 21\}$ and $\{27, 14, 17, 20, 24, 28, 17, 18\}$, respectively

our results refer to a single round of communication. In Figure 4 a remotely stored string with length equal to $500 \cdot 10^3$ characters is considered, and the client, server and communication costs are shown as a function of the radix b employed in the Lipmaa's PIR algorithm. As expected, increasing values of b allows to significantly decrease the computational cost on server side; conversely, the client and communication costs, which include a factor $O(b \log_b^2(n))$ (see Section 3.3), increase with the values of b, save for small values of b. The results suggest that the optimal value of b must be found considering the overall response time of a query, and should be differentiated between the phases `Qnum` and `Qocc` of the `Query` procedure as b_n and b_o, respectively.

In the next batch of tests, we consider a single-core implementation where we employ the same value $b = 20$ for genomes of increasing size, to observe how the performances are affected only by the size of the document collection. In addition, we consider also a multi-core implementation of the `Search` procedure of the Lipmaa's PIR protocol. Specifically, we employ a simple parallelization strategy which employs b cores to simultaneously compute all the b recursive calls of Algorithm 4. For these tests, we employ the optimal values b_n and b_o for each document size. The results of these tests are shown in Fig. 5. Regarding the server cost, we observe a linear trend in both the single-core (continuous lines in Fig. 5) and the multi-core implementations (dashed lines in Fig. 5); nevertheless, the multi-core implementation is at least one order of magnitude faster than the single-core, achieving much more practical performances (i.e., approximately 5 minutes to search for the substring $q = CTGCAG$ in a $40 \cdot 10^6$ characters document containing the whole chromosome).

The client and communication costs show the expected poly-logarithmic trend which allows to exchange kilobytes of data to search for the occurrences of $q = CTGCAG$ in the whole chromosome. Furthermore, in Fig. 5 the dashed lines on plots reporting the client and communication costs show the benefits of employing specific values b_n and b_o tailored for the size of the document.

Willing to compare the execution time of our protocol with the one of the BWT-based substring-search procedure outlined in Algorithm 1 (that features no security guarantees), we focused on querying a single occurrence of the substring $q = CTGCAG$ in the outsourced document. The experiment showed an execution time for Algorithm 1 equal to a few microseconds. We remark that querying for a single occurrence of q makes the computational complexity of Algorithm 1 unrelated to the size of the outsourced document, while the PIR-based `Query` procedure outlined in Algorithm 3 has a

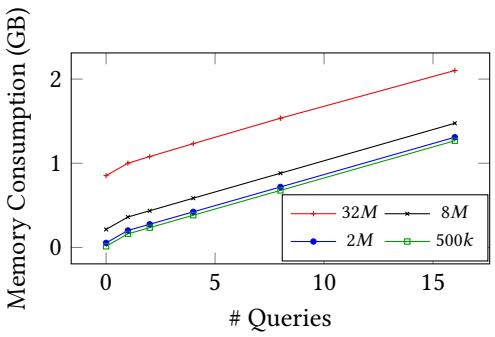

Figure 6: Memory consumption of our PPSS protocol when multiple simultaneous queries are performed. Each line represent a genome with a different size

computational complexity depending linearly on the size of the outsourced document.

Lastly, willing to verify the limited memory consumption when multiple-queries are simultaneously performed, we run each query in a separate thread, measuring the memory consumption of the process, as exposed by the process record in Linux's `proc` virtual filesystem. Figure 6 shows that as the number of simultaneous queries is increased, the memory consumption increases keeping (roughly) the same rate for the four dataset sizes considered. These results agree with the asymptotic spatial evaluations reported at the end of Section 4, where substantial storage savings w.r.t. replicating the whole data structure per-query, are discussed.

