Privacy-aware character pattern matching over outsourced encrypted data

NICHOLAS MAINARDI*, Politecnico di Milano – DEIB
ALESSANDRO BARENGHI*, Politecnico di Milano – DEIB
GERARDO PELOSI*, Politecnico di Milano – DEIB

Providing a method to efficiently search into outsourced encrypted data, without forsaking strong privacy guarantees, is a pressing concern rising from the separation of data ownership and data management typical of cloud-based applications. While several existing solutions allow a client to look-up the occurrences of a substring in an outsourced document collection, the practical application requirements in terms of privacy and efficiency call for the improvement of such solutions. In this work, we present a privacy-preserving substring search protocol with a polylogarithmic communication cost and a limited computational effort on the server side. The proposed protocol provides search pattern and access pattern privacy, for both exact string search, and character-pattern search with wildcards. 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 50 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.*

Additional Key Words and Phrases: Secure substring search, Cryptography, Homomorphic encryption, Privacy-preserving protocol

ACM Reference Format:
Nicholas Mainardi, Alessandro Barenghi, and Gerardo Pelosi. 2020. Privacy-aware character pattern matching over outsourced encrypted data. *Digit. Threat. Res. Pract.* 00, 0, Article 000 (2020), 39 pages. https://doi.org/10.1145/xxxyyyx.yyyxxxy

1 INTRODUCTION

The significant improvements in reliability and total cost of ownership provided by remote data management services have proven on field to be beneficial to a large variety of enterprises. In this context, a company relies on cloud based services to store a significant amount of its own data in an infrastructure located in a third party data centre, beyond the means of its direct control. While this has significant benefits in terms of the reduction

*This article extends the previous work by the authors appeared under the title "Privacy Preserving Substring Search Protocol with Polylogarithmic Communication Cost," in Proceedings of the 35th Annual Computer Security Applications Conference, ACSAC 2019, San Juan, PR, USA, December 09-13, 2019 [27].

Authors' addresses: Nicholas Mainardi, Politecnico di Milano – DEIB, Piazza Leonardo da Vinci, 32, Milano, Italy, 20133, nicholas.mainardi@polimi.it; Alessandro Barenghi, Politecnico di Milano – DEIB, Piazza Leonardo da Vinci, 32, Milano, Italy, 20133, alessandro.barenghi@polimi.it; Gerardo Pelosi, Politecnico di Milano – DEIB, Piazza Leonardo da Vinci, 32, Milano, Italy, 20133, gerardo.pelosi@polimi.it.

2576-5337/2020/0-ART000 $15.00
https://doi.org/10.1145/xxxyyyx.yyyxxxy

of the efforts to be made for infrastructural maintenance, it comes with confidentiality (secrecy) and privacy concerns acting as stopgaps to the adoption of cloud based storage solutions.

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 queries over the data, such as searching for a given pattern, with practical performance. 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.

Problem Statement. A data owner outsources a set of documents $\mathbf{D} = \{D_1, \ldots, D_z\}$, $z \geq 1$, where each document is a sequence of symbols (*string*) over an alphabet Σ with length $\texttt{len}(D_i)$, encrypted with a cryptographic primitive of choice. 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) [36] or Private Information Retrieval (PIR) protocols [25].

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 occurrences of the queried substring in $\mathbf{D}$.

Prior Art Approaches. The seminal work on searching over data in an encrypted state [35] (a.k.a. *searchable encryption* schemes), as well as many of the subsequent improvements in terms of computational and communication resources [3, 9], relies on pre-registering searchable keywords, without supporting free form searching over the encrypted data. Such a limitation is overcome by *substring searchable encryption* schemes [6, 11, 18, 23, 38]. These schemes exhibit computation/communication complexities linear or quadratic in the length of the searched substring (hence independent from the size of the document collection), with different server side storage savings and assumptions on the adversary capabilities. Notably, the protocol described in [11] is the only one that enables querying on patterns including the wildcard symbol ?, which acts as a placeholder in matching an arbitrary single character. Furthermore, since the design of this protocol stems from an existing *dynamic* searchable encryption scheme, it is the only substring searchable encryption scheme that allows to efficiently add new documents to the collection without re-encrypting significant portions of the search index employed for queries. While these solutions cope with the problem of substring search, the works in [6, 11, 23, 38] do not provide protection of search and access pattern, while the information leakage shown in [18] 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, 33], 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, 18, 23, 38] employ symmetric-key or order-preserving cryptographic primitives, obtaining practical 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 privacy 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 or 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 preserving both search and access pattern confidentiality and enabling queries for patterns containing wildcard characters. 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 [25], which is based on the generalized Pailler homomorphic encryption scheme proposed in [10], because of its limited communication cost. This design leads to a PPSS protocol with $O(m{+}o_q)$ communication rounds and an $O((m{+}o_q)\log^2 n)$ communication cost between client and service provider, where o_q denotes the number of occurrences of the queried string q ($m{=}\texttt{len}(q)$) over the document collection $\mathbf{D} = \{D_1, \ldots, D_z\}$, $z \geq 1$, and $n{=}\sum_{i=1}^{z} \texttt{len}(D_i)$. Our PPSS protocol exhibits an $O((m{+}o_q)\log^4 n)$ 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 $O((m{+}o_q)n)$ and $O(n)$, respectively. An enhanced version of the same protocol allows to retrieve all o_q occurrences of a queried string q in a single communication round instead of o_q ones, making the computational cost at server side also independent from o_q. This version improves all the figures of merit exhibiting an $O(m\log^2(n){+}o_q\log(n))$ communication cost, an $O(m\log^4(n){+}o_q\log^4(\frac{n}{o_q}))$ computational cost at client side and an $O(m{\cdot}n)$ computational cost at server side. 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. Finally, to make our PPSS protocol resilient against accidental or misconfiguration errors, we complement its features with an efficient mechanism that allows the client to verify the correctness of the remotely accessed data.

2 RELATED WORK

We now briefly revise existing PPSS protocols, comparing their functionalities and performance with our solution and its enhanced version. An overview of such comparison is reported in Tab. 1. In [40], 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 [34] described how to use the Burrows Wheeler Transform (BWT) [4] and Pailler's additive homomorphic encryption (AHE) scheme [31] 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 $O((m{+}o_q)\sqrt{n})$ ciphertexts from client to server. Such a cost was reduced by Ishimaki *et. al.* [21] to $O((m{+}o_q)\log(n))$, at the price of employing a fully homomorphic encryption (FHE) scheme [16], 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 $O((m{+}o_q)n\log(n))$, which also hides a large constant overhead (about 10^6) required to compute on FHE ciphertexts.

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 [21] hides a large constant factor C, e.g., $C \geq 16 \times 10^6$, for providing 80-bit security parameters
‡ Improved version of our protocol featuring batching retrieval of occurrences
§ Search and Access pattern privacy

PPSS Protocol	Communication Cost	Server Cost	S. & A.§ Privacy	Data Owner Off-line	Multi User	Wildcard Queries	Adversary
[40]	$O(n)$	$O(m \cdot n)$	✓	✓	×	×	Semi-honest
[34]	$O((m+o_q)\sqrt{n})$	$O((m+o_q)n)$	✓	✓	×	×	Semi-honest
[21]	$O(C(m+o_q)\log(n))^{\dagger}$	$O(C(m+o_q)n\log n)^{\dagger}$	✓	✓	×	×	Semi-honest
[12]	$O((m+o_q)\log(n))$	$\Omega((\frac{n}{m}+o_q)\log(n))$	S.✓A.×	×	✓	×	Malicious
[32]	$O(m+o_q)$	$O(m \cdot n)$	S.✓A.×	✓	✓	×	Semi-honest
[30]	$\Omega(m\log^5(n)+o_q\log^2(n))$	$\Omega(m\log^5(n)+o_q\log^2(n))$	✓	✓	×	×	Semi-honest
[28]	$O(m+o_q)$	$O((m+o_q)\log^3(n))$	✓	✓	×	×	Malicious
[24]	$O(m+o_q)$	$O(n)$	S.✓A.×	×	✓	✓	Semi-honest
[41]	$O(m+o_q)$	$O(n)$	S.✓A.×	✓	×	✓	Semi-honest
Ours	$O((m+o_q)\log^2(n))$	$O(m+o_q)n$	✓	✓	✓	✓	Semi-honest
Ours+‡	$O(m\log^2(n)+o_q\log(n))$	$O(m \cdot n)$	✓	✓	✓	✓	Semi-honest

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 [12]. 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. This protocol has been recently improved in [32], removing the interaction between the client and the data owner during queries. Remarkably, this is the first substring search protocol that allows multiple users to perform queries over data coming from multiple data owners with an access control mechanism that allows to restrict, for each document, the users authorized to perform queries. Nonetheless, the protocol is still affected by a lack of access pattern privacy; furthermore, the client must perform a distinct query for each document the client is interested in.

The suffix-array based solutions proposed by Moataz *et. al.* in [30] 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 [36] – 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 [30] 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 $\log^2(n)$. Each of the o_q occurrences can be retrieved with o_q accesses to the ORAM, yielding an additional $O(o_q\log^2(n))$ communication cost.

ObSQRE [28] is the first PPSS protocol with optimal communication cost concealing both the search and the access patterns. This goal is achieved by relying on Intel SGX technology [8], which allows to execute an application on an untrusted machine while guaranteeing the confidentiality and the integrity of the application code and data. Nonetheless, since this technology is based on trusted hardware modules provided by Intel, ObSQRE can only be deployed on servers equipped with Intel CPUs where such hardware is available. Furthermore, ObSQRE

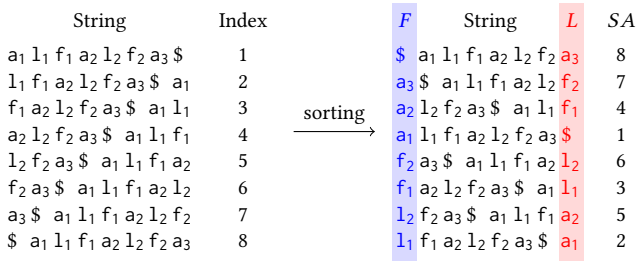

Fig. 1. Burrows Wheeler Transform L and Suffix Array SA of the string alfalfa

lacks two important features provided by our PPSS protocol: indeed, it supports neither simultaneous queries from multiple users, nor queries for patterns containing wildcard characters.

A multi-user PPSS protocol allowing queries with wildcard characters was first proposed in [24]. This solution allows users to identify the documents in the collection that belong to the language specified by a Deterministic Finite Automaton (DFA). Specifically, authorized users can independently send an obfuscated version of the DFA to the server, which computes the set of matched documents, i.e., the ones accepted by the DFA. Therefore, this PPSS protocol enables the matching of any regular pattern in the document collection, making it more expressive than our solution. Nonetheless, although the structure of the DFA is concealed to the server in such a way to prevent search pattern leakage, the server learns which documents are matched, hereby leaking the access pattern. Furthermore, a relevant limitation of this protocol is the need for each user to interact with a trusted authority to build a properly obfuscated search query (relying on a master secret key known only to the trusted authority). The said interaction between a user and the trusted authority has been removed in a further modification of this protocol proposed in [41], which however requires the data owner to encrypt the documents employing a pairing-based cryptosystem (specifically designed by the authors) with a per-user public key, hence requiring a distinct per-user encrypted copy of the outsourced document collection. Because of this limitation, we do not classify in Tab. 1 this solution as a multi-user PPSS protocol.

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, $L = \mathtt{BWT}(s)$, 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 [13]. The substring search algorithm has a linear time complexity in the length of the substring to be searched for, and it 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 we define as *index* of a substring in s the position of its leading character in the original string, counting from 1 onwards.

As shown in Fig. 1, taking as an example $s = \texttt{alfalfa\$}$, 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 Fig. 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, and by $succ_s(c)$ the character subsequent to c in the string s, 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, Alg. 1 computes the positions of the 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 first i characters of L, i.e., the substring $L[1, \ldots, i]$. 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 identified by $\alpha+1$ and β, which denote the positions of the first and the last of them in F, respectively. 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 $q[m-1, m]$.

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 $L[1], \ldots, L[i]$, $1 \le i \le n$.
Rank, dictionary of size $|\Sigma|+1$, of pairs $\langle c, l \rangle$, with $c \in \Sigma$, l=Rank(c), $0 \le l \le n+1$ number of chars in s smaller than c.
SA, suffix array with length $n+1$ of the string s;
q, a substring with length $1 \le m \le n$.

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

```
1  c ← q[m]
2  α ← Rank(c),  β ← α + M[c][n + 1]
3  for i ← m − 1 downto 1 do
4      c ← q[i],  r ← Rank(c)
5      α ← r + M[c][α]
6      β ← r + M[c][β]
7  R_q ← ∅
8  for i ← α + 1 to β do
9      R_q ← R_q ∪ {SA[i]}
10 return R_q
```

In particular, all the repetitions of $q[m{-}1]$ among the predecessors of $q[m]$ in $s = \mathtt{BWT}^{-1}(L)$ 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 $q[m-1]_{\mathtt{first}}$ and $q[m-1]_{\mathtt{last}}$. 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 $pos_F(q[m{-}1]_{\mathtt{first}})$ and $pos_F(q[m{-}1]_{\mathtt{last}})$, which represent the updated values for $\alpha{+}1$ and β, respectively.

The value $pos_F(q[m{-}1]_{\mathtt{first}})$ can be obtained adding to the position of the leading character in F (i.e., 1) the number $r = \mathtt{Rank}(q[m{-}1])$ 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 $q[m-1]_{\mathtt{first}}$. 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.

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 $r = \mathtt{Rank}(q[m{-}1])$ 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 $F[i]{=}s[SA[i]]$, $1 \le i \le n+1$, the set R_q of integers in $SA[\alpha{+}1, \ldots, \beta]$ includes the position of the leading character of each repetition of q in s. Algorithm 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.: $s = D_1\$D_2\$ \ldots D_z\$$; each character of s is coupled with a pair $\langle pos, off \rangle$, where pos is the starting position in s of the document where the character at hand is found, and off is the offset of the said character in the document. It is easy to adapt Alg. 1 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 $n = \texttt{len}(s)$, a pair of values $\langle \texttt{pos}, \texttt{off} \rangle$. Each occurrence of q is thus identified by the pair $\langle \texttt{pos}, \texttt{off} \rangle$ relative to the leading character of the occurrence at hand; from such pair, it is trivial to both compute the position in s of any occurrence of q and to retrieve from the string s the entire document where an occurrence is found. If the application scenario requires also to obtain an identifier of the document where an occurrence is located, then the augmented suffix array may also store this id for each of its entries.

Algorithm 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):

- $(pk, sk, evk) \leftarrow \texttt{KeyGen}(1^\lambda)$ 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.
- $c \leftarrow \texttt{E}(pk, m)$, denoted also as $\texttt{E}_{pk}(c)$, 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.
- $m \leftarrow \texttt{D}(sk, c)$, denoted also as $\texttt{D}_{sk}(c)$, is a deterministic algorithm which, given the secret key sk and a ciphertext $c \in C$, recovers the plaintext value $m \in \mathcal{M}$.
- $c_{add} \leftarrow \texttt{Add}(evk, c_1, c_2)$, 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: $\forall m \in \mathcal{M}(\texttt{D}_{sk}(\texttt{E}_{pk}(m)) = m)$

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

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

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

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, as well as 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 [31], 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: $\forall m_1, m_2 \in \mathbb{Z}_N(\mathtt{D}_{sk}(\mathtt{E}_{pk}(m_1) \cdot \mathtt{E}_{pk}(m_2) \bmod N^2) = m_1 + m_2 \bmod N)$.

