Abstract. We study the problem of private outsourced sorting of encrypted data. We start by proposing a novel sorting protocol that allows a user to outsource his data to a cloud server in an encrypted form and then request the server to perform computations on this data and sort the result. To perform the sorting the server is assisted by a secure coprocessor with minimal computational and memory resources. The server and the coprocessor are assumed to be honest but curious, i.e., they honestly follow the protocol but are interested in learning more about the user data. We refer to the new protocol as private outsourced sorting since it guarantees that neither the server nor the coprocessor learn anything about user data as long as they are non-colluding. We formally define private outsourced sorting and provide an efficient construction that is based on semi-homomorphic encryption.

As an application of our private sort, we present MRSE: the first scheme for outsourced search over encrypted data that efficiently answers multi-term queries with the result ranked using frequency of query terms in the data, while maintaining data privacy. To construct MRSE we use searchable encryption techniques combined with our new private sort framework. Finally, although not discussed in this work, we believe that our private sort framework can turn out to be an important tool for more applications that require outsourced sorting while maintaining data privacy, e.g., database queries.

1 Introduction

Consider the following scenario: Mr. Smith owns an array of data elements $A$ that he outsources to an honest-but-curious untrusted party, Brad. Mr. Smith then asks Brad to perform various linear operations on the elements of $A$ resulting in an array $B$ and then, sort $B$ and return the sorted result, $B_{\text{sorted}}$, back to him. However, Mr. Smith does not trust Brad and wishes to keep $A$, $B$ and $B_{\text{sorted}}$ secret. Thus, he decides to encrypt every element of $A$ using a public key semantically secure cryptosystem. To let Brad perform computations on the encrypted array $A$, Mr. Smith can simply use a semi-homomorphic cryptosystem that supports addition of ciphertexts. Hence, the remaining question is: how is Brad going to sort the encrypted $B$?

If the array $A$ was encrypted under a fully homomorphic encryption scheme (FHE) [19, 44], then Brad could perform sorting himself. FHE allows one to perform both homomorphic addition and multiplication, thus, Brad could simply translate a sorting network into a circuit and apply it to $B$. Unfortunately, all known FHE schemes are still too far away from being practical for real life applications and cannot be implemented by Brad. Hence, Brad suggests to Mr. Smith to use order preserving encryption (OPE) [8] for $A$ since this makes sorting a trivial task for him. Mr. Smith gets excited but soon realizes that an encryption scheme that supports homomorphic addition and comparison of ciphertexts is not secure even against a ciphertext attack (as shown by Rivest et al. [41]). If Mr. Smith just wanted Brad to sort $A$, then OPE would be sufficient but it is crucial to Mr. Smith that Brad can also perform certain operations on $A$. Moreover, allowing Brad to learn the relative order of elements in $A$ violates owner’s privacy requirements.

Mr. Smith is determined to design a protocol for private outsourced sorting that will be efficient, preserve his data privacy and allow Brad to perform certain computations on his data. Thus he decides to encrypt his data with a semi-homomorphic cryptosystem and add another party to the model: Angelina. Angelina is given the decryption key and her sole role is to help Brad with sorting. Mr. Smith assumes that Brad and Angelina are not colluding with each other but both are interested in learning more about his data. Hence, he extends his privacy requirements as follows: after Brad’s and Angelina’s interaction Brad receives $B_{\text{sorted}}$ which is the sorting of an encrypted $B$, while neither of them learns anything about the plaintext values of $B$ nor $B_{\text{sorted}}$. It follows from the privacy requirement that Angelina never sees an encryption of neither $B$ nor $B_{\text{sorted}}$, otherwise she could trivially decrypt them.

* Full version of [4].
The Brad and Angelina model is often encountered in reality. We can see Brad as the provider of cloud storage and computation who is trusted to perform operations on clients’ data but at the same time may be curious to learn something about them. Angelina models a secure coprocessor (e.g., the IBM PCIe\(^1\) or the Freescale C29x\(^2\)) that resides in the cloud server and is invoked only to perform relatively small computations. Secure coprocessors provide isolated execution environments, which is important for our model since it ensures that the two parties are separated. We note that the assumption of non-colluding is justified since the cloud provider and secure co-processor usually are supplied by different companies and, hence, have also commercial interests not to collude.

In this paper, we present private outsourced sort executed by two parties such that neither of them learns anything about the data involved. This setting is perfect for letting one use not only storage but also computing services of the cloud environment without sacrificing privacy. We give the formal definition and present an efficient construction that implements private outsourced sort by relying only on additively homomorphic properties of an encryption scheme. Sorting is, arguably, one of the most common and well studied computations \([30]\) over data in the “before cloud era” which indicates that it will be of interest as an outsourced computation to the cloud. Our model is of particular interest since it does not only allow the cloud to privately and efficiently sort encrypted data but at the same time allows for certain computations on the data. Hence, it can be a useful tool for answering sophisticated queries on databases of encrypted data and, for example, return top results satisfying the query. To give a concrete application of our new sorting framework, we consider the problem of outsourcing search over encrypted data to the cloud where the result has to be ranked according to its relevance to the query.