7 CONCLUDING REMARKS

We presented the first substring search protocol with proven guarantees of search and access pattern privacy that enables the simultaneous execution of queries from multiple users without the need of the data owner being online, and exhibiting a sub-linear (poly-logarithmic) communication cost per user. Our experimental validation with a case study on genomic data shows practical execution times and communication costs, and highlights the possibility of achieving significant benefits tuning the radix b for Lipmaa's PIR element representation. As interesting further developments, we will investigate how to reduce the overall query response time by employing different parameters and a different FLAHE scheme as building block of the Lipmaa's PIR. Indeed, in this work we aimed at minimizing the communication cost of the protocol, while results reported in [21] suggests that in some scenarios the overall query response time is improved by tuning the PIR parameters to tradeoff a low-bandwidth for significant computational savings at server side, and/or to employ a lattice-based FLAHE cryptoscheme as building block of the Lipmaa's PIR instead of generalized Pailler one.

ACKNOWLEDGEMENTS

This work was supported in part by the EU Commission grant: "WorkingAge" (H2020 RIA) Grant agreement no. 826232.

REFERENCES

[1] Pierre Baldi, Roberta Baronio, Emiliano De Cristofaro, Paolo Gasti, and Gene Tsudik. 2011. Countering GATTACA: efficient and secure testing of fully-sequenced human genomes. In *Proc. of the 18th ACM Conf. on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, Y. Chen, G. Danezis, and V. Shmatikov (Eds.). ACM, 691–702. https://doi.org/10.1145/2046707.2046785

[2] John Black and Phillip Rogaway. 2002. Ciphers with Arbitrary Finite Domains. In *Topics in Cryptology - CT-RSA 2002, The Cryptographer's Track at the RSA Conf., 2002, San Jose, CA, USA, February 18-22, 2002, Proc. (Lecture Notes in Computer Science)*, Bart Preneel (Ed.), Vol. 2271. Springer, 114–130. https://doi.org/10.1007/3-540-45760-7_9

[3] Christoph Bösch, Pieter H. Hartel, Willem Jonker, and Andreas Peter. 2014. A Survey of Provably Secure Searchable Encryption. *ACM Comput. Surv.* 47, 2 (2014), 18:1–18:51. https://doi.org/10.1145/2636328

[4] Michael Burrows and David Wheeler. 1994. *A block-sorting lossless data compression algorithm.* Technical Report. Digital Equipment Corporation. 18 pages. http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.pdf

[5] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. 2015. Leakage-Abuse Attacks Against Searchable Encryption. In *Proc. of the 22nd ACM SIGSAC Conf. on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, Indrajit Ray, Ninghui Li, and Christopher Kruegel (Eds.). ACM, 668–679. https://doi.org/10.1145/2810103.2813700

[6] Melissa Chase and Emily Shen. 2015. Substring-Searchable Symmetric Encryption. *PoPETs* 2015, 2 (2015), 263–281. http://www.degruyter.com/view/j/popets.2015.2015.issue-2/popets-2015-0014/popets-2015-0014.xml

[7] P.J. Cock, C.J. Fields, N. Goto, M.L. Heuer, and P.M. Rice. 2010. The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. *Nucleic Acids Research* 38, 6 (2010), 1767–1771. https://doi.org/10.1093/nar/gkp1137

[8] Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. 2006. Searchable symmetric encryption: improved definitions and efficient constructions. In *Proc. of the 13th ACM Conf. on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, October 30 - November 3, 2006*, A. Juels, R. N. Wright, and S. De Capitani di Vimercati (Eds.). ACM, 79–88. https://doi.org/10.1145/1180405.1180417

[9] Ivan Damgård and Mads Jurik. 2001. A Generalisation, a Simplification and Some Applications of Paillier's Probabilistic Public-Key System. In *Public Key Cryptography, 4th Intl. Workshop on Practice and Theory in Public Key Cryptography, PKC 2001, Cheju Island, Korea, February 13-15, 2001, Proc. (Lecture Notes in Computer Science)*, Kwangjo Kim (Ed.), Vol. 1992. Springer, 119–136. https://doi.org/10.1007/3-540-44586-2_9

[10] Sebastian Faust, Carmit Hazay, and Daniele Venturi. 2018. Outsourced pattern matching. *Int. J. Inf. Sec.* 17, 3 (2018), 327–346. https://doi.org/10.1007/s10207-017-0374-0

[11] Paolo Ferragina and Giovanni Manzini. 2005. Indexing compressed text. *J. ACM* 52, 4 (2005), 552–581. https://doi.org/10.1145/1082036.1082039

[12] Paul Flicek et. al. 2000. *Ensembl Genome Browser.* www.ensembl.org/.