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:

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

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, both with $n \geq 1$ elements; this operation, referred to as *hybrid dot product*, is computed as follows:

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

An FLAHE variant is described in [10] 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, $\forall m_1 \in \mathbb{Z}_{N^{l_1}}, m_2 \in \mathbb{Z}_{N^{l_2}}$:

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

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 property of the `HybridMul` operation in the Paillier FLAHE scheme is extremely useful, as it allows to perform the *homomorphic dot product* between an array $\langle A \rangle_{l_2}$ encrypted cell-wise with length l_2, and an array $\langle B \rangle_{l_1}$ encrypted cell-wise with length l_1:

$$\mathtt{D}^{l_2}_{sk}\left(\prod_{i=1}^{n}(\langle A \rangle_{l_2}[i])^{\langle B \rangle_{l_1}[i]} \bmod N^{l_2+1}\right) = \sum_{i=1}^{n} A[i] \cdot \langle B \rangle_{l_1}[i] \bmod N^{l_2} \tag{2}$$

This homomorphic operation is at core of the *Private Information Retrieval* (PIR) protocol introduced by Lipmaa in [25], which, in turn, is an important building block of our PPSS protocol.

Algorithm 2: `PIR-Trapdoor` in Lipmaa's PIR

Function `PIR-Trapdoor`(h, b):
Input: h, $0 \le h < n$: Index of the element to be retrieved from remotely stored array A with n entries
$b \ge 2$: Integer employed as a radix to represent h
Output: $\langle h \rangle$: Trapdoor to privately retrieve $A[h]$
Data: pk: Public key of a Paillier FLAHE keypair
1 **begin**
2 $t \leftarrow \lceil \log_b(n) \rceil$
3 $\{h_0, \ldots, h_{t-1}\} \leftarrow$ `RadixDecompose(h,b)`
4 $\langle h \rangle \leftarrow \emptyset$
5 **for** $i \leftarrow 0$ **to** $t-1$ **do**
6 **for** $x \leftarrow 0$ **to** $b-1$ **do**
7 **if** $x = h_i$ **then**
8 $\texttt{hdigit}_i[x] \leftarrow \mathsf{E}_{pk}^{i+1}(1)$
9 **else**
10 $\texttt{hdigit}_i[x] \leftarrow \mathsf{E}_{pk}^{i+1}(0)$
11 $\langle h \rangle \leftarrow \langle h \rangle \cup \texttt{hdigit}_i$
12 **return** $\langle h \rangle$

Algorithm 3: `PIR-Search` in Lipmaa's PIR

Function `PIR-Search`(A, b, $\langle h \rangle$):
Input: A: remotely stored array with n entries
b: Radix employed to construct the trapdoor $\langle h \rangle$ in the `PIR-Trapdoor` procedure
$\langle h \rangle = \{\texttt{hdigit}_0, \ldots, \texttt{hdigit}_{t-1}\}$, $t \leftarrow \lceil \log_b(n) \rceil$: Trapdoor to privately retrieve $A[h]$
Output: $\langle A_{h_{t-1}} \rangle$: t-layered encryption of the requested entry $A[h]$
Data: N: Public modulus employed for homomorphic operations in Paillier FLAHE scheme
1 **begin**
2 $t \leftarrow \lceil \log_b(n) \rceil$
3 **for** $j = 0$ **to** $\lceil \frac{n}{b} \rceil - 1$ **do**
4 $\langle A_{h_0} \rangle[j] \leftarrow \prod_{z=0}^{b-1} \texttt{hdigit}_0[z]^{A[j \cdot b + z]} \bmod N^2$
5 **for** $i = 1$ **to** $t-1$ **do**
6 **for** $j = 0$ **to** $\lceil \frac{n}{b^{i+1}} \rceil - 1$ **do**
7 $\langle A_{h_i} \rangle[j] \leftarrow \prod_{z=0}^{b-1} \texttt{hdigit}_i[z]^{\langle A_{h_{i-1}} \rangle[j \cdot b + z]} \bmod N^{i+2}$
8 **return** $\langle A_{h_{t-1}} \rangle$

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, $0 \le h \le n-1$, 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 [25] assumes that both the client and the server read the positions of the cells of the array in positional notation with radix $b \ge 2$, i.e., an index h is represented by the sequence of $t = \lceil \log_b(n) \rceil$ digits $\{h_0, \ldots, h_{t-1}\}$, with each $h_i \in \{0, \ldots, b-1\}$, 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., $A_{h_0}[j] = A[j \cdot b + h_0]$, $0 \le j \le \lceil \frac{n}{b} \rceil - 1$. 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 $A_{h_1}[j] = A_{h_0}[j \cdot b + h_1] = A[j \cdot b^2 + h_1 \cdot b + h_0]$, $0 \le j \le \lceil \frac{n}{b^2} \rceil - 1$. 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 [25], the client initially generates a public/private Paillier FLAHE keypair (pk, sk) with a public modulus $N \ge 2^\omega$, 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, reported in Alg. 2, takes as input an integer $b \ge 2$ 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 $\langle h \rangle$, also referred to as *trapdoor*. The first step of the trapdoor computation considers the value h as the sequence of $t = \lceil \log_b(n) \rceil$ digits in b-radix positional representation (lines 2–3). Then, in the subsequent loop (lines 4–12), the algorithm computes the trapdoor $\langle h \rangle$ as the set of t arrays with b ciphertexts $\{\texttt{hdigit}_0, \ldots, \texttt{hdigit}_{t-1}\}$, each related to a digit h_i, $0 \le i \le t-1$, of h. Specifically, the i-th array has b ciphertexts of length $l = i+1$; all the b entries of such array encrypts the plaintext value $0 \in \mathbb{Z}_{N^l}$ (line 10), except for the entry $\texttt{hdigit}_i[h_i]$, which encrypts the plaintext value $1 \in \mathbb{Z}_{N^l}$(line 8). Since each of the t arrays has b elements of the ring $\mathbb{Z}^*_{N^{l+1}}$, then the size of the trapdoor $\langle h \rangle$ amounts to $\sum_{i=0}^{t-1} b(i+2)\log(N) = \Theta(bt^2 \log(N)) = \Theta(b \log_b^2(n) \log(N))$

Algorithm 4: `PIR-Retrieve` procedure in Lipmaa's PIR protocol

Function `PIR-Retrieve`(b, `ctx`):
Input: b: Radix employed to construct the trapdoor $\langle h \rangle$ in the `PIR-Trapdoor` procedure
`ctx`: ciphertext computed by the `PIR-Search` procedure
Output: The element $A[h]$ privately retrieved from the remotely stored array A with n entries
Data: sk: Secret key of Paillier FLAHE keypair
1 **begin**
2 **for** $i \leftarrow \lceil \log_b(n) \rceil$ **to** 1 **do**
3 `ctx` $\leftarrow \mathsf{D}^i_{sk}$(`ctx`)
4 **return** `ctx`

bits. The computational cost of the `PIR-Trapdoor` procedure amounts to $O(b \log^3(N) \log^4_b(n))$ 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 and reported in Alg. 3, takes as input the trapdoor $\langle h \rangle$, the value of the radix b employed by the client to construct $\langle h \rangle$ and the array A of items to be accessed, returning 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-rounds over the array A reported in the draft description of the PIR protocol, although in this case these rounds are no longer interactive between the client and the server, that is they are all performed consecutively at server side without sending back the results of each intermediate round to the client. In particular, in the first round (lines 3–4), 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., $\mathsf{D}_{sk}(\langle A_{h_0} \rangle[j]) = A[j \cdot b + h_0]$). To this end, each item $\langle A_{h_0} \rangle[j]$ is computed as the *hybrid dot product*, defined in Eq. (1), between the sub-array $A[j \cdot b, \ldots, j \cdot b + b - 1]$, whose entries are plaintexts in $\mathbb{Z}_N$, and the array $\mathtt{hdigit}_0$ found in the trapdoor $\langle h \rangle$, whose b ciphertexts are in $\mathbb{Z}^*_{N^2}$ (line 4).

In the subsequent round, which corresponds to the first iteration of loop at lines 5–7 in Alg. 3, 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*, defined in Eq. (2), 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 array $\mathtt{hdigit}_1$ found in the trapdoor, whose b ciphertexts are in $\mathbb{Z}^*_{N^3}$; 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, which encrypts the item $A[(j \cdot b + h_1) \cdot b + h_0)] = A[j \cdot b^2 + h_1 \cdot b + h_0]$ of the array A, 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 $\langle A_{h_1} \rangle[j]$: i.e., $A[j \cdot b^2 + h_1 \cdot b + h_0] = \mathsf{D}_{sk}(\mathsf{D}^2_{sk}(\langle A_{h_1} \rangle[j]))$. In general, in the i-th iteration of the loop at lines 5–7, $i \in \{1, \ldots, t-1\}$, the algorithm computes the array $\langle A_{h_i} \rangle$ with $\lceil \frac{n}{b^{i+1}} \rceil$ items, where the $\langle A_{h_i} \rangle[j]$ item, $0 \leq j \leq \lceil \frac{n}{b^{i+1}} \rceil - 1$, is computed as the *homomorphic dot product* (line 7) between the sub-array $\langle A_{h_{i-1}} \rangle[j \cdot b], \ldots, \langle A_{h_{i-1}} \rangle[j \cdot b + b - 1]$, whose entries are ciphertexts in $\mathbb{Z}^*_{N^{i+1}}$, and the array $\mathtt{hdigit}_i$ found in the trapdoor, whose b ciphertexts are in $\mathbb{Z}^*_{N^{i+2}}$; the item $\langle A_{h_i} \rangle[j]$ is an $i+1$-layered encryption of the entry $A[j \cdot b^{i+1} + \sum_{z=0}^{i} b^z \cdot h_z]$.

After $t-1$ iterations, $t = \lceil \log_b(n) \rceil$, the server computes a single *t-layered* ciphertext $\langle A_{h_{t-1}} \rangle$ that encrypts with t layers the item $A[0 \cdot b^t + \sum_{z=0}^{t-1} b^z \cdot h_z] = A[h]$, and then it sends $\langle A_{h_{t-1}} \rangle$ back to the client. 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 and reported in Alg. 4, employs the secret key sk to decrypt the ciphertext $\langle A_{h_{t-1}} \rangle$ computed by the `PIR-Search` procedure, obtaining the requested element $A[h]$. Since $\langle A_{h_{t-1}} \rangle$ is a *t-layered* ciphertext, then the client must remove all these t encryption layers by decrypting t times with decreasing length, as done in the loop at lines 2–3 of Alg. 4. The computational cost of the `PIR-Retrieve` amounts to $O(\log^5_b(n) \log^2(N))$ bit operations to derive the target value $A[h]$. Finally, 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 with $\texttt{len}(D_i)$, $1 \leq i \leq z$, characters 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 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

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

A privacy-preserving substring search (PPSS) protocol allows the server to provide the functionality specified in Def. 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 data owner, 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 Def. 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 reported in Alg. 1 and the Lipmaa's PIR protocol based on the FLAHE Paillier scheme, we now provide the operational description of the proposed PPSS protocol, reported in Alg. 5 and Alg. 6.

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. 5) 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 Alg. 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

Algorithm 5: `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 $s \leftarrow \mathsf{concat}(D_1, \$, D_2, \$, \ldots, D_z, \$)$
3 $n \leftarrow \sum_{i=1}^{z} \mathsf{len}(D_i) + 1$
/* 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 $base \leftarrow \mathtt{Order}(c) \cdot (n+1)$
6 **for** $j \leftarrow 1$ **to** $n+1$ **do**
7 $C[base+j] \leftarrow \mathsf{Rank}(c) + M[c][j]$
8 $(pk_{\mathcal{E}}, sk_{\mathcal{E}}) \leftarrow \mathcal{E}.\mathsf{KeyGen}(\lambda)$
9 **for** $i \leftarrow 1$ **to** $n+1$ **do**
10 $\langle SA \rangle[i] \leftarrow \mathcal{E}.\mathsf{Enc}(pk_{\mathcal{E}}, SA[i])$
11 **for** $i \leftarrow 1$ **to** $(n+1) \cdot (|\Sigma|+1)$ **do**
12 $\langle C \rangle[i] \leftarrow \mathcal{E}.\mathsf{Enc}(pk_{\mathcal{E}}, C[i])$
13 $aux_s \leftarrow (\mathtt{Order}, sk_{\mathcal{E}})$
14 [[**D**]] $\leftarrow (\langle C \rangle, \langle SA \rangle)$
15 **return** $(aux_s$, [[**D**]])

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

the character indexing a row of M. At lines 4–7, 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 7 of Alg. 5 to save the additions that should be executed later as per lines 5–6 of Alg. 1.

As the data structures C and SA are sufficient to reconstruct s, and thus the document collection **D**, they are cell-wise encrypted (lines 9–12) before being outsourced, obtaining arrays $\langle C \rangle$ and $\langle SA \rangle$. 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 8), where $pk_{\mathcal{E}} = sk_{\mathcal{E}}$ if $\mathcal{E}$ is a symmetric-key cipher.

At line 13, the secret information kept by the client aux_s is computed as the dictionary `Order` and the secret key of cipher $\mathcal{E}$. Finally, the `Setup` procedure in Alg. 5 returns the secret data to be kept by the client, aux_s=(`Order`, $sk_{\mathcal{E}}$), and the privacy-preserving representation $[[\mathbf{D}]]$, given by the pair of encrypted data structures ($\langle C \rangle$, $\langle SA \rangle$), of the indexing structure of the document collection, to be outsourced to the server.

The `Query` procedure, reported in Alg. 6, takes as input the m-character string to be searched q, the secret auxiliary information aux_s= (`Order`, $sk_{\mathcal{E}}$), 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 2–12), corresponds to lines 1–6 in Alg. 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 Alg. 1 are realized accessing the cells of the array $\langle C \rangle$. Realizing each access via the primitives of 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 PPSS protocol, the Lipmaa PIR protocol described 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 12 resp.).

The second phase, labeled as `Qocc` (lines 13–17), corresponds to lines 7–9 in Alg. 1, and it 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 Alg. 1 is realized by accessing privately the cells of the outsourced 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 encrypt 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 n 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 Alg. 6 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 $[[\mathbf{D}]]$ on the server requires $O(n)$ space. The cost of the `Query` procedure is split between the client and the server, obtaining $O((m + |R_q|) \cdot b \log^3(N) \log_b^4(n))$ and $O((m + |R_q|) \cdot \frac{n}{b} \log^3(N))$ complexities, respectively, where N is the modulus employed in the FLAHE Paillier keypair. The amount of data exchanged between the client and the server amounts to $O((m + |R_q|) \cdot \log(N) b \log_b^2(n))$.