**Outsourced Ranked Text Search.** Imagine a client who outsources a collection of documents to an honest-but-curious server and then asks the server to return a subset of the document collection satisfying a search query. Our goal is to return search results sorted using the tf-idf method that is based on term (or keyword) frequency (tf) and inverse document frequency (idf) \([48]\). This basic ranking method uses the frequency of keywords in each document and whole collection in order to decide how important a word is to the collection and the query\(^3\). Moreover, this ranking fits well free text search queries \([35]\) that are, arguably, the most common and intuitive queries to online search engines. In order to perform a ranked search of this type efficiently, a search index is created in advance where an idf of every term in every document in the collection is stored.

In the cloud based information retrieval setting, where the cloud server is not trusted, the client outsources the search index to the server in an encrypted format and then submits keyword search queries to the server. If we only allow single term queries then a solution is relative easy: the client creates the search index where each term is stored with a list of documents sorted by relevance. Then, he encrypts the index using some symmetric searchable encryption scheme (SSE) and outsources it to the server. When the client wants to search for a term, he submits a trapdoor to the server, who using the trapdoor can locate the term in the index and return the encrypted list of documents to the user.

However, precomputing sorted results becomes infeasible and not scalable when the system is required to handle multi-term queries, since the result depends on all the keywords in the query which is not known in advance. Hence, the client has to upload the search index where frequencies (idfs) for every term are ordered according to document identifiers. When querying the system, the client creates a trapdoor for every term in the query and submits them to the server. The server then locates the corresponding rows in the SSE encrypted search index and is left with two tasks. First, he has to add the located rows of encrypted frequencies together in order to compute the score of every document w.r.t. the query. Second, he has to sort the resulting list of encrypted scores to be able to return the most relevant document identifiers (ids) to the client.

It is easy to see that our private outsourced sorting is the perfect tool for the scenario described above. The client can encrypt the keyword frequencies using a semi-homomorphic encryption scheme (e.g., Paillier \([39]\)) and then outsource them to the cloud server, \(S_1\). \(S_1\) is equipped with a secure co-processor, \(S_2\), who stores the decryption key. Our mechanism allows the cloud server to first add the encrypted frequencies of the keywords in the query and then sort them with the help of the secure co-processor. Similar to the model of private outsourced sorting, \(S_1\) and \(S_2\) are non-colluding and behave according to the protocol when interacting with each other. However, both of them would like to learn about client’s document collection.

\(^1\) [http://www-03.ibm.com/security/cryptocards/pciecc/overview.shtml](http://www-03.ibm.com/security/cryptocards/pciecc/overview.shtml)


\(^3\) For example, a short document is ranked higher than a long document when a keyword appears the same number of times in both of them. It would also give higher preference to keywords in the query that appear less frequently in the collection.
Our resulting system allows the client to perform multi-keyword search over outsourced encrypted data and maintain privacy against both $S_1$ and $S_2$: $S_1$ only learns the search pattern\(^4\) during the query phase and nothing about the collection besides the number of documents and unique words it contains. $S_2$, on the other hand, learns nothing but the number of documents in the collection, $N$, and that the client has queried the system. Data owner, user and client are used interchangeably, although our system can support the data owner sharing search access to his cloud stored data with other users. We refer to our proposed construction as Multi-keyword Ranked Searchable Encryption (MRSE) and give its overview in Figure 1. We note that in MRSE, and previous work, $S_1$ returns only document ids, e.g., document titles or URLs to public locations, but not the actual documents which $S_1$ also does not have to store. In Section 6 we argue why it is the case and why a secure system that also returns the actual documents is likely to be very expensive.

Our contributions are summarized below:

- Formally define private outsourced sorting (Definition 4) and present a simulation based privacy definition (Definition 5) that guarantees that neither the cloud server nor the secure coprocessor learn anything about the user data as long as they are not colluding.
- We present an efficient implementation of private outsourced sorting in Section 3.2 that requires $O(N (\log N)^2)$ time for sorting, where $N$ is the total number of elements to be sorted.
- We apply our sorting tool in a three party model (client-$S_1$-$S_2$) for ranked multi-keyword search over encrypted data, MRSE. This is the first system that can efficiently support multi-keyword search and compute a ranked result based on word frequencies in a secure and private way. We give privacy definitions for MRSE in Section 4.1 and present an efficient construction in Section 4.2. Finally, in Section 5, we present an extension of MRSE that offers better efficiency but weaker privacy guarantees.

2 Building Blocks for Private Sort and MRSE

In Table 1 we summarize the notation and in Table 2 the protocols used throughout the paper. Then, we present the building blocks used in our construction.