[13] Craig Gentry. 2009. Fully homomorphic encryption using ideal lattices. In *Proc. of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, Michael Mitzenmacher (Ed.). ACM, 169–178. https://doi.org/10.1145/1536414.1536440

[14] Torbjörn Granlund and the GMP development team. 2012. *GNU MP: The GNU Multiple Precision Arithmetic Library.* http://gmplib.org/.

[15] Florian Hahn, Nicolas Loza, and Florian Kerschbaum. 2018. Practical and Secure Substring Search. In *Proc. of the 2018 Intl. Conf. on Management of Data, SIGMOD Conf. 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 163–176. https://doi.org/10.1145/3183713.3183754

[16] Yu Ishimaki, Hiroki Imabayashi, and Hayato Yamana. 2017. Private Substring Search on Homomorphically Encrypted Data. In *2017 IEEE Intl. Conf. on Smart Computing, SMARTCOMP 2017, Hong Kong, China, May 29-31, 2017*. IEEE Computer Society, 1–6. https://doi.org/10.1109/SMARTCOMP.2017.7947038

[17] Ben Kaduk et. al. 2015. *OpenSSL – Cryptography and SSL/TLS Toolkit*. https://www.openssl.org.

[18] Iraklis Leontiadis and Ming Li. 2018. Storage Efficient Substring Searchable Symmetric Encryption. In *Proc. of the 6th Intl. Workshop on Security in Cloud Computing, SCC@AsiaCCS 2018, Incheon, Republic of Korea, June 04-08, 2018*, Aziz Mohaisen and Qian Wang (Eds.). ACM, 3–13. https://doi.org/10.1145/3201595.3201598

[19] Helger Lipmaa. 2005. An Oblivious Transfer Protocol with Log-Squared Communication. In *Information Security, 8th Intl. Conf., ISC 2005, Singapore, September 20-23, 2005, Proc. (Lecture Notes in Computer Science)*, J. Zhou, J. López, R. H. Deng, and F. Bao (Eds.), Vol. 3650. Springer, 314–328. https://doi.org/10.1007/11556992_23

[20] Nicholas Mainardi. 2019. *Privacy Preserving Substring Search Protocol with Polylogarithmic Communication Cost – Software implentation*. https://dx.doi.org/10.5281/zenodo.3384814.

[21] Carlos Aguilar Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. 2016. XPIR: Private Information Retrieval for Everyone. *PoPETs* 2016, 2 (2016). https://doi.org/10.1515/popets-2016-0010

[22] Tarik Moataz and Erik-Oliver Blass. 2015. Oblivious Substring Search with Updates. *IACR Cryptology ePrint Archive* 2015 (2015), 722. http://eprint.iacr.org/2015/722

[23] Pascal Paillier. 1999. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *Advances in Cryptology - EUROCRYPT '99, Intl. Conf. on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding (Lecture Notes in Computer Science)*, Jacques Stern (Ed.), Vol. 1592. Springer, 223–238. https://doi.org/10.1007/3-540-48910-X_16

[24] Cédric Van Rompay, Refik Molva, and Melek Önen. 2017. A Leakage-Abuse Attack Against Multi-User Searchable Encryption. *PoPETs* 2017, 3 (2017), 168. https://doi.org/10.1515/popets-2017-0034

[25] Kana Shimizu, Koji Nuida, and Gunnar Rätsch. 2016. Efficient privacy-preserving string search and an application in genomics. *Bioinformatics* 32, 11 (2016). https://doi.org/10.1093/bioinformatics/btw050

[26] Dawn Xiaodong Song, David A. Wagner, and Adrian Perrig. 2000. Practical Techniques for Searches on Encrypted Data. In *2000 IEEE Symposium on Security and Privacy, Berkeley, California, USA, May 14-17, 2000*. IEEE Computer Society, 44–55. https://doi.org/10.1109/SECPRI.2000.848445

[27] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: an extremely simple oblivious RAM protocol. In *2013 ACM SIGSAC Conf. on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung (Eds.). ACM, 299–310. https://doi.org/10.1145/2508859.2516660

[28] Mikhail Strizhov, Zachary Osman, and Indrajit Ray. 2016. Substring Position Search over Encrypted Cloud Data Supporting Efficient Multi-User Setup. *Future Internet* 8, 3 (2016). https://doi.org/10.3390/fi8030028

[29] Marc Tiehuis. 2015. *libhcs: A partially Homomorphic C library*. https://github.com/tiehuis/libhcs/tree/master/include/libhcs.