We remark that any PIR protocol can be employed in our PPSS protocol, although the computational and communication costs depend on the PIR at hand. Therefore, improvements in terms of computation or communication in a PIR solution may be immediately applicable in our PPSS protocol. In particular, XPIR [29] and SealPIR [1] are two recently proposed PIR solutions that improve upon the scheme introduced by Stern in [37] showing significant improvements in terms of computation by relying on lattice-based additive homomorphic encryption schemes instead of number-theoretic ones such as the Paillier scheme employed in this work. Furthermore, XPIR and SealPIR rely on the *Learning With Error* trapdoor, which in turn provides post-quantum security assurances. Nonetheless, their communication cost amounts to $d \cdot n^{\frac{1}{d}} + F^d$, where $F = \Theta(1)$ and d is a value chosen by the client at each query. We observe that these protocols cannot achieve a polylogarithmic communication cost: indeed, if $d = O(1)$, then the communication cost is $O(n^{\frac{1}{O(1)}})$; if $d = \Theta(\log(n))$, then the communication cost becomes $O(\log(n) + F^{\log(n)}) = O(n)$. Since the main goal of this work is achieving a polylogarithmic communication cost, to make our protocol usable in scenarios with limited bandwidth (e.g., on mobile phones), we decide to employ

Table 2. Performance improvements enabled by batched retrieval of occurrences in our PPSS protocol

Performance Metric	Our PPSS Protocol	Our Enhanced Protocol with Batched Retrieval
Client Computation	$O((m+o_q)\cdot b\log^3(N)\log_b^4(n))$	$O(m\cdot b\log^3(N)\log_b^4(n)+o_q\cdot\log^2(N)\log_b^5(\frac{n}{o_q}))$
Server Computation	$O((m+o_q)\cdot\frac{n}{b}\log^3(N))$	$O((m+1)\cdot\frac{n}{b}\log^3(N))$
Communication	$O((m+o_q)\cdot\log(N)b\log_b^2(n))$	$O(m\cdot\log(N)b\log_b^2(n)+o_q\log(N)\log_b(\frac{n}{o_q}))$
Number of Rounds	$m+o_q$	$m+1$

the Lipmaa's PIR despite a possibly higher computational cost at server side. In application scenarios where a non polylogarithmic communication cost is acceptable, replacing the Lipmaa's PIR protocol with either XPIR or Seal PIR remains a viable solution.

Batched Retrieval of Occurrences. In the `Qocc` phase of the `Query` procedure (lines 13–17 in Alg. 6), the client performs $o_q = |R_q| = \beta - \alpha$ Lipmaa's PIR queries, one for each occurrence of the string q. We now introduce an optimization that allows to retrieve all the occurrences in a single communication round, reducing both computational and communication costs.

This optimization is based on enabling the retrieval of multiple entries (i.e., batches) of the outsourced dataset (array) managed by the Lipmaa's PIR protocol. To this extent, we introduce an additional parameter of the PIR protocol, denoted as a, that specifies the number of entries to be retrieved with a single PIR query; specifically, the dataset A with n elements is split in $\lceil\frac{n}{a}\rceil$ chunks of a consecutive entries each to allow the i-th chunk, $i\in\{0\ldots,\lceil\frac{n}{a}\rceil-1\}$, to be retrieved by the client issuing a single PIR query. In particular, the client generates a Lipmaa's PIR trapdoor to retrieve the i-th entry from a dataset with $\lceil\frac{n}{a}\rceil$ entries; the server splits the dataset A in a arrays, with the j-th one, $j\in\{0,\ldots,a-1\}$, containing all the entries $A[h]$ such that $h \bmod a = j$, and employs the trapdoor to retrieve the i-th entry from each of these a arrays; these a entries are then sent back to the client. The computational and communication costs for the `PIR-Trapdoor` procedure are reduced to $O(b\log^3(N)\log_b^4(\frac{n}{a}))$ and $O(b\log(N)\log_b^2(\frac{n}{a}))$, respectively, as a trapdoor for a dataset with $\lceil\frac{n}{a}\rceil$ entries is generated in place of a trapdoor for a dataset with n entries; conversely, both these costs increase by a factor a in the `PIR-Retrieve` procedure, because a entries are sent back from the server and then decrypted by the client. Most importantly, the computational cost at the server side is unchanged: indeed, the server performs a `PIR-Search` operations over a dataset with $\lceil\frac{n}{a}\rceil$ entries, thus yielding a $O(a\frac{n}{ab}\log(N)) = O(\frac{n}{b}\log(N))$ cost.

In the `Query` procedure of our PPSS protocol, the client can hinge upon this batched retrieval strategy in the `Qocc` phase to retrieve from the encrypted suffix array $\langle SA\rangle$ the $o_q = \beta - \alpha$ elements $\{\langle SA\rangle[\alpha+1],\ldots,\langle SA\rangle[\beta]\}$ with a single PIR query. Specifically, the client chooses a value $a \geq o_q$ as the minimum value such that $\lfloor\frac{\alpha+1}{a}\rfloor = \lfloor\frac{\beta}{a}\rfloor$: this choice guarantees that all the o_q elements $\{\langle SA\rangle[\alpha+1],\ldots,\langle SA\rangle[\beta]\}$ are found in the $\lfloor\frac{\alpha+1}{a}\rfloor$-th chunk among the $\lceil\frac{n}{a}\rceil$ ones of $\langle SA\rangle$. This strategy improves both the computational and communication costs of the queries in our PPSS protocol, as reported in Tab. 2. Remarkably, the computational cost at server side becomes independent from the number of occurrences o_q, as well as a significant improvement is observed in terms of communication cost because of the `Qocc` phase being performed in a single round instead of o_q ones.

4.1 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.

Algorithm 7: Optimized PIR-Search algorithm

Function PIR-Search(A, b, $\langle h \rangle$):
  Input: A, remote array with n entries
    $b \geq 2$, radix chosen by the client to construct $\langle h \rangle$;
    $\langle h \rangle$, trapdoor for the index h, given by t arrays $\texttt{hdigit}_0, \ldots, \texttt{hdigit}_{t-1}$, $t = \lceil \log_b(n) \rceil$, each with b ciphertexts (see Section 3.3).
  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 $\leftarrow \left\lfloor \frac{\text{end}-\text{begin}}{b} \right\rfloor$, acc $\leftarrow 1$
  for $i \leftarrow 1$ ***to*** b **do**
    el $\leftarrow$ RecursiveRet($\langle h \rangle$, A, $l-1$, begin, begin + size)
    begin $\leftarrow$ begin + size + 1
    acc $\leftarrow$ (acc $\cdot$ $\texttt{hdigit}_l[i]^{\text{el}}$) mod N^{l+1}
  return acc

In such a setting, each user is guaranteed to perform its own substring search queries 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 Alg. 5, computing the pair of arrays $[[\mathbf{D}]]=(\langle C \rangle, \langle SA \rangle)$ to be outsourced and sharing the secret auxiliary information $aux_s \leftarrow (\texttt{Order}, sk_{\mathcal{E}})$ with the authorized users. Each authorized user in turn can independently run a modified version of the Query procedure shown in Alg. 6 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's PIR-Search primitive with the one reported in Alg. 7, which aims to reduce the memory consumption of the Lipmaa's 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 $\langle A_{h_{i-1}} \rangle$ with $\lceil \frac{n}{b^i} \rceil$ elements. In particular, the first iteration computes an array $\langle A_{h_0} \rangle$ 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 naive PIR-Search procedure serializes the computation of the entire arrays $\langle A_{h_0} \rangle, \ldots, \langle A_{h_{t-1}} \rangle$. Nonetheless, it is possible to compute the element $\langle A_{h_1} \rangle[0]$ as soon as the b elements $\langle A_{h_0} \rangle[0], \ldots, \langle A_{h_0} \rangle[b-1]$ are computed, and, similarly, compute $\langle A_{h_1} \rangle[1]$ as soon as the b elements $\langle A_{h_0} \rangle[b], \ldots, \langle A_{h_0} \rangle[2b-1]$ are computed. Considering a generic element $\langle A_{h_i} \rangle[j]$, $1 \leq i \leq t-1$, $0 \leq j \leq \lceil \frac{n}{b^{i+1}} \rceil$, we can compute it as soon as the b elements $\langle A_{h_{i-1}} \rangle[b \cdot j], \ldots, \langle A_{h_{i-1}} \rangle[b \cdot j + b - 1]$ are available. We note that an element of $\langle A_{h_0} \rangle$ can always be computed as all the elements in the dataset A are available; therefore, to avoid that all elements of $\langle A_{h_0} \rangle$ are computed earlier than all the other elements of arrays $\langle A_{h_1} \rangle, \ldots, \langle A_{h_{t-1}} \rangle$, we rely on the following rule to schedule the operations: an entry of an array $\langle A_{h_i} \rangle$, $0 \leq i \leq t-1$, is computed only if there are no entries of arrays $\langle A_{h_j} \rangle$, $t-1 \geq j > i$, that are ready to be computed. This schedule of the operations is achieved by the recursive computation in Alg. 7.

The computational complexity of this algorithm is clearly equivalent to the naive iterative implementation, as the same operations are performed. Nevertheless, it exhibits a sublinear 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 stores $O(l \log(N))$ bits due to Paillier ciphertexts in $\mathbb{Z}^*_{N^{l+1}}$, thus 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.

Since each user employs for Lipmaa's PIR its own keypair for Paillier FLAHE scheme, then even if all the other users collude with the server to eavesdrop queries of the user at hand, they observe only ciphertexts of a semantically secure encryption scheme that can be decrypted only by the user issuing the substring search query. Therefore, users colluding with the server cannot learn any meaningful information about queries of non colluding users. This property is crucial to discourage collusion between the untrusted server and authorized users in our PPSS protocol: indeed, although the server is interested in such a collusion, as it would allow to learn the content of the document collection, the authorized users have no incentive to collude with the server.

4.2 Verifiability of retrieved data

In the following we enhance the design of the PPSS protocol presented in the previous section to provide a simple, yet effective, mechanism allowing clients to verify the correctness of the retrieved data with strong guarantees that they have not been accidentally tampered with by the storage service provider. In the considered semi-honest adversarial settings, the storage service provider is trustworthy to execute the steps of the PPSS protocol faithfully, even if it is interested to acquire as much information as possible on the stored data. However, chances of accidental or misconfiguration errors in the implementation or deployment of the protocol at server side make a mechanism to verify the correctness of accessed data elements a desirable feature.

We introduce a mechanism allowing the client to check whether the retrieved element matches the one prepared by the data owner or not. In particular, the retrieved entry may either be corrupted or corresponding to an entry different from the requested one. To this end, similarly to the approach followed in [20], we rely on a cryptographic (keyed) message authentication code (MAC) to detect that an outsourced data element a is corrupted (i.e., $\text{MAC}(k, a)$), while to prevent the chance that another legitimate entry is retrieved in place of the requested one, each entry of the array is associated with a unique MAC key, specific for the entry at hand. Specifically, the value of the secret cryptographic key employed to compute the MAC of each entry of the outsourced array must fulfill the following properties: *i)* it must depend on the index of the array entry; *ii)* it must be efficiently computable by the client given the index of an entry as an input.

In our design both properties are provided by generating the MAC key of an array entry via a keyed Pseudo Random Function (PRF) fed with a (secret) master key, shared by the data owner with each user, and the index of the entry at hand. A cryptographic PRF is an efficiently-computable function which emulates a random oracle. In particular, there is no efficient algorithm able to distinguish (with significant advantage) between the output of the chosen PRF and a the output of a random oracle, i.e., the outputs of a PRF are fixed completely at random.

The PPSS protocol presented in the previous section is therefore extended in the following way. The data owner selects a master secret key $\texttt{msk} \xleftarrow{\mathcal{R}} \{0,1\}^\lambda$, where λ is the security margin (e.g., $\lambda \in \{128, 192, 256\}$), and a keyed PRF $F : \{0,1\}^\lambda \times \{0,1\}^{\lceil \log(n) \rceil} \mapsto \{0,1\}^\lambda$, which takes as input a master key and an integer value denoting the index of an n-cell array. A possible instance of the said PRF is given by the AES-CBC encryption function yielding only the last block of the ciphertext and employing a λ-bit key, $\lambda \in \{128, 192, 256\}$. Subsequently, the value of each entry of the array A to be outsourced is augmented with its corresponding MAC, i.e., each $A[i] = a_i$, $i \in \{1, \ldots, n\}$, is replaced with $A[i] = (a_i, \text{MAC}(k_i, a_i))$, where $k_i = F(\texttt{msk}, i-1)$. When the client issues a query to retrieve the array element with index h, it gets back the pair of values $(a_{h'}, \text{MAC}(k_{h'}, a_{h'}))$ and can verify the correctness of the retrieved data by deriving $k_h = F(\texttt{msk}, h-1)$ and checking if the re-computation of $\text{MAC}(k_h, a_h)$ yields the same value $\text{MAC}(k_{h'}, a_{h'})$ returned by the server, thus concluding whether $h = h'$ or not. In case of an unsuccessful verification, the client can provide strong (cryptographic) evidence that the server either returned an accidentally corrupted element or wrongly returned an uncorrupted entry in place of the requested one. We remark that the communication and computational overheads due to the transmission of the MAC value and its

Table 3. Wildcard meta-characters available in the Unix glob patterns with their semantic

Wildcard	Semantic
$*$	Match zero or more arbitrary characters
?	Match exactly one character
$[abc]$	Match either a, b or c
$[0-9]$	Match a single digit
$[!ACG]$	Match any character except for A, C ad G
$[!A-Z]$	Match any character except for uppercase letters

re-computation at client-side, respectively, impact on the overall performance of the PPSS protocol in a negligible way. Indeed, they involve the transmission of a few tens of bytes and the computation of a symmetric-key cryptographic primitive, which is way more efficient (by two to three orders of magnitude) than the asymmetric cryptographic operations performed in the `PIR-Retrieve` procedure.

5 QUERIES WITH WILDCARD CHARACTERS

We now extend our PPSS protocol to enable queries for a string q containing meta-characters, also called wildcards, that allow to define a language (i.e., a set of strings) over the alphabet instead of a single string. We call a pattern, denoted by $\mathfrak{p}$ from here on, any string containing at least one of these wildcards; the language defined by a pattern $\mathfrak{p}$ is denoted by $L_{\mathfrak{p}}$. Although a pattern $\mathfrak{p}$ is usually employed to filter out strings that do not belong to $L_{\mathfrak{p}}$, in our PPSS protocol we want to find the positions of the (sub)strings in the document collection $\mathbf{D}$ that are also in $L_{\mathfrak{p}}$. From now on, these (sub)strings will be referred to as *matches* or *occurrences* of the pattern $\mathfrak{p}$ over the document collection; each occurrence is identified by the document where it is located and its starting position in the document at hand:

Definition 5.1 (Occurrence of a Pattern). Given a document collection $\mathbf{D}$ with $z \geq 1$ documents $D_1, \ldots, D_z$ and a pattern $\mathfrak{p}$, the set of positions of the occurrences of $\mathfrak{p}$ in D_i, $i \in \{1, \ldots, z\}$, is defined as:

$$O_{D_i,\mathfrak{p}} = \left\{ j \mid 1 \leq j \leq \mathtt{len}(D_i) \wedge \exists\, k \geq j \, (D_i[j, \ldots, k] \in L_{\mathfrak{p}}) \right\}$$

Coherently with Def. 4.1, our PPSS protocol aims at computing, for a pattern $\mathfrak{p}$ and a document collection $\mathbf{D}$, the set of positions $O_{\mathbf{D},\mathfrak{p}} = \bigcup_{i=1}^{z} O_{D_i,\mathfrak{p}}$. To specify a pattern $\mathfrak{p}$ in the queries of our PPSS protocol, we define our own format building upon the well-known *glob patterns*[1], which denote a simple syntax in the command line interface (CLI) of Unix-based systems that is largely used to filter out the filenames not belonging to the language defined by the specified pattern.