2.1 Homomorphic encryption

Homomorphic encryption allows one to perform computations directly on ciphertexts without first decrypting them. For example, it allows to add two ciphertexts $c_1, c_2$ to a new ciphertext $c'$ such that when $c'$ is decrypted, the result corresponds to the addition of plain values of $c_1$ and $c_2$, $m_1 + m_2$. A variety of homomorphic cryptosystems have been proposed in the literature. Some schemes are “partially homomorphic” and support only a specific type of computation (addition or a single multiplication) [39, 40]. Fully homomorphic encryption schemes [20], on the other hand, support arbitrary number of additions and multiplications which means that any circuit can be homomorphically evaluated. Although being so powerful, those schemes are computationally expensive to be used in practice. However, “partially homomorphic” schemes are efficient and as we show are sufficient for our purposes.

\(^4\) The search pattern includes the number of terms in each query as well as if some of the terms were already requested and in which queries. Note that this is also leaked in most of the known SSE schemes.
Table 1: Notation.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k$</td>
<td>security parameter</td>
</tr>
<tr>
<td>$Gen_{SS}, E_{SS}, (PK,SK)$</td>
<td>semantically secure (SS) keygen, encr. and keys</td>
</tr>
<tr>
<td>$KP = (PK_{P}, SK_{P})$</td>
<td>Paillier public/secret keys</td>
</tr>
<tr>
<td>$K_{QR} = (PK_{QR}, SK_{QR})$</td>
<td>QR public/secret keys</td>
</tr>
<tr>
<td>$[m], [m]$</td>
<td>$m$ encrypted using first and second layers of Paillier</td>
</tr>
<tr>
<td>$[[m]]$</td>
<td>$m$ encrypted using both layers of Paillier</td>
</tr>
<tr>
<td>$</td>
<td></td>
</tr>
<tr>
<td>$Gen_{SSE}, E_{SSE}, SK_{SSE}$</td>
<td>SSE keygen, encr. and secret key</td>
</tr>
<tr>
<td>$D = { D_{1}, \ldots, D_{N} }$</td>
<td>document collection of size $N$</td>
</tr>
<tr>
<td>$t, T$</td>
<td>term/keyword and its SSE trapdoor</td>
</tr>
<tr>
<td>$M$</td>
<td>number of unique terms in $D$</td>
</tr>
<tr>
<td>$q = (t_{1}, \ldots, t_{q})$</td>
<td>query of $l_{q}$ terms</td>
</tr>
<tr>
<td>$I$</td>
<td>secure search index</td>
</tr>
<tr>
<td>$F, EF$</td>
<td>frequency table and encryption of it</td>
</tr>
<tr>
<td>$ScT$</td>
<td>secure score data structure</td>
</tr>
</tbody>
</table>

Table 2: Summary of interactive cryptographic protocols run between two parties, $S_{1}$ and $S_{2}$, where $EncSelect2$ is run only by $S_{1}$. As denoted below, some of these protocols are extensions of existing protocols, while others were developed in this work.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>$S_{1}$ Input</th>
<th>$S_{2}$ Input</th>
<th>$S_{1}$ Output</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>StripEnc</td>
<td>$PK_{P}, [x]$</td>
<td>$KP$</td>
<td>$[x]$</td>
<td>Strips one layer of encryption</td>
</tr>
<tr>
<td>EncSelect  [2]</td>
<td>$PK_{P}, [a], [b], [v]$</td>
<td>$KP$</td>
<td>$(1-v)a + vb$</td>
<td>$EncSelect2$ w/ different encryption</td>
</tr>
<tr>
<td>ReEncryptBit</td>
<td>$PK_{P}, PK_{QR}, [v]$</td>
<td>$K_{P}, K_{QR}$</td>
<td>$[v]$</td>
<td>Re-encrypts $v$ using Paillier</td>
</tr>
<tr>
<td>EncCompare [10]</td>
<td>$PK_{P}, [a], [b]$</td>
<td>$KP$</td>
<td>$[c], [d]$</td>
<td>$v$ is result of comparison of $a, b$</td>
</tr>
<tr>
<td>EncPairSort</td>
<td>$PK_{P}, [a], [b]$</td>
<td>$KP$</td>
<td>$[e], [d]$</td>
<td>Sorts encrypted $a, b$</td>
</tr>
<tr>
<td>EncSort</td>
<td>$PK_{P}, A$</td>
<td>$KP$</td>
<td>$B$</td>
<td>Sorts encrypted array $A$</td>
</tr>
</tbody>
</table>

* The protocols have been extended to fit our purposes.

**Paillier Cryptosystem** The Paillier cryptosystem [39] is a semantically secure public key encryption scheme based on the Decisional Composite Residuosity assumption. We use $[m]$ to denote an encryption of a message under Paillier cryptosystem with a public, secret key pair $KP = (PK_{P}, SK_{P})$. Paillier cryptosystem is homomorphically additive, that is, $[m_{1}] : [m_{2}] = [m_{1} + m_{2}]$. More specifically, the Paillier cryptosystem is defined as follows:

**Key Generation.** To construct the public key, set an RSA modulus $n = pq$ of $k$ bits where $p$ and $q$ large primes such that $\gcd(pq, (p-1)(q-1)) = 1$. Let $K = \text{lcm}((p-1)(q-1))$ and pick $g \in \mathbb{Z}_{n}^{*}$. The public key is the pair $PK_{P} = (n,g)$ and the secret is $SK_{P} = K$.