[30] Bing Wang, Wei Song, Wenjing Lou, and Y. Thomas Hou. 2017. Privacy-preserving pattern matching over encrypted genetic data in cloud computing. In *2017 IEEE Conf. on Computer Communications, INFOCOM 2017, Atlanta, GA, USA, May 1-4, 2017*. IEEE, 1–9. https://doi.org/10.1109/INFOCOM.2017.8057178

A SECURITY PROOF

Theorem 5.3 is proven by showing the existence of a simulator $\mathcal{S}$ which interacts with any semi-honest adversary $\mathcal{A}$, according to the $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}$ experiment of Definition 5.2, to produce transcript for this experiment which is computationally indistinguishable from the transcript of the $\mathbf{Real}_{\mathcal{P},\mathcal{A}}$ experiment, where $\mathcal{A}$ interacts with a client through our PPSS protocol. As the simulator $\mathcal{S}$ knows only the leakage $\mathcal{L}$ as defined in Theorem 5.3, the transcript of the $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}$ experiment necessarily depends only on the leakage; thus, if this transcript is computationally indistinguishable from the one of the $\mathbf{Real}_{\mathcal{P},\mathcal{A}}$ experiment, then it necessarily means that no other information than $\mathcal{L}$ can be inferred from the latter transcript, as otherwise this additional information could be exploited by the adversary to distinguish between the two experiments. Since the transcript of the $\mathbf{Real}_{\mathcal{P},\mathcal{A}}$ experiment corresponds to the information observed and derived by the adversary in our PPSS protocol, then no other information than $\mathcal{L}$ can be inferred from the adversary in our PPSS protocol, in turn proving that the protocol leaks no more information than $\mathcal{L}$ to the adversary. For the sake of clarity, in the following we denote all the variables involved in the $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}$ experiment with a superscript Id (e.g., $[[\mathbf{D}]]^{Id}$ is the privacy-preserving representation $[[\mathbf{D}]]$ computed by the simulator $\mathcal{S}_D$).

Simulator Construction. We now show how to construct the simulator $\mathcal{S}$. Specifically, for a document collection $\mathbf{D}$ of z documents $D_1, \ldots, D_z$ and a string q, $\mathcal{S}$ is realized by constructing two simulators $\mathcal{S}_D$ and $\mathcal{S}_q$. The former employs the leakage $\mathcal{L}_D$ to build a privacy-preserving representation $[[\mathbf{D}]]^{Id}$ which is computationally indistinguishable from the privacy-preserving representation $[[\mathbf{D}]]$ computed by the client in our PPSS protocol. The latter simulator employs both the leakage $\mathcal{L}_D$ and $\mathcal{L}_q$ to build a trapdoor $[[q]]^{Id}_j$, $j = 1, \ldots, w$ for each of the w rounds of the `Query` procedure for the string q; all these trapdoors must be computationally indistinguishable from the trapdoors constructed by the client in the w rounds of our PPSS protocol.