5.1 Format of Patterns in Queries

Table 3 reports the meta-characters defined in glob patterns and their semantic. Given the set of characters defined by an alphabet Σ, the wildcard $*$ is used to match any sequence of characters (even the empty one) over Σ, while the wildcard ? matches exactly one character in Σ. A *character class* denotes a syntax to match exactly one character belonging to a collection (class) of characters specified within square brackets. The collection of characters can be denoted by either listing all of them or specifying the first and the last of them (in lexicographic order) separated by a dash (e.g, $[a-z]$ denotes the class of lowercase latin letters). The meta-character ! can be employed only next to the left square bracket of a character class to denote the elements of Σ different from the ones belonging to the collection pointed out afterwards (e.g., $[!a-z]$ denotes all the characters in the alphabet except for lowercase latin ones).

[1] http://man7.org/linux/man-pages/man3/glob.3.html

To increase the expressiveness of a pattern, in our own format we employ the additional meta-character | to denote the *union operator*, i.e., given $k \geq 2$ strings $\beta_1, \ldots, \beta_k$, the pattern $\beta_1|\ldots|\beta_k$ matches any string among the k given ones. Together with the union operator we also introduce *round brackets* as meta-characters to specify unambiguously its scope, e.g., $a(a|b)c$. Besides querying strings containing wildcard characters, we also consider prefix (resp. suffix) queries to match strings positioned as prefix (resp. suffix) of each document in the collection $\mathbf{D}$, as customary in information retrieval contexts [14]. For instance, the widely used glob pattern $\mathfrak{p}_q = q*$ (resp. $\mathfrak{p}_q = *q$) can be used to issue a prefix (resp. suffix) query, requesting to match all documents starting (resp. ending) with a string equal to q. Since we define an occurrence of a pattern $\mathfrak{p}$ as the position in the document where a sequence of characters match (see Def. 5.1), we need to introduce a special symbol to specify that such a sequence must appear at the beginning or at the end of a document. To this extent, we add to our format the special symbol &, called *meta-delimiter*, which should appear only as the first or last character of a pattern.

The definition of the format of patterns in the expression of queries in our PPSS protocol is formally captured by Def. 5.2, Def. 5.3 and Def. 5.4, introduced to properly frame the use of the wildcard $*$.

Definition 5.2 (Star-Free Pattern). Given an alphabet Σ, the set $\mathcal{G}$ of glob wildcards reported in Tab. 3, the union operator | and the round brackets meta-characters, a pattern $\mathfrak{p}$ is *star-free* if and only if it is built as the concatenation of $k \geq 1$ strings $\mathfrak{p} = \alpha_1\alpha_2\cdots\alpha_k$, where each α_i belongs to one of the following types:

(1) $\alpha_i \in \Sigma^+$ (strings composed by at least one alphabet character)
(2) $\alpha_i \in \{\mathcal{G} \setminus \{*\}\}^+$ (strings composed by at least one meta-character in $\mathcal{G}$, except $*$)
(3) $\exists\, \beta_1, \cdots, \beta_h, h \geq 2 : \alpha_i = (\beta_1|\ldots|\beta_h) \vee \alpha_i = (\beta_1|\cdots|\beta_{h-1}|\varepsilon)$, where all $\beta_1, \cdots, \beta_h$ are star-free patterns. ε denotes here the empty string (i.e., a string with no characters) that is appended to a meta-character | to point out a possible match with no character in Σ.

Definition 5.3 (Well formed Star-Free Pattern). Given a star-free pattern $\mathfrak{p} = \alpha_1 \cdots \alpha_k$ over the alphabet Σ, it is a *well formed star-free* pattern if and only if there exists $1 \leq h \leq k$ such that $\alpha_h \in \Sigma^+$.

A query string is considered as a well formed star-free pattern if it contain at least one type (1) substring and does not contain the wildcard $*$. It is worth noting that such a query string may contain the union operator only if it is applied to star-free patterns (not necessarily well formed). We introduce such a restriction for simplicity and efficiency reasons, as the extended PPSS protocol enabling queries for strings with wildcard characters (reported in the next section) exhibit a computational complexity linear in the length of the longest string among the ones defined by the queried pattern. Indeed, the presence of a $*$ wildcard would easily increase the complexity to be linear in the size of the outsourced index.

Definition 5.4 (Well Formed Patterns). Given an alphabet Σ, the set $\mathcal{G}$ of glob wildcards reported in Tab. 3, the union operator |, the round brackets meta-characters, and the meta-delimiter symbol &, a pattern $\mathfrak{p}$ is a *well formed pattern* if and only if there exist $k \geq 1$ well formed star-free patterns $\alpha_1, \cdots, \alpha_k$ such that $\mathfrak{p}$ exhibits one of these structures:

(1) $p = \alpha_1 * \alpha_2 * \cdots * \alpha_k$
(2) $p = \&\alpha_1 * \alpha_2 * \cdots * \alpha_k$ (prefix pattern)
(3) $p = \alpha_1 * \alpha_2 * \cdots * \alpha_k\&$ (suffix pattern)
(4) $p = \&\alpha_1 * \alpha_2 * \cdots * \alpha_k\&$ (prefix-suffix pattern)

A well formed pattern has some restrictions on the usage of $*$ wildcard: indeed, the structure of a well formed pattern mandates that it cannot start or end with a $*$. We introduce this restriction since a pattern $\mathfrak{p}_\gamma = *\gamma*$, $\gamma \in \Sigma^+$, in our format would match any sequence of characters in the document collection having γ as a substring thus yielding too many (unuseful) occurrences (see Def. 5.1). Indeed, given a document $D_i \in \mathbf{D}$, $i \in \{1, \ldots, z\}$, $z \geq 1$, if j is the position of an occurrence of $\mathfrak{p}_\gamma$ in D_i, then every position $1 \leq h \leq j$ is also an occurrence of $\mathfrak{p}_\gamma$ in D_i.

5.2 PPSS Protocol for Pattern

In the following, we show how to extend our PPSS protocol to deal with queries asking for the occurrences of a well-formed pattern $\mathfrak{p}$. We proceed in two steps: we first show how to perform queries asking for the occurrences of a well-formed star-free pattern (see Def. 5.3) in Alg. 8; then, we show how to perform queries for a generic well-formed pattern (see Def. 5.4) in Alg. 9, relying on Alg. 8 as a building block.

Queries with Well-formed Star-free Patterns. Algorithm 8 shows how to perform queries for a well-formed star-free pattern hinging upon the following decomposition of the input pattern.

Lemma 5.5 (Decomposition of Well-Formed Star-Free Pattern). *Given a well-formed star-free pattern $\mathfrak{p}$ over an alphabet Σ, there exists a set of $2k+1$, $k \geq 1$, strings $\{\gamma_0, \omega_1, \gamma_1, \ldots, \omega_k, \gamma_k\}$ such that $\mathfrak{p} = \gamma_0\omega_1\gamma_1 \ldots \omega_k\gamma_k$, where:*

- *$\omega_1, \ldots, \omega_k \in \Sigma^+$ (i.e., type (1) strings in Def. 5.3)*
- *All strings $\gamma_0, \ldots, \gamma_k$ are a concatenation of type (2) or type (3) strings only (see Def. 5.3)*
- *γ_0 and γ_k may equal the empty string ε*

Proof. Following Def. 5.3, a well-formed star-free pattern $\mathfrak{p}$ is composed by $h \geq 1$ type (1), (2) or (3) strings, $\mathfrak{p} = \alpha_1 \cdots \alpha_h$, where at least one of them is a type (1) string. We prove the lemma by induction over the number h of type (1), (2) or (3) strings composing $\mathfrak{p}$. Assuming $h = 1$, there is only the pattern $\mathfrak{p} = \alpha_1$, where α_1 is a type (1) string: in this case, $\mathfrak{p}$ can be decomposed as $\gamma_0\alpha_1\gamma_1$, where $\gamma_0 = \gamma_1 = \varepsilon$, which satisfies the lemma. Assuming $h = 2$, $\mathfrak{p} = \alpha_1\alpha_2$, there are two possible cases: if both α_1 and α_2 are type (1) strings, then $\mathfrak{p}$ can be decomposed as $\gamma_0\omega_1\gamma_1$, where $\gamma_0 = \gamma_1 = \varepsilon$ and $\omega_1 = \alpha_1\alpha_2$, which satisfies the lemma; if α_1 (resp. α_2) is a type (1) string and α_2 (resp. α_1) is either a type (2) or a type (3) string, then $\mathfrak{p}$ can be decomposed as $\gamma_0\alpha_1\alpha_2$ (resp. $\alpha_1\alpha_2\gamma_1$), where $\gamma_0 = \varepsilon$ (resp. $\gamma_1 = \varepsilon$), which satisfies the lemma.

We now proceed with the inductive step. We want to prove that any well-formed star-free pattern $\mathfrak{p}$ composed by h+1 type (1), (2) or (3) strings $\mathfrak{p} = \alpha_1 \cdots \alpha_{h+1}$ can be decomposed as in the lemma. Consider $\mathfrak{p} = \alpha_1\mathfrak{p}'$, with $\mathfrak{p}' = \alpha_2 \cdots \alpha_{h+1}$. The pattern $\mathfrak{p}'$ is composed by h strings, thus, by inductive hypothesis, it can be decomposed as $\gamma'_0\omega'_1\gamma'_1 \cdots \omega'_k\gamma'_k$ for a $k \geq 1$. To encompass a generic pattern $\mathfrak{p}$ with h+1 strings, we have four different situations:

(1) α_1 is a type (1) string and $\gamma'_0 = \varepsilon$, as a consequence $\mathfrak{p}$ can be decomposed as $\gamma'_0\omega_1\gamma'_1 \cdots \omega'_k\gamma'_k$, with $\omega_1 = \alpha_1\omega'_1$
(2) α_1 is a type (1) string and $\gamma'_0 \neq \varepsilon$, as a consequence $\mathfrak{p}$ can be decomposed as $\gamma_0\alpha_1\gamma'_0\omega'_1\gamma'_1 \ldots \omega'_k\gamma'_k$, with $\gamma_0 = \varepsilon$
(3) α_1 is either a type (2) or type (3) string and $\gamma'_0 = \varepsilon$, as a consequence $\mathfrak{p}$ can be decomposed as $\alpha_1\omega'_1\gamma'_1 \ldots \omega'_k\gamma'_k$
(4) α_1 is either a type (2) or type (3) string and $\gamma'_0 \neq \varepsilon$, as a consequence $\mathfrak{p} = \gamma_0\omega'_1\gamma'_1 \ldots \omega'_k\gamma'_k$, with $\gamma_0 = \alpha_1\gamma'_0$

□

We recall that, in our PPSS protocol, an occurrence is identified by a pair (pos, off), where pos denotes the starting position of the document D_i in the string $s = D_1\$D_2\$ \ldots D_z\$$ (derived from the outsourced document collection $\mathbf{D} = \{D_1, \ldots, D_z\}$) where the occurrence is located, while off denotes the relative position of the occurrence in D_i. Along with this information, the `StarFreeQuery` procedure in Alg. 8 associates to each of the occurrences in the returned set $R_{\mathfrak{p}}$ also its ending position in the document D_i where it is located in order to ease the procedure in Alg. 9 concerning a query with a well-formed pattern. Analogously to the `Query` procedure reported in Alg. 6, the `StarFreeQuery` procedure in Alg. 8 takes as input a triple consisting of a well-formed star-free pattern $\mathfrak{p}$, the auxiliary secret information which is needed to generate PIR trapdoors and decrypt the results, and the privacy-preserving representation of the document collection $[[\mathbf{D}]]$. Nonetheless, $[[\mathbf{D}]]$ has to be enriched with the encrypted string $\langle s \rangle$ obtained applying the same semantically secure cipher $\mathcal{E}$ and key K employed for the original components of the privacy preserving representation of the outsourced document collection, i.e., $\langle C \rangle$ and $\langle SA \rangle$.

Algorithm 8: Query procedure for well-formed star-free patterns in our PPSS protocol

```
Function StarFreeQuery(p, aux_s, [[D]]):
    Input: p: well-formed star-free pattern to be searched
           aux_s = (Order, sk_E): secret auxiliary information employed by the client for the queries
           [[D]] = (⟨C⟩, ⟨SA⟩, ⟨s⟩): remotely accessed privacy-preserving representation of D
    Output: R_p: set of starting and ending positions of the occurrences of p in D
1   begin
2       (γ_0, ω_1, ..., ω_k, γ_k) ← ParsePattern(p)
3       if k = 1 ∧ γ_0 = ε ∧ γ_k = ε then
4           occ ← Query(ω_1), R_p ← ∅
5           foreach o ∈ occ do
                /* Query returns the pair (pos, off) denoting the starting position of the document where
                   the occurrence is located and its relative position from it */
6               R_p ← R_p ∪ (o.pos, o.off, o.off+len(ω_1)−1)
7           return R_p
8       α_min ← 0, β_min ← n
9       for i ← 1 to k do
10          (α, β) ← QueryNum(ω_i, aux_s, [[D]])
11          if β − α < β_min − α_min then
12              β_min ← β, α_min ← α, ω_min ← ω_i
13      Occ ← BatchedRetrieval(α_min+1, β_min, ⟨SA⟩, aux_s)
14      R_p ← ∅, len ← ComputeMaxLength(ω_min ... ω_k γ_k)
15      max_len ← ComputeMaxLength(p), min_len ← ComputeMinLength(p)
16      foreach o ∈ Occ do
17          end ← o.pos + o.off + len
18          start_1 ← end − max_len, start_2 ← end − min_len
19          str ← BatchedRetrieval(start_1, end, ⟨s⟩, aux_s)
20          for j ← start_1 to start_2 do
21              match_len ← MatchShortestPrefix(str[j − start_1 + 1, ..., max_len], p)
22              if match_len > 0 then
23                  offset ← j − 1 − o.pos
24                  R_p ← R_p ∪ (o.pos, offset, offset+match_len−1)
25      return R_p
```

The procedure follows the decomposition of $\mathfrak{p}$ defined in Lemma 5.5, parsing it properly (line 2). If the pattern is a type (1) string (line 3) it invokes the `Query` procedure in Alg. 6 as shown in line 4 of Alg. 8. After getting the occurrences of the pattern, the `StarFreeQuery` procedure enriches each of them with the corresponding ending position in the document where they are located. Such an ending position is computed considering the starting position and the length of the instance of the pattern at hand as shown in lines 5–7.