**Encryption.** To encrypt a message $m \in \mathbb{Z}_{n}$: choose $r \in \mathbb{Z}_{n}^{*}$ and compute $[m] = g^{mr} \mod n^{2}$.

**Decryption.** To decrypt a ciphertext $c = [m]$ compute:

$$ m = \frac{L(g^{SK_{P}}) \mod n^{2}}{L(g^{SK_{P}}) \mod n^{2}} \mod n, \text{ where } L(u) = \frac{u-1}{n}. $$

We can easily see that the Paillier cryptosystem is homomorphically additive.

$$ [m_{1}] = g^{m_{1}r_{1}^{n}} \mod n^{2} $$

$$ [m_{2}] = g^{m_{2}r_{2}^{n}} \mod n^{2} $$

$$ [m_{1}] \cdot [m_{2}] = g^{m_{1} + m_{2}(r_{1}r_{2})^{n}} \mod n^{2} = [m_{1} + m_{2}] $$
the nested encryption preserves the structure over inner ciphertexts and allows one to manipulate it as follows [2]:

\[\text{EncSelect}\]

privacy guarantees of revealing which one was picked (we adopt this operation from [2]). In particular we define

\[\text{EncSelect}\]

\[\text{Additive homomorphism and generalized Paillier encryption can be used to select one of two plaintexts without}\]

\[\text{for ciphertext multiplication).}\]

\[\text{2.2 Private Selection of Encrypted Data}\]

We note that this is the only homomorphic property that our protocols rely on (i.e., we do not require support under the same secret key. By [\(m\)] that allows to doubly encrypt messages and use the additive homomorphism of the inner encryption layer

\[\text{for stripping off one layer of Paillier encryption.}\]

\[\text{Note that the result } c \text{ is doubly encrypted.}\]

\[\text{For our setting of private sort we need a modified version of EncSelect that runs between } S_1 \text{ and } S_2 \text{ where } S_1 \text{ has as input } (\text{PK}_P, [a], [b], [v]) \text{ and using } S_2, \text{ that knows } \text{SK}_P, \text{ outputs } c \text{ encrypted using the first layer of Paillier encryption only. Thus, we propose a protocol StripEnc, where } S_1 \text{ randomizes the encryption of the value } x \text{ he wants } S_2 \text{ to re-encrypt, receives the re-encryption and removes the randomization. Hence, when } S_2 \text{ decrypts the element he receives a random value and learns nothing about } x. \text{ The complete protocol StripEnc is presented in Table 3 where we rely on the homomorphic properties of layered Paillier encryption.}\]

\[\text{Combining EncSelect2 and StripEnc, we can define } [c] \leftarrow \text{EncSelect2}(\text{PK}_P, [a], [b], [v]) \text{ that has the same functionality as EncSelect2 but returns } [c] \text{ instead of } [[c]]. \text{ We instantiate it by first calling EncSelect2 to compute } [c] \text{ and then StripEnc to securely remove one layer of encryption. Hence, } S_1\text{'s private inputs are } [a], [b], [v], \text{ } S_2\text{'s private input is } \text{SK}_P \text{ and } S_1\text{'s output is } [c]. \text{ Our EncSelect protocol is described in Table 4. We capture the privacy guarantees of EncSelect in the following definition and theorem.}\]

**Definition 1.** Let \(\Pi_{\text{EncSelect}}\) be a two party protocol for computing EncSelect functionality. \(S_1\) takes as input \((\text{PK}_P, [a], [b], [v])\) and \(S_2\) takes as input \((\text{PK}_P, \text{SK}_P)\). When \(\Pi_{\text{EncSelect}}\) terminates \(S_1\) receives the output \([c]\)

\[\text{Generalized Paillier}\]

Our construction relies on a generalization of the Paillier cryptosystem introduced by Damgård and Jurik [18]. For the generalization \(mod n^2\) replaced with \(mod n^{s+1}\) and the plaintext space of \(Z_n\) is replaced with \(Z_{n^s}\) where \(s \geq 1\) is a layer of Paillier encryption. Note that the generalized version of Paillier uses \(g \in Z_{n^{s+1}}^*\) as a public key while the secret is \(K'\) such that \(K' mod n \in \mathbb{Z}_n^*\) and \(K' \equiv 0 \mod K\), where \(K = \text{lcm}(p - 1)(q - 1)\) is the secret key of the original Paillier.

As observed by Lipmaa [34] and Adida and Wikström [2] the Paillier generalization has the special property that allows to doubly encrypt messages and use the additive homomorphism of the inner encryption layer under the same secret key. By \([m]\), as before, we denote an encryption of \(m\) using the first layer (basic Paillier encryption) and by \([m]\) we denote encryption of \(m\) using the second layer.