- $\mathcal{S}_D$. Given the leakage $\mathcal{L}_D = (\sum_{i=1}^{z}(\mathtt{len}(D_i) + 1), \omega)$, where the first term $\sum_{i=1}^{z}(\mathtt{len}(D_i) + 1)$ is denoted in the following as n, the simulator constructs two arrays SA^{Id} and C^{Id} with, respectively, $n + 1$ and $(n + 1) \cdot (|\Sigma| + 1)$ elements (we assume that the alphabet Σ for the documents in $\mathbf{D}$ is publicly known); each entry of these arrays contains a randomly generated string of ω bits. Lastly, the simulator outputs the privacy-preserving representation $[[\mathbf{D}]]^{Id} = (C^{Id}, SA^{Id})$

- $\mathcal{S}_q$. Given the leakages $\mathcal{L}_D$, $\mathcal{L}_q = (\texttt{len}(q), b, |O_{\mathbf{D},q}|)$ and the public modulus N for the FLAHE Paillier scheme employed by the client in the $\mathbf{Real}_{\mathcal{P},\mathcal{A}}$ experiment, the simulator simulator computes the values $t_C = \lceil \log_b(n+1) \rceil$ and $t_{SA} = \lceil \log_b((n+1)\cdot(|\Sigma|+1)) \rceil$. Then, the simulator constructs $m = \texttt{len}(q)$ trapdoors $[[q]]_1^{Id}, \ldots, [[q]]_m^{Id}$ as follows. Each trapdoor is an array with $b \cdot t_C$ elements, where the first b entries are integers randomly sampled in $\mathbb{Z}^*_{N^2}$, then the subsequent b entries are integers randomly sampled in $\mathbb{Z}^*_{N^3}$: in general, the j-th entry contains an integer randomly sampled in $\mathbb{Z}^*_{N^{\lceil \frac{j}{b} \rceil + 1}}$. Subsequently, the simulator generates $o_q = |O_{\mathbf{D},q}|$ trapdoors $[[q]]_{m+1}^{Id}, \ldots, [[q]]_{m+o_q}^{Id}$, where each trapdoor is an array with $b \cdot t_{SA}$ elements constructed in the same manner as the previous m trapdoors (i.e., the j-th entry contains an integer randomly sampled in $\mathbb{Z}^*_{N^{\lceil \frac{j}{b} \rceil + 1}}$).

We now prove that, for any probabilistic polynomial time adversary $\mathcal{A}$, the output of the $\mathbf{Real}_{\mathcal{P},\mathcal{A}}$ experiment is computationally indistinguishable from the output of the $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}$ experiment when the simulator $\mathcal{S}$ we have just constructed is employed. Specifically, we analyze each step of the two experiments and we show that the adversary cannot distinguish the simulator from a legitimate client of our PPSS protocol. In both the experiments, the adversary initially chooses a document collection $\mathbf{D}$ of z documents over a publicly known alphabet Σ. In the $\mathbf{Real}_{\mathcal{P},\mathcal{A}}$ experiment, $\mathbf{D}$ is sent to the client, which constructs a privacy-preserving representation $[[\mathbf{D}]]$ by running the `Setup` procedure of our PPSS protocol; specifically, $[[\mathbf{D}]]$ is composed by two cell-wise encrypted arrays $\langle C \rangle$ and $\langle SA \rangle$ with, respectively, $(n+1)\cdot(|\Sigma|+1)$ and $n+1$ elements. Conversely, in the $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}$ experiment, the simulator $\mathcal{S}_D$ obtains the leakage $\mathcal{L}_D$ and constructs the privacy-preserving representation $[[\mathbf{D}]]^{Id}$ as two arrays C^{Id}, SA^{Id} whose size is the same as $\langle C \rangle$, $\langle SA \rangle$, respectively. The semantic security of the scheme $\mathcal{E}$ employed to encrypt $\langle C \rangle$ and $\langle SA \rangle$ in our PPSS protocol guarantees that a ciphertext of ω bits computed by $\mathcal{E}.\texttt{Enc}$ is computationally indistinguishable from a random bit string of size ω, which implies that the two privacy-preserving representations $[[\mathbf{D}]]$ and $[[\mathbf{D}]]^{Id}$ are computationally indistinguishable too.