In case the pattern $\mathfrak{p}$ has some wildcards, the algorithm proceeds in two phases. In the first one (lines 8–13), the client considers the $k \geq 1$ type (1) strings $\omega_1, \ldots, \omega_k$ in $\mathfrak{p}$, with the aim of locating the one with the minimum number of occurrences in $\mathbf{D}$. To this extent, the algorithm executes, for each type (1) string ω_i, $i \in \{1, \ldots, k\}$, the same steps performed in the `Qnum` phase of the `Query` procedure in Alg. 6 (denoted as `QueryNum` in line 10), computing the indexes α, β identifying the portion $SA[\alpha{+}1, \ldots, \beta]$ of the suffix array that stores the $o_{\omega_i} = \beta - \alpha$ positions of occurrences of ω_i, later identifying the string ω_{min} with the least number of them (lines 11–12). Then, the batched retrieval method is employed to retrieve with a single PIR query the set of occurrences of ω_{min}, denoted as Occ in line 13.

The second phase (lines 14–25) uses the occurrences in Occ to finally compute the ones of the pattern $\mathfrak{p}$, returning them in the output set $R_{\mathfrak{p}}$. Indeed, each sequence of characters matching the pattern $\mathfrak{p}$ must contain ω_{min} as a substring, thus there will be an occurrence $o \in Occ$ having position $p_o = o.pos + o.off$ over s between the starting and ending positions of the occurrence of the pattern $\mathfrak{p}$ at hand. As a consequence, the occurrences of $\mathfrak{p}$ are computed by analyzing the characters of s preceding and succeeding each occurrence $o \in Occ$. To this end, Alg. 8 employs two procedures, `ComputeMaxLength` and `ComputeMinLength`, that, given a star-free pattern $\mathfrak{p}$, compute the lengths of the longest and shortest strings in $L_{\mathfrak{p}}$ (the set of all possible strings matching $\mathfrak{p}$), respectively. Indeed, since a star free pattern may contain a union operator applied to strings $\beta_1, \ldots, \beta_h$ with different length, the length of an occurrence of a pattern may vary depending on which string among $\beta_1, \ldots, \beta_h$ is matched in the occurrence. In the `StarFreeQuery` procedure, the algorithm first employs the `ComputeMaxLength` procedure to obtain the length, len, of the longest possible string matching the pattern $\mathfrak{p}$ shortened to start from ω_{min}(line 14); then, it computes the lengths of the longest and shortest strings matching the whole $\mathfrak{p}$ as max_len and min_len at line 15.

For each occurrence $o \in Occ$, the biggest possible position of the final character of a substring matching $\mathfrak{p}$ and including $s[p_o]$, $p_o = o.pos+o.off$, is computed as $end = p_o + len$ (line 17). A range $start_1, \cdots, start_2$ for the possible position of the first character of a substring matching $\mathfrak{p}$ and including $s[p_o]$, $p_o = o.pos+o.off$, is evaluated by computing $start_1 = end - max\_len$ and $start_2 = end - min\_len$ (line 18). Subsequently, the sequence of characters $str = s[start_1, \cdots, end]$ is obliviously fetched via the enhanced version of the Lipmaa's PIR protocol employing a batched data retrieval (line 19). The retrieved sequence of characters str includes all the occurrences of $\mathfrak{p}$ containing the character $s[p_o]$ and with initial character positioned over str between 1 and $start_2 - start_1 + 1$. Thus, the occurrences of $\mathfrak{p}$ over str are evaluated searching for the shortest prefix match between $str[j - start_1 + 1 \cdots max\_len]$ and the possible strings matching $\mathfrak{p}$ (i.e., the ones in $L_{\mathfrak{p}}$), with $start_1 \leq j \leq start_2$. If the said shorted prefix match exists, the starting and ending positions of the occurrence $\mathfrak{p}$ are inserted in the output set $R_{\mathfrak{p}}$ together with the starting position of the document in s where the occurrence at hand is located, i.e., $o.pos$, (lines 21–24).

We now analyze the computational and communication costs of the `StarFreeQuery` algorithm fed with a pattern $\mathfrak{p}$ that is analyzed by decomposing it as $\mathfrak{p} = \gamma_0\omega_1\gamma_1 \ldots \omega_k\gamma_k$, $k \geq 1$. Let us denote as o_{ω_i} the number of occurrences of ω_i in the document collection $\mathbf{D}$, as m the length of the longest possible string matching $\mathfrak{p}$, i.e., m=`ComputeMaxLength`($\mathfrak{p}$), and as m_ω=$\sum_{i=1}^{k}$ `len`(ω_i) the sum of the lengths of type (1) strings $\omega_1, \ldots, \omega_k$.

Concerning the communication cost, in the first phase of the algorithm (lines 2–13) the client sends $O(m_\omega b \log_b^2(n) \log(N))$ bits to the server (with N and b being parameters employed in the Lipmaa's PIR protocol as the cryptographic modulus of the Pailler's scheme and the radix value to compute trapdoors, respectively), while the server sends back $O((m_\omega+o_{min}) \cdot \log_b(n) \log(N))$ bits, where $o_{min} = \mathtt{min}(o_{\omega_1}, \ldots, o_{\omega_k})$; this cost is largely dominated by the communication cost of the second phase (lines 14–25), which amounts to $O(o_{min} \cdot (b \log_b^2(n) \log(N) + m \log_b(n) \log(N)))$, that is the cost of o_{min} PIR queries, each of which retrieving $O(m)$ characters from $\langle s \rangle$.

The computational cost at client side, which amounts to $O(o_{min} \cdot (b \log_b^4(n) \log^3(N) + m \log_b^5(n) \log^2(N)))$, is due to the o_{min} queries that retrieve $O(m)$ characters from $\langle s \rangle$; indeed, the computational cost to match the pattern $\mathfrak{p}$ with the string downloaded from the server is negligible with respect to the cryptographic operations. Lastly, the computational cost at server side amounts to $O((m_\omega+o_{min}) \cdot \frac{n}{b} \log^3(N))$, which is obtained by adding the cost of executing the `QueryNum` procedure k times and the cost of retrieving o_{min} substrings from $\langle s \rangle$.

Dealing with Meta-Delimiters. To employ the `StarFreeQuery` algorithm as a building block during the execution of queries with well-formed patterns (Def. 5.4), it must be extended to manage also the meta-delimiters preceeding or succeeding a well-formed star-free pattern $\mathfrak{p}$. To this extent, the `ParsePattern` procedure must

convert each input meta-delimiter character & into the end-of-string delimiter \$ employed to concatenate the documents in the collection **D** into the string s. Two cases are possible:

- if the replaced \$ preceedes (resp. succeedes to) the type (1) string ω_1 (resp. ω_k), then it is merged with the string at hand and the `StarFreeQuery` procedure proceeds as shown in Alg. 8. Indeed, the type (1) string enriched with the merged symbol \$ may be processed by either the `Query` or the `QueryNum` procedure. Both procedures support prefix, suffix and prefix-suffix queries without any further modifications. Indeed, the pattern $\$\omega_1$ (resp. $\omega_k\$$) can occur only at the beginning (resp. end) of a document in s, while the pattern $\$\omega_1\$$ can match a document only if the entire document is equal to ω_1.
- If the replaced \$ neither preceeds nor succeeds a type (1) string, then the algorithm exploits the fact that the number of occurrences of the pattern with the meta-delimiters is at most z, one for each document D_i in $s = D_1\$\cdots D_z\$$. Therefore, if, at the end of the loop at lines 9–12, the string ω_{min} has more than z occurrences, then the `StarFreeQuery`, instead of retrieving its occurrences at line 13, retrieves the occurrences of \$ in s. Specifically, these are stored in the first z entries of the suffix array and so they can be retrieved at line 13 by feeding `BatchedRetrieval` procedure with $\alpha_{min} = 0, \beta_{min} = z$. Obviously, since now the occurrences in Occ no longer refers to the string ω_{min} but to the symbol \$, the computation of the variable len at line 14 must be modified accordingly: that is, if the meta-delimiter preceeds $\mathfrak{p}$, then len equals `ComputeMaxLength`($\mathfrak{p}$) plus the number of meta-delimiters, otherwise $len = 0$.

Queries with Well-formed Patterns. The procedure `QueryPattern` in Alg. 9 extends our PPSS protocol allowing to locate all the occurrences of a well-formed pattern (see Def. 5.4). The algorithm hinges upon the decomposition of a well-formed pattern $\mathfrak{p}$ in a set of $k \geq 1$ well-formed star-free patterns $\alpha_1, \ldots, \alpha_k$ (see Def. 5.4). Specifically, given all the occurrences of each of these k patterns in the document collection $\mathbf{D} = \{D_1, \ldots, D_z\}$, it is possible to construct an occurrence of $\mathfrak{p}$ over a document D_j in **D**, $1 \leq j \leq z$, by finding a set of occurrences $\mathfrak{o}_{k,j} = \{o_{1,j}, \ldots, o_{k,j}\}$ located in D_j such that $o_{i,j}$ is an occurrence of α_i, $i \in \{1, \ldots, k\}$, and $\forall\, i \leq k-1$, $o_{i,j}.end < o_{i+1,j}.begin$, where $o_{i,j}.end$ (resp. $o_{i,j}.begin$) denotes the ending (resp. starting) position of the occurrence $o_{i,j}$. Indeed, $\mathfrak{o}_{k,j}$ identifies an occurrence of $\mathfrak{p}$ in document D_j starting at position $o_{1,j}.begin$ and ending at $o_{k,j}.end$, which is composed by occurrences of well-formed star-free patterns $\alpha_1, \ldots, \alpha_k$ interleaved by an arbitrary number of characters (due to presence of the wildcard $*$ in $\mathfrak{p}$).

Algorithm 9 starts by decomposing the well-formed pattern $\mathfrak{p}$ in $k \geq 1$ well-formed star-free patterns $\alpha_1, \ldots, \alpha_k$ (line 2). Any meta-delimiter & found at the beginning (resp. at the end) of $\mathfrak{p}$ is merged with the pattern α_1 (resp. α_k). Then, for each of these k patterns, the `StarFreeQuery` procedure is run to retrieve the set Occ_i of occurrences of the pattern α_i in the document collection (line 4); each set Occ_i is further partitioned in z portions $(Occ_i^1, \ldots, Occ_i^z)$, each containing all the occurrences located in j-th document D_j, $j \in \{1, \ldots, z\}$ (line 5). This partitioning is simply performed by grouping the occurrences in Occ_i, $i \in \{1, \ldots, k\}$, according to the position pos of the document where each occurrence is located.

Subsequently, for each document D_j, the algorithm (lines 6–9) constructs occurrences of the well-formed pattern $\mathfrak{p}$ from the sets $Occ_1^j, \ldots, Occ_k^j$ corresponding to the well-formed star-free patterns $\alpha_1, \ldots, \alpha_k$. The occurrences of $\mathfrak{p}$ are found by the procedure `MatchOcc`, which, for each $o_{1,j}^h \in Occ_1^j$, $h \in \{1, \ldots, |Occ_1^j|\}$, computes a set of occurrences $\mathfrak{o}_{k,j}^h$ corresponding to an occurrence of $\mathfrak{p}$; although many sets $\mathfrak{o}_{k,j}^h$ may exists for each $o_{1,j}^h$, according to Def. 5.1 they represent the same occurrence of the pattern $\mathfrak{p}$, as they share the same starting position $o_{1,j}.begin$. Therefore, there are at most $|Occ_1^j|$ occurrences of $\mathfrak{p}$ in the document D_j, one for each $o_{1,j}^h$. The `MatchOcc` procedure can naively construct all these $|Occ_1^j|$ sets $\mathfrak{o}_{k,j}^h$ in time $O(|Occ_1^j| \cdot o_{\alpha,j})$, where $o_{\alpha,j} = \sum_{i=1}^{k} |Occ_i^j|$, that is the sum of the occurrences of each pattern α_i over document D_j; we remark that as $|Occ_1^j| = O(o_{\alpha,j})$, then the cost of this naive implementation becomes $O(o_{\alpha,j}^2)$.

Algorithm 9: Query procedure for well-formed patterns in our PPSS protocol

Function QueryPattern($\mathfrak{p}$, aux_s, [[D]]):
Input: $\mathfrak{p}$: well-formed pattern to be searched
$aux_s = (\texttt{Order}, sk_{\mathcal{E}})$: secret auxiliary information employed by the client for queries
[[D]] = $(\langle C\rangle, \langle SA\rangle, \langle s\rangle)$: remotely accessed privacy-preserving representation of **D**
Output: $R_{\mathfrak{p}}$: set of positions of occurrences of $\mathfrak{p}$ in **D**

1 **begin**
2 $(\alpha_1, \ldots, \alpha_k) \leftarrow$ ParsePattern($\mathfrak{p}$)
3 **for** $i \leftarrow 1$ **to** k **do**
4 $Occ_i \leftarrow$ StarFreeQuery(α_i, aux_s, [[D]])
5 $(Occ_i^1, \ldots, Occ_i^z) \leftarrow$ SplitByDocID(Occ_i)
6 $R_{\mathfrak{p}} \leftarrow \emptyset$
7 **for** $j \leftarrow 1$ **to** z **do**
8 $R_{\mathfrak{p}} \leftarrow R_{\mathfrak{p}} \cup$ MatchOcc($Occ_1^j, \ldots, Occ_k^j$)
9 **return** $R_{\mathfrak{p}}$

Algorithm 10: Optimized MatchOcc procedure to find occurrences of well-formed patterns

Function MatchOcc $(Occ_1, \ldots, Occ_k)$:
Input: $Occ_1, \ldots, Occ_k$: set of occurrences of well-formed star free patterns $\alpha_1, \ldots, \alpha_k$ over the same document. Each occurrence o in Occ_i stores the position $o.pos$ of the document where the occurrence is located in the string s and the starting and ending positions ($o.begin$ and $o.end$) of the occurrence in the document
Output: $R_{\mathfrak{p}}$: set of occurrences of the well-formed pattern $p = \alpha_1 * \cdots * \alpha_k$ over the same document

1 **begin**
2 **for** $i \leftarrow 1$ **to** k **do**
3 SortOccByEnd(Occ_i)
4 **foreach** $o_1 \in Occ_1$ **do**
5 $pos \leftarrow o_1.end$
6 **for** $i \leftarrow 2$ **to** k **do**
7 **foreach** $o \in Occ_i$ **do**
8 **if** $o.begin > pos$ **then**
9 $pos \leftarrow o.end$, **break**
10 DeleteOcc(o)
11 **if** $Occ_i = \emptyset$ **then**
12 **return** $R_{\mathfrak{p}}$
13 $R_{\mathfrak{p}} = R_{\mathfrak{p}} \cup (o_1.pos, o_1.begin)$
14 **return** $R_{\mathfrak{p}}$

In Alg. 10, we report an improved version of MatchOcc that reduces the computational cost to $O\left(o_{\alpha,j} \cdot (\log(o_{\alpha,j}) + k)\right)$. This procedure relies on the fact that it is possible to build the occurrences of pattern $\mathfrak{p}$ in the document D_j much more efficiently if each of the k sets $Occ_1^j, \ldots, Occ_k^j$ is sorted in ascending order according to the ending positions of the occurrences in it. We now describe how the MatchOcc procedure efficiently finds all the occurrences of $\mathfrak{p}$ over document D_j from the k sorted sets $Occ_1^j, \ldots Occ_k^j$; then, we prove its correctness. After sorting all the sets $Occ_1^j, \ldots Occ_k^j$ (line 3), the MatchOcc procedure tries to build a set $\mathfrak{o}_{k,j}^h$ for any occurrence $o_{1,j}^h \in Occ_1^j$ (lines 4-13). Specifically, for each set Occ_i^j, $i \in \{2, \ldots, k\}$, it finds (lines 7-10) the first occurrence satisfying $o_{i,j}.begin > o_{i-1,j}.end$ ($o_{i-1,j}.end$ is stored in variable *pos* in Alg. 10). Any occurrence $o_{i,j}$ such that $o_{i,j}.begin \leq pos$ is erased from set Occ_i^j (line 10); If no occurrence $o_{i,j}$ with $o_{i,j}.begin > pos$ can be found (line 11), no more occurrences of $\mathfrak{p}$ in document D_j can be found (line 12). Conversely, in case an occurrence $o_{i,j}$ is found for every set Occ_i^j, $i \in \{2, \ldots, k\}$, then the set $\mathfrak{o}_{k,j}^h$ can be built and thus the occurrence identified by this set, which starts in position $o_{1,j}^h.begin$, is added to $R_{\mathfrak{p}}^j$ (line 13), that is the set of occurrences of the well-formed pattern $\mathfrak{p}$ over the document D_j.