This extension allows a ciphertext of the first layer to be treated as a plaintext at the second layer. Moreover, the nested encryption preserves the structure over inner ciphertexts and allows one to manipulate it as follows [2]:

\[\text{Table 3: } [x] \leftarrow \text{StripEnc}(\text{PK}_P, \text{SK}_P, [x]): \text{Interactive protocol between } S_1 \text{ and } S_2 \text{ for stripping off one layer of Paillier encryption.}\]

\[
\begin{array}{c|c}
S_1(\text{PK}_P, [x]) & S_2(\text{PK}_P, \text{SK}_P) \\
\hline
\text{pick } r \in \{0, 1\}^{l+1} & \\
[[x + r]] := [[x]]^r & [[x + r]] \text{ decrypt } [[x + r]] \\
[x] := [x + r][r]^{-1} & [x + r] \text{ encrypt } [x + r] \\
\end{array}
\]

\[\text{Table 4: } [c] \leftarrow \text{EncSelect}(\text{PK}_P, \text{SK}_P, [a], [b], [v]): \text{Interactive protocol between } S_1 \text{ and } S_2 \text{ for selecting one ciphertext.}\]

\[
\begin{array}{c|c}
S_1(\text{PK}_P, [a], [b], [v]) & S_2(\text{PK}_P, \text{SK}_P) \\
\hline
[[c]] \leftarrow \text{EncSelect2}(\text{PK}_P, [a], [b], [v]) & \\
[c] \leftarrow \text{StripEnc}(\text{PK}_P, \text{SK}_P, [[c]]) \\
\end{array}
\]

\[\text{We note that this is the only homomorphic property that our protocols rely on (i.e., we do not require support for ciphertext multiplication).}\]

\[\text{2.2 Private Selection of Encrypted Data}\]

Additive homomorphism and generalized Paillier encryption can be used to select one of two plaintexts without revealing which one was picked (we adopt this operation from [2]). In particular we define \(\text{EncSelect2}(\text{PK}_P, [a], [b], [v])\) where \(a\) and \(b\) are the two plaintext values and \(v\) is a bit that indicates whether \(a\) or \(b\) should be returned. If \(v = 0\), \(\text{EncSelect2}\) returns a re-encryption of \(a\), otherwise it returns a re-encryption of \(b\). \(\text{EncSelect2}\) imitates the computation \(c = (1 - v) \times a + v \times b\) but over ciphertexts as follows:

\[\text{EncSelect2}\]

\[\text{Note that the result } c \text{ is doubly encrypted.}\]

\[\text{For our setting of private sort we need a modified version of EncSelect that runs between } S_1 \text{ and } S_2 \text{ where } S_1 \text{ has as input } (\text{PK}_P, [a], [b], [v]) \text{ and using } S_2, \text{ that knows } \text{SK}_P, \text{ outputs } c \text{ encrypted using the first layer of Paillier encryption only. Thus, we propose a protocol StripEnc, where } S_1 \text{ randomizes the encryption of the value } x \text{ he wants } S_2 \text{ to re-encrypt, receives the re-encryption and removes the randomization. Hence, when } S_2 \text{ decrypts the element he receives a random value and learns nothing about } x. \text{ The complete protocol StripEnc is presented in Table 3 where we rely on the homomorphic properties of layered Paillier encryption.}\]

\[\text{Combining EncSelect2 and StripEnc, we can define } [c] \leftarrow \text{EncSelect2}(\text{PK}_P, \text{SK}_P, [a], [b], [v]) \text{ that has the same functionality as EncSelect2 but returns } [c] \text{ instead of } [[c]]. \text{ We instantiate it by first calling EncSelect2 to compute } [[c]] \text{ and then StripEnc to securely remove one layer of encryption. Hence, } S_1\text{'s private inputs are } [a], [b], [v], \text{ } S_2\text{'s private input is } \text{SK}_P \text{ and } S_1\text{'s output is } [c]. \text{ Our EncSelect protocol is described in Table 4. We capture the privacy guarantees of EncSelect in the following definition and theorem.}\]

\[\text{Definition 1.}\]

\[\text{Let } \Pi_{\text{EncSelect}} \text{ be a two party protocol for computing EncSelect functionality. } S_1 \text{ takes as input } (\text{PK}_P, [a], [b], [v]) \text{ and } S_2 \text{ takes as input } (\text{PK}_P, \text{SK}_P). \text{ When } \Pi_{\text{EncSelect}} \text{ terminates } S_1 \text{ receives the output } [c]\]

\[\text{\footnote{We note that } v \text{ has to be encrypted using the second layer of Paillier in order to use the homomorphic properties of the cryptosystem.}\]
of EncSelect. Let \( \text{VIEW}^{	ext{EncSelect}}_{S_1} \) be all the messages that \( S_1 \) receives while running the protocol on inputs \((PK_P, SK_P, [a], [b], [v])\) and \( \text{OUTPUT}^{	ext{EncSelect}}_{S_1} \) be the output of the protocol received by \( S_1 \).