After receiving the privacy-preserving representations $[[\mathbf{D}]]$ and $[[\mathbf{D}]]^{Id}$, the adversary chooses a string q_1. In the $\mathbf{Real}_{\mathcal{P},\mathcal{A}}$ experiment, the string q_1 is sent to the client, which employs the `Query` procedure of our PPSS protocol to find all the positions of the occurrences of q_1 in $\mathbf{D}$. In each of the w rounds of the `Query` procedure, the client employs the `Trapdoor` procedure to generate a trapdoor $[[q_1]]_j$, $j = 1, \ldots, w$, which corresponds to a trapdoor in the Lipmaa's PIR protocol. In the $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}$ experiment, the simulator $\mathcal{S}_{q_1}$ receives the leakage $\mathcal{L}_{q_1}$, which is employed to build a trapdoor $[[q_1]]_j^{Id}$, $j = 1, \ldots, w$ for each of the w rounds. The semantic security of the FLAHE Paillier scheme guarantees that a ciphertext computed by the encryption procedure with length l (i.e., $\texttt{FLAHE.E}^l_{pk}$) is computationally indistinguishable from a random integer in $\mathbb{Z}^*_{N^{l+1}}$, which means that the set of trapdoors $[[q_1]]_j$ are computationally indistinguishable from the set of trapdoors $[[q_1]]_j^{Id}$.

Subsequently, in the $\mathbf{Real}_{\mathcal{P},\mathcal{A}}$ (resp. $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}$) experiment, the trapdoor $[[q_1]]_j$ (resp. $[[q_1]]_j^{Id}$) generated by the client (resp. $\mathcal{S}_{q_1}$) in each of the w rounds is received by the adversary which employs the `Search` procedure of Lipmaa's PIR protocol to compute a ciphertext $[[res_j]]$ (resp. $[[res_j]]^{Id}$). The semantic security of the FLAHE Paillier scheme guarantees that all the intermediate values computed by each homomorphic operation of the `Search` procedure in the $\mathbf{Real}_{\mathcal{P},\mathcal{A}}$ experiment are computationally indistinguishable from the corresponding intermediate values in the $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}$ experiment. Indeed, in the former experiment, given two ciphertext c_1 and c_2 in $\mathbb{Z}^*_{N^l}$ for the FLAHE Paillier scheme, each *homomorphic addition* computes $c_{add} = c_1 \cdot c_2 \bmod N^l$, with c_{add} being a ciphertext in $\mathbb{Z}^*_{N^l}$; in the latter experiment, the homomorphic addition multiplies two random integers in $\mathbb{Z}^*_{N^l}$, obtaining a new random integer in $\mathbb{Z}^*_{N^l}$ which is computationally indistinguishable from c_{add}. Similarly, in the $\mathbf{Real}_{\mathcal{P},\mathcal{A}}$ experiment, given a ciphertext $c_1 \in \mathbb{Z}^*_{N^l}$ and a ciphertext $c_2 \in \mathbb{Z}^*_{N^{l+1}}$, each *hybrid homomorphic multiplication* computes $c_{hmul} = c_2^{c_1} \bmod N^{l+1}$, with c_{hmul} being a ciphertext in $\mathbb{Z}^*_{N^{l+1}}$; In the $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}$ experiment, each hybrid homomorphic multiplication computes the exponentiation between a random integer in $\mathbb{Z}^*_{N^{l+1}}$ and a random integer in $\mathbb{Z}^*_{N^l}$, obtaining a new random integer in $\mathbb{Z}^*_{N^{l+1}}$ which is computationally indistinguishable from c_{hmul}. Therefore, as the `Search` procedure of Lipmaa's PIR performs only homomorphic operations, we conclude that all values (including the outcomes $[[res_j]]$ and $[[res_j]]^{Id}$) observed by the adversary throughout this computation in the $\mathbf{Real}_{\mathcal{P},\mathcal{A}}$ and $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}$ experiments are computationally indistinguishable. In conclusion, the adversary cannot distinguish an interaction with a legitimate client in our PPSS protocol from an interaction with the simulator $\mathcal{S}_{q_1}$ for the first query q_1.

We note that the same reasoning allows to prove that all the trapdoors and the intermediate values observed by the adversary in the subsequent $d-1$ queries in the two experiments are computationally indistinguishable. Indeed, in each query, the simulator simply needs to construct trapdoors which looks like generic FLAHE Paillier ciphertexts as computed by the legitimate client in our PPSS protocol independently from their corresponding plaintext value, as the semantic security of the scheme hides any information about the encrypted information stored in these trapdoors.