We now prove that MatchOcc allows to find all and only the occurrences of $\mathfrak{p}$ in a document D_j. The set $R_{\mathfrak{p}}^j$ computed by the MatchOcc procedure contains only occurrences of $\mathfrak{p}$ over document D_j: indeed, an entry is added to this set if and only if a set $\mathfrak{o}_{k,j}^h = \{o_{1,j}, \ldots, o_{k,j}\}$ of occurrences of $\alpha_1, \ldots, \alpha_k$, with $o_{1,j} = o_{1,j}^h$ and $o_{i,j}.end < o_{i+1,j}.begin$, for every $i < k$, is found. We now prove that the procedure finds all the occurrences. In particular, we want to show these two facts: an occurrence $o_{i,j}$ is erased from set Occ_i^j (line 10) only if $o_{i,j}$

cannot belong to any other set $\mathfrak{o}^h_{k,j}$ except for the ones already found; if, for any of the sets $Occ^j_2, \ldots, Occ^j_k$, all occurrences in the set at hand are erased (line 12), then there are no more sets $\mathfrak{o}^h_{k,j}$ (and thus no more occurrences of $\mathfrak{p}$) to be found. We start by proving the following property:

Lemma 5.6. *Consider the $k \geq 1$ sets of occurrences $Occ^j_1, \ldots, Occ^j_k$ of well-formed star-free patterns $\alpha_1, \ldots, \alpha_k$ over document D_j, with the set Occ^j_i, $i \in \{1, \ldots, k\}$, being sorted according to the ending position of each of its occurrences. If, for an occurrence $o_{i,j} \in occ^j_i$, $i \in \{2, \ldots, k\}$, there is no set $\mathfrak{o}^h_{i,j} = \{o_{1,j}, \ldots, o_{i,j}\}$, with $o_{1,j} = o^h_{1,j}$, such that $\forall i' < i, o_{i',j}.end < o_{i'+1,j}.begin$, then, for any $h' \geq h$, there is no set $\mathfrak{o}^{h'}_{i,j} = \{o_{1,j}, \ldots, o_{i,j}\}$, with $o_{1,j} = o^{h'}_{1,j}$, such that $\forall i' < i, o_{i',j}.end < o_{i'+1,j}.begin$*

Proof. Assume that the thesis of the Lemma is false. This implies that, even if there is no set $\mathfrak{o}^h_{i,j} = \{o_{1,j}, \ldots, o_{i,j}\}$, with $o_{1,j} = o^h_{1,j}$, for an occurrence $o_{i,j} \in occ^j_i$, there exists a set $\mathfrak{o}^{h'}_{i,j} = \{o_{1,j}, \ldots, o_{i,j}\}$, with $o_{1,j} = o^{h'}_{1,j}$, for any $h' \geq h$. Given the occurrence $o_{2,j}$ in $\mathfrak{o}^{h'}_{i,j}$, it holds that $o_{2,j}.begin > o^{h'}_{1,j}.end \geq o^h_{1,j}.end$, since the set Occ^j_1 is sorted according to the ending position of its occurrences; therefore, by replacing $o^{h'}_{1,j}$ with $o^h_{1,j}$ in $\mathfrak{o}^{h'}_{i,j}$, we obtain a set $\mathfrak{o}^h_{i,j} = \{o_{1,j}, \ldots, o_{i,j}\}$, with $o_{1,j} = o^h_{1,j}$, such that $\forall i' < i, o_{i',j}.end < o_{i'+1,j}.begin$. This contradicts the hypothesis that the set $\mathfrak{o}^h_{i,j}$ does not exist, so we conclude that there is no set $\mathfrak{o}^{h'}_{i,j}$ for the occurrence $o_{i,j} \in Occ^j_i$ for any $h' \geq h$. □

Lemma 5.7. *If an occurrence $o_{i,j} \in Occ^j_i$ is erased at line 10 of* `MatchOcc` *procedure in the h-th iteration of the loop at lines 4-13, then, for any $h' \geq h$, there is no set $\mathfrak{o}^{h'}_{i,j} = \{o_{1,j}, \ldots, o_{i,j}\}$, with $o_{1,j} = o^{h'}_{1,j}$, such that $\forall i' < i, o_{i',j}.end < o_{i'+1,j}.begin$.*

Proof. We prove the lemma by induction over the sets $Occ^j_2, \ldots, Occ^j_k$. We start with Occ^j_2. Suppose that an occurrence $o_{2,j} \in Occ^j_2$ is erased by `MatchOcc` procedure in the h-iteration of the loop at lines 4-13: then, it means that $o_{2,j}.begin \leq pos$, where $pos = o^h_{1,j}.end$; thus, it immediately follows that $\mathfrak{o}^h_{2,j} = \{o^h_{1,j}, \ldots, o_{2,j}\}$ cannot exist. Therefore, by Lemma 5.6, for any $h' \geq h$, no set $\mathfrak{o}^{h'}_{2,j} = \{o^{h'}_{1,j}, \ldots, o_{2,j}\}$ exists. We now look at the general case for any occurrence $o_{i,j}$ in the set Occ^j_i erased in the h-th iteration of loop at lines 4-13. Our inductive hypothesis is that for any element $o_{i-1,j} \in Occ^j_{i-1}$ already erased, there is no set $\mathfrak{o}^h_{i-1,j} = \{o^h_{1,j}, \ldots, o_{i-1,j}\}$. If occurrence $o_{i,j}$ is erased, then it means that there is an occurrence $o'_{i-1,j} \in Occ_{i-1}$ such that $o_{i,j}.begin \leq o'_{i-1,j}.end$; therefore, no set $\mathfrak{o}^h_{i,j} = \{o^h_{1,j}, \ldots, o'_{i-1,j}, o_{i,j}\}$ exists. If we consider any $o_{i-1,j}$ already erased from Occ_{i-1}, then by inductive hypothesis there is no set $\mathfrak{o}^h_{i-1,j} = \{o^h_{1,j}, \ldots, o_{i-1,j}\}$, which implies that there is also no set $\mathfrak{o}^h_{i,j} = \{o^h_{1,j}, \ldots, o_{i-1,j}, o_{i,j}\}$. If we consider any $o_{i-1,j} \neq o'_{i-1,j}$ still in Occ^j_{i-1}, then $o_{i-1,j}.end \geq o'_{i-1,j}.end$, as the set Occ^j_{i-1} is sorted in ascending order according to the ending position of its occurrences; therefore, $o_{i,j}.begin \leq o'_{i-1,j}.end \leq o_{i-1,j}.end$, which means that no set $\mathfrak{o}^h_{i,j} = \{o^h_{1,j}, \ldots, o_{i-1,j}, o_{i,j}\}$ exists. In conclusion, there is no element in $o_{i-1,j} \in Occ_{i-1}$ such that it is possible to construct a set $\mathfrak{o}^h_{i,j} = \{o^h_{1,j}, \ldots, o_{i-1,j}, o_{i,j}\}$, therefore, this set does not exist for the occurrence $o_{i,j}$. By applying Lemma 5.6, we can generalize this result to any $h' > h$. □

Lemma 5.7 implies that when an occurrence $o_{i,j}$ is erased in the h-th iteration of the loop at lines 4-13, then $o_{i,j}$ cannot belong to any set $\mathfrak{o}^{h'}_{k,j}$ for any $h' \geq h$. As an occurrence of $\mathfrak{p}$ over document D_j is identified by a set $\mathfrak{o}^h_{k,j} = \{o_{1,j}, \ldots, o_{k,j}\}$, with $o_{1,j} = o^h_{1,j}$, of k occurrences from $Occ^j_1, \ldots, Occ^j_k$, this means that occurrence $o_{i,j}$ cannot belong to any occurrence of $\mathfrak{p}$ identified by a set $\mathfrak{o}^{h'}_{k,j}$ for any $h' \geq h$. Since all the occurrences of $\mathfrak{p}$ identified by sets $\mathfrak{o}^{h'}_{k,j}$, for $h' < h$, have been already found at iteration h of the aforementioned loop, then the

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

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

Fig. 2. Security game experiments

occurrence $o_{i,j}$ cannot belong to any other occurrence of $\mathfrak{p}$ except for the ones already found. Therefore, $o_{i,j}$ can be safely erased from its set.

Furthermore, if all occurrences are erased in the h-th iteration of the loop at lines 4-13 from a set Occ_i^j, $i \in \{2, \ldots, k\}$, then all these occurrences by Lemma 5.7 cannot belong to any set $\mathfrak{o}_{k,j}^{h'}$ for any $h' \geq h$. Since, by definition, a set $\mathfrak{o}_{k,j}^{h} = \{o_{1,j}, \ldots, o_{k,j}\}$, with $o_{1,j} = o_{1,j}^{h}$, identifying an occurrence of $\mathfrak{p}$ over the document D_j is composed by k occurrences, one for each set Occ_i^j, then any set $\mathfrak{o}_{k,j}^{h'}$, for $h' \geq h$, cannot be built. Since all the sets $\mathfrak{o}_{k,j}^{h'}$, for $h' < h$, have been already built at the h-th iteration, there are no more occurrences of $\mathfrak{p}$ to be found, which means that the `MatchOcc` procedure can immediately stop at line 12 returning the set $R_{\mathfrak{p}}^j$ of occurrences of $\mathfrak{p}$ in document D_j found so far.

We conclude this section by analyzing the computational and communication costs of the `QueryPattern` procedure in Alg. 9, which mostly amount to the costs of the k queries for the well formed star-free patterns $\alpha_1, \ldots, \alpha_k$. In case these patterns contain wildcard characters, the computational cost at server side is linear in the length of the pattern m and in the number $o_\alpha = \sum_{i=1}^{k} o_{\alpha_i}$ of occurrences of the patterns, while the communication cost is linear in $\sum_{i=1}^{k} m_{\alpha_i} \cdot o_{\alpha_i}$, that is the sum of the products between the lengths of each pattern and the numbers of its occurrences; remarkably, in case all these patterns do not include wildcard characters, then the computational cost at server side remains independent from the number of occurrences, while the communication cost is only linear in m and $o_{\alpha,j}$.

6 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 [9], 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.

Definition 6.1 (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 Def. 6.2 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 6.2 (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 $\mathcal{D}$:

$\mathtt{Pr}\Big(\mathcal{D}(o)=1 | o \leftarrow \mathbf{Real}_{\mathcal{P},\mathcal{A}}(\lambda)\Big) - \mathtt{Pr}\Big(\mathcal{D}(o)=1 | o \leftarrow \mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)\Big) \leq \epsilon(\lambda)$, 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 $\mathcal{L}_D$, $\mathcal{L}_{q_j}$, $j \in \{1, \ldots, i\}$.

Theorem 6.3. *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 Def. 6.2, 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 Def. 4.1 and b_i is the radix chosen to execute the Lipmaa PIR protocol.*

Proof. See Appendix A. □

We remark that Theorem 6.3 guarantees search and access pattern privacy, as they are not enclosed in the leakage $\mathcal{L}$.

Theorem 6.3 applies also to the enhanced version of our PPSS protocol with the batched retrieval of occurrences: indeed, it is possible to build a simulator $\mathcal{S}'$, simulating this enhanced version in the $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}'}$ experiment of Def. 6.2, with a simple modification of the simulator $\mathcal{S}$ constructed in the proof reported in Appendix A. The leakage $\mathcal{L}$ does not change in the enhanced version of the protocol: indeed, the number of elements sent back to the client is close to the actual number of occurrences.

Privacy Guarantees of Queries with Wildcards. To analyze the information leakage of queries containing wildcard characters, we assume the version of our PPSS protocol enhanced with batched retrieval procedure and we distinguish two possible use case scenarios: in the first one, which is more unlikely, the adversary knows that the client is performing a single query (e.g., an application scenario where each user is allowed to perform a single query per day); in the second one, the client may perform an arbitrary number of queries.

In the first scenario, the adversary can infer the number k of $*$ wildcards in the queried pattern $\mathfrak{p} = \alpha_1 * \cdots * \alpha_{k+1}$ and, for each of the k+1 well-formed star-free patterns $\alpha_1, \ldots, \alpha_{k+1}$ if there is at least a wildcard different from $*$ and &. The value k can be inferred by the number of private accesses to the encrypted array $\langle SA \rangle$ performed during the execution of the query: indeed the `QueryPattern` procedure (line 4 in Alg. 9) runs k+1 times the `StarFreeQuery` function in Alg. 8, which in turn accesses the array $\langle SA \rangle$ only once per run either by performing a batched retrieval when the `Query` function in Alg. 6 is run (line 4 in Alg. 8) or by executing the `BatchedRetrieval` function at line 13 in Alg. 8. Furthermore, if the client performs a private access to the encrypted array $\langle s \rangle$ (line 19 in Alg. 9) after the i-th access to $\langle SA \rangle$, $1 \leq i \leq k+1$, then the adversary learns that the i-th well-formed star-free pattern α_i found in $\mathfrak{p}$ contains a wildcard character different from &. Once the adversary has reconstructed the structure of $\mathfrak{p}$, for each well-formed star-free pattern with no wildcard character (except for &) she learns its length and its number of occurrences; otherwise, for each well-formed star-free pattern α_i with at least a wildcard character, she learns the length of the longest string in L_{α_i} (from the number of elements retrieved at line 19 in Alg. 8) and the upper bound o_{min} (see communication cost of `StarFreeQuery` procedure in Section 5) on the number of its occurrences (from the number of elements retrieved at line 13 in Alg. 8).
Therefore, the adversary cannot learn the actual number of occurrences of a pattern unless it is a well-formed star-free pattern with no wildcards other than &.