We say that \( \Pi_{\text{EncSelect}} \) privately computes \( \text{EncSelect} \) if there exists a pair of probabilistic polynomial time (PPT) simulators \((\text{Sim}_{S_1}, \text{Sim}_{S_2})\) such that

\[
(1) \ (\text{Sim}_{S_2}(PK_P, [a], [b], [v]), \text{EncSelect}(PK_P, [a], [b], [v])) \simeq \text{VIEW}^{	ext{EncSelect}}_{S_1}(PK_P, [a], [b], [v]);
\]

\[
(2) \ \text{Sim}_{S_1}(PK_P) \simeq \text{VIEW}^{	ext{EncSelect}}_{S_2}(PK_P, [a], [b], [v]),
\]

where \( PK_P \) denotes the key pair \((PK_P, SK_P)\).

**Theorem 1.** The protocol in Table 4 privately computes \( \text{EncSelect} \) functionality according to Definition 1.

**Proof.** (Sketch) \( \text{EncSelect} \) consists of \( S_1 \) running \( \text{EncSelect2} \) and then invoking an interactive protocol \( \text{StripEnc} \) between \( S_1 \) and \( S_2 \).

We build \( \text{Sim}_{S_2} \) to show that \( S_1 \) learns nothing from his interactions with \( S_2 \) as follows. \( \text{Sim}_{S_2} \) has access to \( PK_P, [a], [b] \) and \([v]\). Hence, \( \text{Sim}_{S_2} \) can run \( \text{EncSelect2} \) himself. \( \text{EncSelect2} \) uses homomorphic properties of the underlying semantically secure encryption and, hence, reveals nothing more than what can be already computed from the inputs. Now consider the messages \( S_1 \) receives during \( \text{StripEnc} \) protocol: \([x+r]\). Since \( x+r \) is encrypted and \( \text{Sim}_{S_2} \) does not know the secret key, he picks a random value \( r' \) and substitutes \([x+r]\) with \([r']\). Since the encryption scheme is semantically secure, \( S_1 \) cannot distinguish \([x+r]\) from \([r']\).

Consider the following simulator \( \text{Sim}_{S_1} \) who has access to the public, secret key pair \( PK_P \) and \( SK_P \). When participating in \( \text{EncSelect} \), \( \text{VIEW} \) of \( S_2 \) consists of \([([x+r])]\) messages in \( \text{StripEnc} \). \( \text{Sim}_{S_1} \) can decrypt and obtain \( x+r \). However, \( x+r \) is distributed independently of \( x \). Hence, \( \text{Sim}_{S_1} \) simply sends \([([r']])\) where \( r' \) is a uniform random element in \([0,1]^\ell+1\) of his choice. Values \( x+r \) and \( r' \) come from the same distribution and hence are indistinguishable for \( S_2 \).

### 2.3 Private Comparison of Encrypted Data

Imagine the problem of having two millionaires who would like to learn which of them is richer without revealing their actual wealth. Secure multiparty computation was first proposed by Yao [47] who presented a protocol based on garbled circuits that was exponential in time and space. Following Yao’s paradigm many other solutions, either based on garbled circuits or on homomorphic encryption, have been proposed [7, 16, 17]. In this setting each party \( S_1 \) and \( S_2 \) owns a number, say \( a \) and \( b \), and their goal is to solve the equation \( a \leq b \) without revealing the actual numbers \( a \) and \( b \) to each other.

In this work however, we are interested in the following private comparison setting: the first server \( S_1 \) owns two encrypted numbers \([a]\) and \([b]\) and the second server \( S_2 \) owns the secret key \( SK_P \). The goal of the protocol is for \( S_1 \) to obtain the encryption of the relation between \( a \) and \( b \) without learning neither the actual numbers nor the comparison result \( v \), where \( v = 1 \) if \( a \geq b \) and \( v = 0 \), otherwise. We also require \( S_2 \) to learn nothing about the relation between \( a \) and \( b \) but just help \( S_1 \) to obtain an encryption of the comparison result. A private comparison protocol was given by Veugen [45], however in his construction \( S_1 \) finally learns the comparison result. Bost et al. [10] recently presented a protocol based on [45] that suits our setting perfectly: \( S_1 \) only learns the encryption of the comparison result. The basic observation for private comparison is that for any two \( \ell \) bit numbers \( a \) and \( b \), the most significant bit of \( z = 2^{\ell} + a - b \) (\( \ell + 1 \) bits) reveals the relation between \( a \) and \( b \). Specifically, \( z_{\ell} = 1 \Leftrightarrow a \geq b \). The protocol makes use of two semantically secure homomorphic encryption schemes: Paillier [39] and the QR (Quadratic Residuosity) cryptosystem due to Goldwasser and Micali [23] which allows single bit encryption. As noted in Table 1 we refer to encryption of message \( m \) using Paillier and QR as \([m]\) and \([|m|]\), correspondingly.

We now give an overview of this protocol: \( S_2 \) knows the encryption and decryption keys for both Paillier and QR, \((PK_P, SK_P, PK_{QR}, SK_{QR})\), while \( S_1 \) knows the corresponding public keys \((PK_P, PK_{QR})\) and two values \( a \) and \( b \) encrypted under Paillier’s scheme. \( S_1 \) first computes \( z = [a] \cdot [b]^{-1} \cdot [2^\ell] \mod n^2 \) and blinds it with a random value \( r \) before sending it to \( S_2 \) (or else \( S_2 \) would learn the comparison result). \( S_2 \) computes \( d = d \mod 2^\ell \), \( S_1 \) similarly computes \( \tilde{r} = r \mod 2^\ell \) and they engage in a private input comparison protocol (we can use the DGK protocol [17] as suggested by Bost et al. [10]) that compares \( d \) and \( \tilde{r} \). At the end of this protocol, \( S_1 \) receives an encrypted bit \( \lambda \) that shows the relation between \( d \) and \( \tilde{r} \) (\( \lambda = 1 \Leftrightarrow d < \tilde{r} \)). The output \( \lambda \) from the private input
comparison protocol is encrypted using QR scheme. Finally, $S_1$ computes the most significant bit of $z$, denoted by $v$, by computing $||v|| = ||d_r|| \cdot ||r_i|| \cdot ||\lambda||$. The important security property of this protocol is that $S_1$ never sees the comparison result in the clear and $S_2$ never receives an encryption of it.

The above protocol returns as a result bit $v$ encrypted using QR cryptosystem, $||v||$, for which only $S_2$ knows the secret key $SK_{QR}$. However, for the purpose of our sorting task (where we require private comparison and a call to EncSelect) $S_1$ needs to know this bit encrypted using second layer of generalized Paillier cryptosystem, that is, $[v]$. To remedy this we introduce ReEncryptBit protocol to securely re-encrypt the bit $v$ such that neither the $S_1$ nor $S_2$ learns its value. We describe the ReEncryptBit protocol in the paragraph below and present the final comparison protocol EncCompare which is defined as $[v] \leftarrow \text{EncCompare}(PK_P, SK_P, PK_{QR}, SK_{QR}, [a], [b])$ in Table 5. QR keys are omitted from EncCompare protocol when it is called from private sort protocol in Table 8 for simplicity since QR keys are used only in EncCompare protocol.

**Bit Re-encryption** The purpose of our bit re-encryption task is to privately decrypt a bit encrypted using QR scheme and then encrypt it using Paillier cryptosystem without neither $S_1$ nor $S_2$ learning the bit. Recall that in this setting $S_1$ has $||v||$ as a private input and $S_2$ has secret keys from QR and Paillier encryption schemes. The ReEncryptBit protocol, given in Table 6, proceeds as follows:

- $S_1$ computes two values $||v|| \cdot ||0||$ and $||v|| \cdot ||1||$ which are equal to $v \oplus 0$ and $v \oplus 1$, respectively.
- $S_1$ permutes these two values according to a random bit $r$ and sends them to $S_2$ as $||s_0||$ and $||s_1||$.
- $S_2$ decrypts them (always gets a “0” and a “1” in an order that is independent of $t$), re-encrypts them using second layer of Paillier scheme and sends them back to $S_1$ in the same order as he received them.
- Given that $S_1$ knows the permutation, he outputs $[s_r]$ which corresponds to the relation between $a$ and $b$.

Note that $S_1$ and $S_2$ can agree on any other encryption layer for bit $v$, however for the bit to be used in EncSelect we require second layer of generalized Paillier cryptosystem.

We capture privacy of ReEncryptBit using the following definition and prove that ReEncryptBit adheres the definition in Theorem 2.

**Definition 2.** Let $II_{\text{ReEncryptBits}}$ be a two party protocol for computing ReEncryptBit functionality. $S_1$ takes as input $(PK_P, PK_{QR}, ||v||)$ and $S_2$ takes as input $(PK_P, SK_P, PK_{QR}, SK_{QR})$, where $v$ is a bit. When $II_{\text{ReEncryptBits}}$ terminates $S_1$ receives the output $[v]$ of ReEncryptBit. Let VIEW$_{SI}$ ReEncryptBit $(PK_P, SK_P, PK_{QR}, SK_{QR}, ||v||)$ be all the messages that $S_i$ receives while running the protocol on inputs $PK_P, SK_P, PK_{QR}, SK_{QR}, ||v||$ and OUTPUT$II_{\text{ReEncryptBits}}$ be the output of the protocol received by $S_1$. 

---

Table 5: $[v] \leftarrow \text{EncCompare}(PK_P, SK_P, PK_{QR}, SK_{QR}, [a], [b])$: Interactive Private Comparison between two parties $S_1$ and $S_2$ such that only $S_1$ learns an encryption of the comparison bit $[v]$. For simplicity, QR keys are omitted when EncCompare is called from private sort protocol in Table 8. This protocol is an adaptation of the comparison protocol from [10].