In the second scenario, where the client may perform an arbitrary number of queries, the adversary can no longer reconstruct the number of $*$ wildcards: indeed, the adversary only observes a set of queries for well-formed star-free patterns, but it cannot determine which of them are portions of the same well-formed pattern. Nonetheless, the adversary can still infer, for each of the observed queries, if the queried well-formed star-free pattern $\mathfrak{p}$ contains at least a wildcard character other than $*$ or & by verifying if the client retrieves any element from the encrypted array $\langle s \rangle$. It is worth noting that this information leakage may be not accurate for the adversary: indeed, although not specified in our PPSS protocol, in some application scenarios the client, once determined the positions of an occurrence in a document, may need to download the portions of the document corresponding to such an occurrence, hereby privately accessing via PIR queries the encrypted array $\langle s \rangle$ too. Similarly to the previous scenario, in case the adversary determines that the queried well-formed star-free pattern $\mathfrak{p}$ contains at least a wildcard character other than $*$ or &, she learns the length of the longest string in L_{α_i} and the upper bound o_{min} on the number of its occurrences; otherwise, she learns the length of $\mathfrak{p}$ and the number of its occurrences. Nonetheless, since in this scenario the client cannot know if $\mathfrak{p}$ is a portion of a bigger well-formed pattern or not, the adversary can never know with certainty both the length and the number of occurrences of the well-formed pattern.

The leakage of the structure of the well-formed pattern in both scenarios can be prevented with the following two modifications: the first one consists of conceiving the encrypted arrays $\langle C \rangle$, $\langle SA \rangle$ and $\langle s \rangle$ in $[[\mathbf{D}]]$ as a single dataset, thus any PIR query will be performed over all these three arrays; the second modification requires that PIR queries always retrieve elements in batches of constant size (otherwise the information concerning which array among $\langle C \rangle$, $\langle SA \rangle$ and $\langle s \rangle$ has been accessed would be leaked). Nonetheless, these two modifications would introduce a significant performance overhead to the PPSS protocol, as the performance benefits of implementing a batching retrieval are lost and the penalty due to the the PIR access of a much larger dataset must be kept into account. Since we deem the leakage related to the execution of queries with wildcards acceptable in most of the practical use case scenarios, we recommend the described countermeasure only for specific use cases where the information leakage about the structure of queries with well-formed pattern is actually a sensitive data.

7 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 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 [17] and a proper parametrization of the generalized Pailler algorithms provided by the LIBHCS library [39], to implement the PIR-related cryptographic operations. We relied on the OPENSSL ver. 1.0.2r [22] for all the symmetric cryptographic operations: the AES-128 CounTeR (CTR) mode primitive for the cell-wise encryption/decryption of $[[\mathbf{D}]]=(\langle C\rangle, \langle SA\rangle)$; the CMAC primitive based on the AES-128 Cipher Block Chaining (CBC) mode of operation to compute the Message Authentication Code (MAC) associated with each entry in $[[\mathbf{D}]]$; the AES-128 primitive with the Electronic Code Book (ECB) mode of operation to implement a PRF yielding an entry-specific secret key employed for the MAC computation when fed with a master secret key and the position of entry at hand. The implementation of our PPSS protocol (except for the batched retrieval optimization and the queries with wildcard characters) is publicly available online [26], 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.

We chose as our case study a genomic dataset in the widely employed FASTA format [7], which employs an alphabet of five characters 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 [15].

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 [2].

In the actual implementation employed for the experimental campaign, we introduced some optimizations which allowed us 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 – 7 of Algorithm 5, 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 each. To reduce the memory footprint of this array, we derived an *hybrid* representation between M and the BWT L: given a parameter R, referred to as *sample period*, we constructed an array C_R with $\lceil \frac{n+1}{R} \rceil$ entries, where $C_R[j]$ is a tuple with two elements, the first one being $M[:][j \cdot R]$, that is the $j \cdot R$-th column of M, and the second one being the substring of the BWT L corresponding to characters at positions $\{j \cdot R, \ldots, j \cdot R + R - 1\}$ (i.e., $L[j \cdot R, \ldots, j \cdot R + R - 1]$). 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 Alg. 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 among the first i mod R characters of the substring 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 bunch of 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 Alg. 6, that compute the number of occurrences and the set of positions of the leading character of the occurrences 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

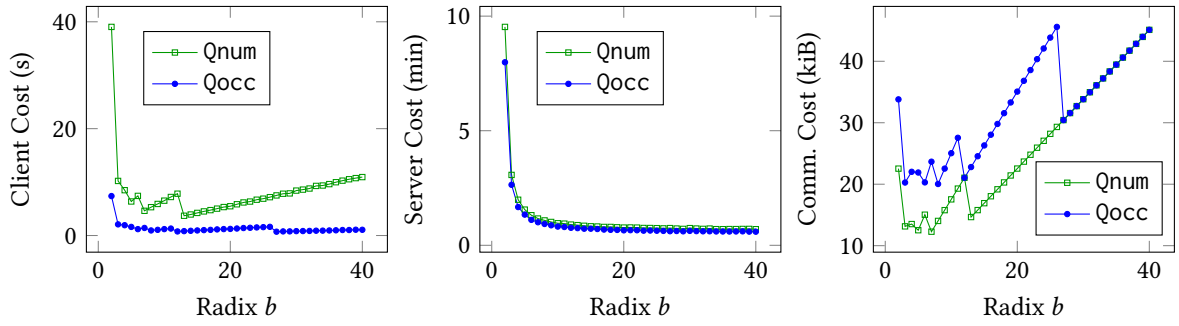

Fig. 3. Performance of our PPSS protocol as a function of the radix b employed in the PIR algorithms. Private search of $q = CTGCAG$ in a genome with $500k$ nucleotides

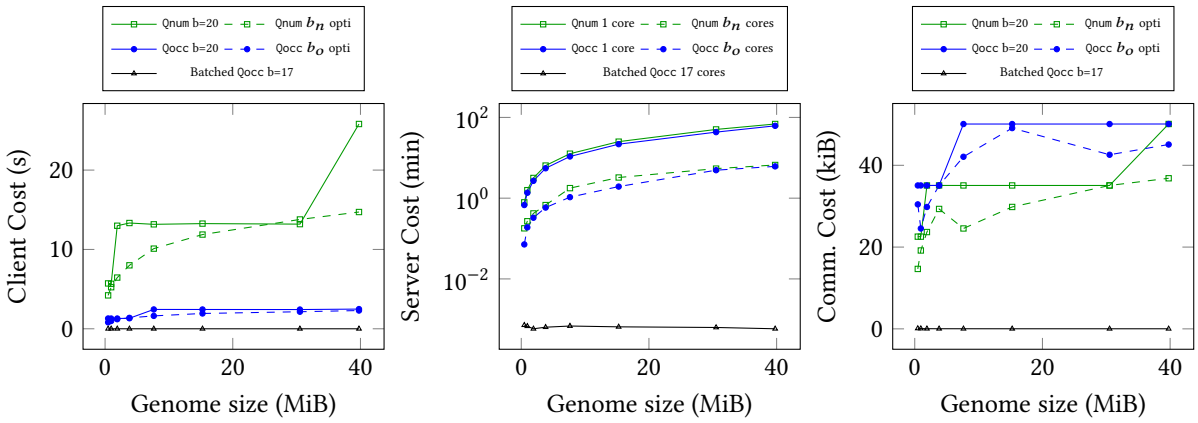

Fig. 4. Performance of our PPSS protocol as a function of the genomic document size to find one occurrence of the substring $q = CTGCAG$. Considering each document size in increasing order, the optimal values of radixes b_n and b_o employed during the experiments are $\{13, 17, 21, 26, 14, 17, 20, 21\}$ and $\{27, 14, 17, 20, 24, 28, 17, 18\}$, respectively. In case of batched retrieval of occurrences, the optimal radix b employed is always 17. The costs for the batched retrieval are divided by the number of retrieved occurrences, which amounts to $\{248, 389, 926, 1501, 2368, 4929, 11138, 18168\}$, respectively.

them is proportional to their number. We remark that the communication costs reported in our results refer to a single round of communication. In Fig. 3 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 considered a single-core implementation where we employed the same value $b = 20$ for genomes of increasing size in order 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 `PIR-Search` procedure of the Lipmaa's PIR protocol. Specifically, we adopted a simple parallelization strategy which employs b cores to simultaneously compute all the b recursive calls of Alg. 7. For these tests, we employed the optimal values

b_n and b_o for each document size. The results of these tests are shown in Fig. 4. Regarding the server cost, we observe a linear trend in both the single-core (continuous lines in Fig. 4) and the multi-core implementations (dashed lines in Fig. 4); 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. 4 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. Concerning the client computational cost, during our experiments we also measured the time required by the client to verify the integrity of the entries retrieved from the outsourced arrays. As the size of each of the said entries is always $O(\log(N))$ bits, the computational effort to compute the MAC tag associated with a single entry is always the same regardless of which one among the three outsourced arrays composing the full-text index at server side is accessed. Our experiments revealed that such an effort is negligible w.r.t. the execution time of the `PIR-Retrieve` operation. Indeed, at client side, the computational cost of the entire process of verifying the integrity of an entry retrieved from the server amounts to 3 μs, while the one of a single run of the `PIR-Retrieve` procedure fetching such an entry from the server takes 930 ms. We remark that the latter cost is averaged over dataset sizes ranging from 0.5 KiB to 40 MiB, as it is dependent from the size of the accessed array.

We also evaluated our enhanced protocol with the batched retrieval method, which is able to fetch all the occurrences in a single round of communication. To fairly compare this approach with the non batched one, we report in Fig. 4 the amortized client, server and communication costs, i.e.: the costs referring to the batched retrieval solution are divided by the total number of occurrences o_q. The results clearly outline the benefits of the batched approach, as all costs in the non-batched version are lowered by a factor roughly proportional to the number of occurrences. Although this result is expected at server side, as its computational cost with the batched approach is independent from o_q, the computational and bandwidth-savings for the client and communication costs, respectively, are definitely more interesting. Indeed, it is worth noting that in our non-batched protocol the number of PIR trapdoors computed and sent by the client and the number of PIR replies received and decrypted by the client are both proportional to o_q, while in our enhanced protocol the size of the single PIR reply is the only component depending from o_q. Thus, considering that both the size of the PIR trapdoor and the client cost to compute it are asymptotically higher than the ones referring to the PIR reply, the reasons for the observed performance benefits become clear. We remark that, for the client cost, the non-amortized cost to retrieve all the o_q occurrences is even smaller than the cost to retrieve one occurrence in the non-batched method, because the client builds a PIR trapdoor for a dataset with $\frac{n}{o_q}$ elements instead of n. Finally, since the client can retrieve in a single round all the o_q occurrences found in the `Qnum` phase of the `Query` algorithm, the performance benefits exhibited by the `Qocc` phase of the enhanced protocol increases proportionally to o_q. Indeed, in Fig. 4, the trend of the amortized costs is roughly constant or slightly decreasing as o_q increases as a function of the document size.

Willing to compare the execution time of our protocol with the one of the BWT-based substring-search procedure outlined in Alg. 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 Alg. 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 Alg. 6 has a computational complexity depending linearly on the size of the outsourced document.

In Fig. 5, we also report the execution time for genomes of increasing size of the `Setup` procedure in Alg. 5, which builds the privacy-preserving representation $[[\mathbf{D}]]$ of the dataset. 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

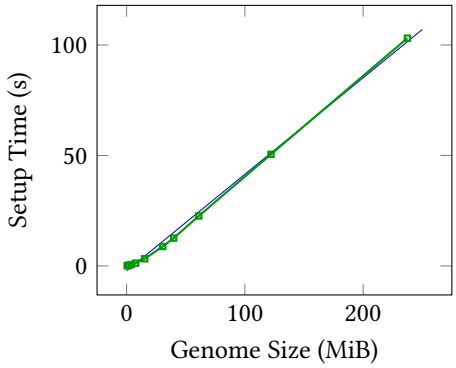

Fig. 5. Execution time of the Setup procedure for genomes of increasing size. The blue line shows the fit between the experimental data and the linear model given by $SetupTime = 0.4369 * GenomeSize - 2.2$

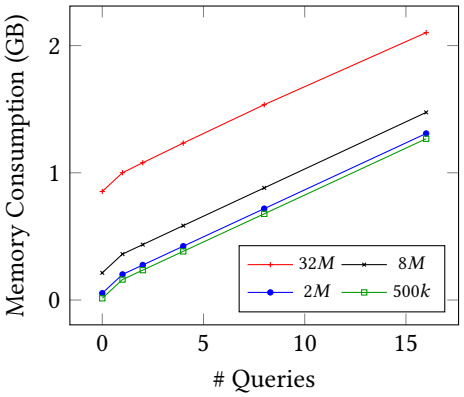

Fig. 6. Memory consumption of our PPSS protocol when multiple simultaneous queries are performed. Each line represent a genome with a different size

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.

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. Fig. 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.

Evaluation of Queries with Wildcards

We also implemented the StarFreeQuery and the QueryPattern procedures, with the aim of experimentally validating their correctness as well as evaluating their performance. In our implementation of the StarFreeQuery procedure, we relied on the Perl Compatible Regular Expressions (PCRE) library [19] (ver. 10.35) for the MatchShortestPrefix procedure (line 21 in Alg. 8), which matches the portions of the string s privately retrieved from the outsourced encrypted array $\langle s \rangle$ (line 19). In particular, we employ the option PCRE_UNGREEDY to find the shortest match of a pattern instead of the longest one, which is the default behavior of PCRE library. Similarly to the construction of the outsourced suffix array, we packed in a single ciphertext of $O(\log(N))$ bits as many characters as possible from s, hence reducing as much as possible the number of entries of the array $\langle s \rangle$. In our evaluation, we employed a parallel implementation of the PIR-Search procedures on server side.

In the evaluation of StarFreeQuery procedure, we considered the well-formed star-free pattern

$$\mathrm{p} = ?(GC|A|)GCCTATCG(G|TAC|??)([!CT]?|)TA?(TG|CGT|TA|[ACG][ATG])GTC(|?)$$

which is decomposed according to Lemma 5.5 in 3 wildcard-free subtrings (highlighted in red) and 5 substrings containing wildcards (highlighted in blue). Such pattern allows to fully validate the capability of our privacy-preserving StarFreeQuery procedure as it includes all the legit wildcards defined in our format, while reasonably representing the type of well-formed star-free patterns that may be matched in our PPSS protocol. Indeed, this pattern matches substrings with length ranging from 18 to 26 characters, with the longest matches including

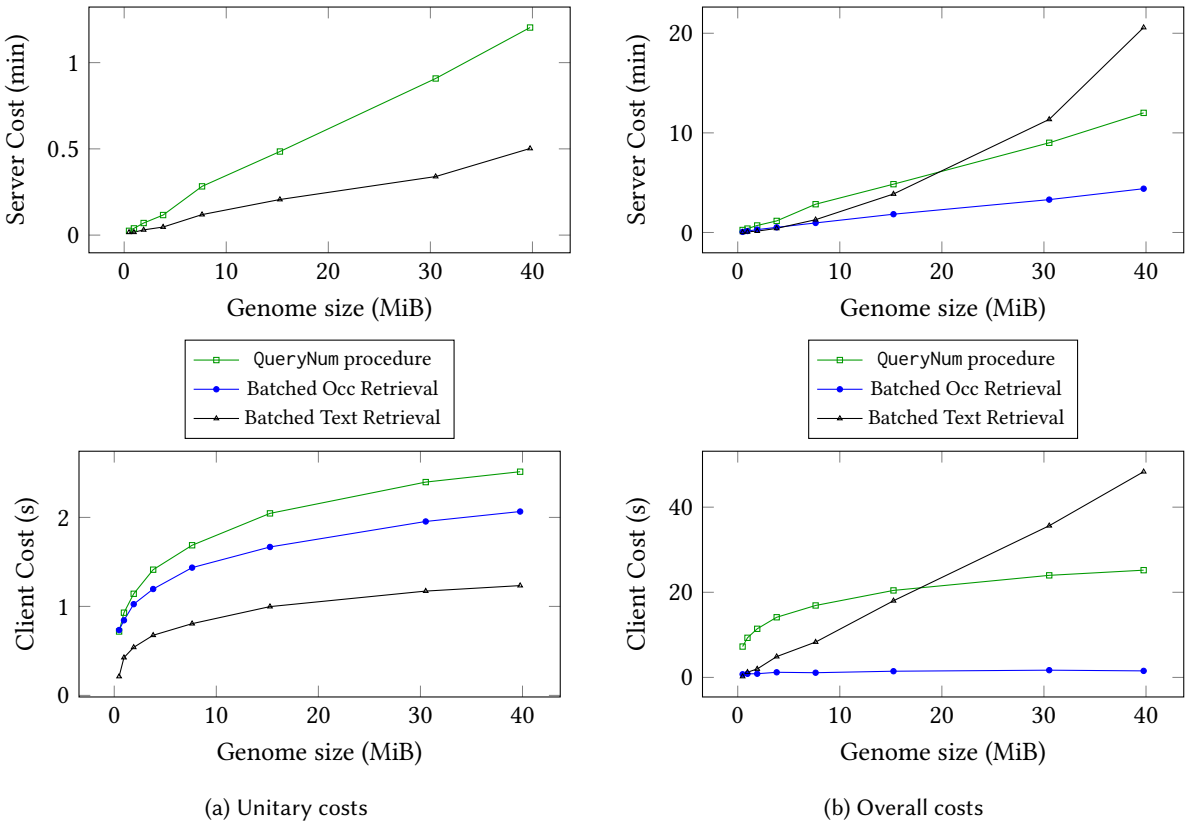

Fig. 7. Computational costs of a multi-core implementation of `StarFreeQuery` procedure of our PPSS protocol as a function of the genome size. The tested query retrieves all the occurrences of the well-formed star-free pattern $\mathfrak{p} = ?(GC|A|)GCCTATCG(G|TAC|??)([!CT]?|)TA?(TG|CGT|TA|[ACG][ATG])GTC(|?)$

50% of wildcard characters. The issuing of queries with patterns that matches more than 50% of the characters through wildcards is not representative of the usual application scenarios.

The computational costs of the `StarFreeQuery` procedure for different dataset sizes are reported in Fig. 7. In our evaluation, we split the computational and communication costs according to the three most computationally intensive operations of the `StarFreeQuery` procedure: the computation of the number of occurrences of each of the k wildcard-free substrings of the searched well-formed star-free pattern $\mathfrak{p}$, performed with the k executions of the `QueryNum` procedure (line 10 in Alg. 8); the batched retrieval of the o_{min} occurrences of the wildcard-free substring with the least number of occurrences (line 13), labeled as *Batched Occ Retrieval* in Fig. 7; the batched retrieval of the portions of the string s where the occurrences of $\mathfrak{p}$ can be found (line 19), labeled as *Batched Text Retrieval* in Fig. 7. Since the costs of the `QueryNum` procedure depend on the number of characters of the k wildcard-free substrings of the pattern $\mathfrak{p}$, and the costs of the *Batched Text Retrieval* depend on the number o_{min} of portions of s that must be retrieved, we report in Fig. 7(a) also the unitary costs for both these operations; conversely, since the server cost of the *Batched Occ Retrieval* is independent from the o_{min} occurrences sent to the client, we do not report its unitary cost. We observe that the costs of the *Batched Text Retrieval* are more than

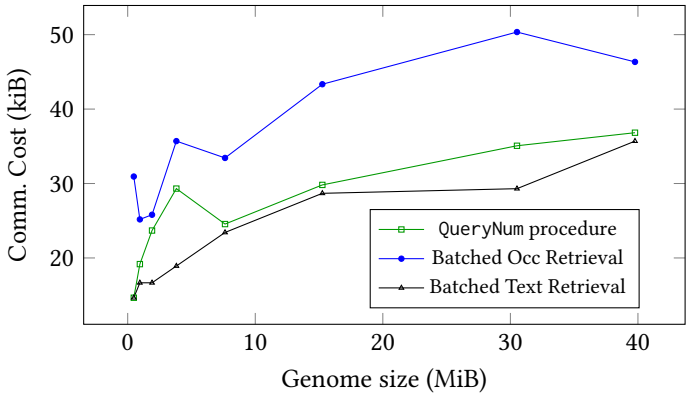

Fig. 8. Communication cost of the `StarFreeQuery` procedure to retrieve all the occurrences of the well-formed star-free pattern $\mathfrak{p}$ = ?(*GC*|*A*|)*GCCTATCG*(*G*|*TAC*|??)([!*CT*]?|)*TA*?(*TG*|*CGT*|*TA*|[*ACG*][*ATG*])*GTC*(|?)

halved w.r.t. the costs of the `QueryNum` procedure, which is mostly due to the smaller size of the outsourced array $\langle s \rangle$ w.r.t. the encrypted array $\langle C \rangle$, in turn leading to faster PIR queries. All the unitary costs show the expected linear and poly-logarithmic trends in server and client costs, respectively.

The overall computational costs of the privacy-preserving query for the well-formed star-free pattern $\mathfrak{p}$ are reported in Fig. 7(b). From the experimental data, we observe that the performance mostly depend on the costs for *Batched Text Retrieval*, which grow linearly with the size of the dataset. In case of the client cost, the linear trend is due to the increasing number o_{min} of portions of s that must be retrieved: indeed, the occurrences o_{min} of the substring *GCCTATCG* (i.e., the wildcard-free one with the least number of occurrences) vary from 1 to 41 with an increasing dataset size. In case of the server cost, the linear trend is given by the growth of both the unitary cost and the number of retrievals, which grow linearly with the dataset size and o_{min}, respectively. The costs of the `QueryNum` and *Batched Occ Retrieval* are less affected by the increasing number of occurrences, showing similar trends to the corresponding operations performed in the evaluation of the `Query` procedure (Fig. 4).

Considering the entire chromosome (i.e., the biggest among the tested datasets), we observe that the overall response time of our `StarFreeQuery` procedure for the well-formed star-free pattern $\mathfrak{p}$, given by the sum of the three components reported in Fig. 7(b), is about 35 minutes, which amounts to approximately four times the cost of the `Query` procedure for wildcard-free substrings. This performance gap is mostly due to the dependence of the server cost on the number o_{min}, which is a non tight upper bound on the number of matches of $\mathfrak{p}$, while we recall that the server cost of the `Query` procedure, when the batched retrieval method is employed, is independent from the number of occurrences of the queried string. Nonetheless, we deem the observed performance gap as acceptable, given the unprecedented expressiveness achieved by our privacy-preserving pattern matching queries.

In addition, although not reported in the client cost of our pattern matching query, we experimentally verified that identifying the occurrences of $\mathfrak{p}$ in the portions of the string s fetched from the outsourced array $\langle s \rangle$, which corresponds to lines 20– 24 in Alg.8, has a negligible impact on client cost: indeed, this operation requires 180 μs averaged over all the dataset sizes reported in Fig. 7, while the overall client cost is on average about 30 seconds, hereby showing 5 order of magnitudes of difference.

Concerning the communication cost of our query, reported in Fig. 8, we observe that it is roughly equivalent to the cost reported for the `Query` procedure in Fig. 4. Indeed, also for the `StarFreeQuery` procedure the communication cost is mostly due to the batched retrieval of the occurrences, as it fetches in a single round an

amount of data proportional to the number of occurrences o_{min}. Despite the non smooth behavior, which is given by the different number of recursive levels obtained by employing an optimal value of the radix b in the Lipmaa's PIR protocol, overall the communication cost still exhibits the expected polylogarithmic trend, showing that our PPSS protocol allows to increase the expressiveness of the queries while retaining approximately the same bandwidth.

Finally, we did not thoroughly evaluate the overall performance of the `QueryPattern` procedure, as it is rather obvious from its description in Alg. 9 that the computational and communication costs can be easily derived from the corresponding costs of the k `StarFreeQuery` procedures invoked at line 4. Conversely, we focused our evaluation on the estimation of the impact on the client cost of the `QueryPattern` procedure of the `MatchOcc` procedure invoked at line 8 in Alg. 9, which computes the occurrences of the well-formed pattern $p = \alpha_1 * \cdots * \alpha_k$ from the occurrences of the k well-formed star-free patterns returned by the k `StarFreeQuery` procedures. In our evaluation, we employed the well-formed pattern $\mathfrak{p} = GCAATC * CTGAC * TGAC$, as we considered it as a good representative of the well-formed patterns that may be searched by users of our PPSS protocol; indeed, since in general the $*$ wildcard does not significantly restrict the number of matches of the well-formed star-free patterns composing the matched well-formed pattern, we expect that users in our PPSS protocol would issue queries for patterns composed by infrequent well-formed star-free patterns, in order to avoid an unnecessary blowup of the number of matched occurrences. Our evaluation revealed that the impact of `MatchOcc` procedure is limited: indeed, it requires only 3 s to compute the 7459 occurrences of the pattern $\mathfrak{p}$ over the entire chromosome, which is the biggest among the tested datasets, from the occurrences of the patterns *GCAATC*, *CTGAC*, *TGA*, as opposed to the 71.5 s of overall client cost of the three `StarFreeQuery` procedures invoked for the patterns *GCAATC*, *CTGAC*, *TGA*.

8 CONCLUDING REMARKS

We presented the first privacy-preserving 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. We further extended the proposed PPSS protocol with the capability of querying strings containing wildcard characters. 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 from the proposed batched retrieval approach.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. 2018. PIR with Compressed Queries and Amortized Query Processing. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 962–979. https://doi.org/10.1109/SP.2018.00062

[2] 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

[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] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016 (2016), 86.

[9] 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

[10] 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. (LNCS)*, Kwangjo Kim (Ed.), Vol. 1992. Springer, 119–136. https://doi.org/10.1007/3-540-44586-2_9

[11] Sky Faber, Stanislaw Jarecki, Hugo Krawczyk, Quan Nguyen, Marcel-Catalin Rosu, and Michael Steiner. 2015. Rich Queries on Encrypted Data: Beyond Exact Matches. In *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part II (Lecture Notes in Computer Science)*, Günther Pernul, Peter Y. A. Ryan, and Edgar R. Weippl (Eds.), Vol. 9327. Springer, 123–145. https://doi.org/10.1007/978-3-319-24177-7_7

[12] 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

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

[14] Paolo Ferragina and Rossano Venturini. 2010. The compressed permuterm index. *ACM Trans. Algorithms* 7, 1 (2010), 10:1–10:21. https://doi.org/10.1145/1868237.1868248

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

[16] 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

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

[18] 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

[19] Philip Hazel. 2015. *PCRE - Perl Compatible Regular Expressions.* https://www.pcre.org.

[20] Thang Hoang, Attila A. Yavuz, F. Betül Durak, and Jorge Guajardo. 2019. A multi-server oblivious dynamic searchable encryption framework. *J. Comput. Secur.* 27, 6 (2019), 649–676. https://doi.org/10.3233/JCS-191300

[21] 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

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

[23] 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

[24] Kaitai Liang, Xinyi Huang, Fuchun Guo, and Joseph K. Liu. 2016. Privacy-Preserving and Regular Language Search Over Encrypted Cloud Data. *IEEE Trans. Inf. Forensics Secur.* 11, 10 (2016), 2365–2376. https://doi.org/10.1109/TIFS.2016.2581316

[25] 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. (LNCS)*, J. Zhou, J. López, R. H. Deng, and F. Bao (Eds.), Vol. 3650. Springer, 314–328. https://doi.org/10.1007/11556992_23

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

[27] Nicholas Mainardi, Alessandro Barenghi, and Gerardo Pelosi. 2019. Privacy preserving substring search protocol with polylogarithmic communication cost. In *Proceedings of the 35th Annual Computer Security Applications Conference, ACSAC 2019, San Juan, PR, USA, December 09-13, 2019*, David Balenson (Ed.). ACM, 297–312. https://doi.org/10.1145/3359789.3359842

[28] Nicholas Mainardi, Davide Sampietro, Alessandro Barenghi, and Gerardo Pelosi. 2020. Efficient Oblivious Substring Search via Architectural Support. In *ACSAC '20: Annual Computer Security Applications Conference, Virtual Event / Austin, TX, USA, 7-11 December, 2020.* ACM, 526–541. https://doi.org/10.1145/3427228.3427296

[29] 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

[30] 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

[31] 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 (LNCS)*, Jacques Stern (Ed.), Vol. 1592. Springer, 223–238. https://doi.org/10.1007/3-540-48910-X_16

[32] Shiyue Qin, Fucai Zhou, Zongye Zhang, and Zifeng Xu. 2020. Privacy-Preserving Substring Search on Multi-Source Encrypted Gene Data. *IEEE Access* 8 (2020), 50472–50484. https://doi.org/10.1109/ACCESS.2020.2980375

[33] 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

[34] 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

[35] 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

[36] 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

[37] Julien P. Stern. 1998. A New Efficient All-Or-Nothing Disclosure of Secrets Protocol. In *Advances in Cryptology - ASIACRYPT '98, International Conference on the Theory and Applications of Cryptology and Information Security, Beijing, China, October 18-22, 1998, Proceedings (LNCS)*, Kazuo Ohta and Dingyi Pei (Eds.), Vol. 1514. Springer, 357–371. https://doi.org/10.1007/3-540-49649-1_28

[38] 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

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

[40] 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

[41] Yang Yang, Xianghan Zheng, Chunming Rong, and Wenzhong Guo. 2020. Efficient Regular Language Search for Secure Cloud Storage. *IEEE Trans. Cloud Comput.* 8, 3 (2020), 805–818. https://doi.org/10.1109/TCC.2018.2814594

A SECURITY PROOF

Theorem 6.3 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 6.2, 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 6.3, 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 $[[q]]_j^{Id}, j = 1, \ldots, w$ 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 $\mathcal{L}_D = (\sum_{i=1}^{z}(\texttt{len}(D_i)+1), \omega)$, where the first term $\sum_{i=1}^{z}(\texttt{len}(D_i)+1)$ 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 $\mathcal{L}_D$, $\mathcal{L}_q = (\texttt{len}(q), b, |O_{\mathrm{D},q}|)$ 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 $t_{SA} = \lceil \log_b(n+1) \rceil$ and $t_C = \lceil \log_b((n+1)\cdot(|\Sigma|+1)) \rceil$. 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 $o_q = |O_{\mathrm{D},q}|$ 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 $\mathcal{E}$.`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., $\texttt{FLAHE.E}_{pk}^{l}$) 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.