<table>
<thead>
<tr>
<th>$S_1$($PK_P, PK_{QR}, [a], [b]$)</th>
<th>$S_2$($PK_P, SK_P, PK_{QR}, SK_{QR}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[z] := [2^\ell] \cdot [a] \cdot [b]^{-1} \mod n^2$</td>
<td>decrypt $[d]$</td>
</tr>
<tr>
<td>pick $r \in {0,1}^{\ell+i}$, $[d] := [z] \cdot [r] \mod n^2$</td>
<td>$\tilde{d} := d \mod 2^\ell$</td>
</tr>
<tr>
<td>$r := r \mod 2^\ell$</td>
<td>$</td>
</tr>
<tr>
<td>$</td>
<td></td>
</tr>
<tr>
<td>$[v] \leftarrow \text{ReEncryptBit}(PK_P, SK_P, PK_{QR}, SK_{QR},</td>
<td></td>
</tr>
</tbody>
</table>
Table 6: \( v \leftarrow \text{ReEncryptBit}(PK_p, SK_p, PK_{QR}, SK_{QR}, v) \): Interactive protocol between \( S_1 \) and \( S_2 \) for converting \( v \) to \( [v] \) where \( v \) is a bit.

<table>
<thead>
<tr>
<th>( S_1(PK_p, PK_{QR}, [v]) )</th>
<th>( S_2(PK_p, SK_p, PK_{QR}, SK_{QR}) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \langle s_r \rangle := [v] \cdot [0] )</td>
<td>decrypt ( |s_0|, |s_1| )</td>
</tr>
<tr>
<td>( |s_1 - r| := [v] \cdot [1] )</td>
<td>encrypt ( |s_0|, |s_1| )</td>
</tr>
<tr>
<td>( [v] := [s_r] )</td>
<td>( |s_0|, |s_1| )</td>
</tr>
</tbody>
</table>

We say that \( \Pi_{\text{ReEncryptBit}} \) privately computes \( \text{ReEncryptBit} \) if there exists a pair of probabilistic polynomial time (PPT) simulators \( (\text{Sim}_{S_1}, \text{Sim}_{S_2}) \) such that

\[
(1) \quad (\text{Sim}_{S_1}(PK_p, PK_{QR}, [v]), \text{ReEncryptBit}(K_p, K_{QR}, [v])) \cong (\text{VIEW}_{S_1}^{\Pi_{\text{ReEncryptBit}}}(K_p, K_{QR}, [t]), \text{OUTPUT}_{S_1}^{\Pi_{\text{ReEncryptBit}}}(K_p, K_{QR}, [t]));
\]

\[
(2) \quad \text{Sim}_{S_1}(K_p, K_{QR}) \cong \text{VIEW}_{S_2}^{\Pi_{\text{ReEncryptBit}}}(K_p, K_{QR}, [v]),
\]

where \( K_p \) and \( K_{QR} \) denote the key pairs \( (PK_p, SK_p) \) and \( (PK_{QR}, SK_{QR}) \), correspondingly.

**Theorem 2.** The protocol in Table 6 privately computes \( \text{ReEncryptBit} \) functionality according to Definition 2.

**Proof.** (Sketch) We construct \( \text{Sim}_{S_1} \) as follows. \( \text{Sim}_{S_1} \), as \( S_1 \), has access to inputs \( PK_p, PK_{QR}, [v] \). When interacting with \( S_2 \), \( S_1 \) receives the following messages \( [s_0] \) and \( [s_1] \). \( \text{Sim}_{S_1} \) can easily simulate them using \( [r_0] \) and \( [r_1] \) where \( r_0, r_1 \) are random bits. \( S_1 \) cannot distinguish \( [r_0], [r_1] \) from \( [s_0], [s_1] \) due to the properties of semantic security that comes from generalized Paillier encryption.

\( \text{Sim}_{S_1} \), similar to \( S_2 \), has access to public and secret keys from Paillier and QR cryptosystems and, hence, can decrypt any messages he receives. \( S_2 \)'s \text{VIEW} consists of messages \( [s_0] \) and \( [s_1] \) while participating in \( \text{ReEncryptBit} \). Note that \( (s_0, s_1) \) equals \( (0, 1) \) or \( (1, 0) \) by construction since \( S_1 \) performs a homomorphic XOR of bit \( v \) with 0 and 1. Moreover, the order in which encryptions of \( s_0 \) and \( s_1 \) are independent of XOR of \( v \) with 0,1 bits since \( S_1 \) sends them permuted according to his secret bit \( r \) and, hence, is independent of \( v \). \( \text{Sim}_{S_1} \) can easily simulate this behavior by picking a random bit \( r' \) and substituting \( [s_0], [s_1] \) with \( [r'], [1 - r'] \) and construct the view that is indistinguishable from \( S_2 \)'s \text{VIEW}.

We capture privacy of \( \text{EncCompare} \) below.

**Definition 3.** Let \( \Pi_{\text{EncCompare}} \) be a two party protocol for computing \( \text{EncCompare} \) functionality. \( S_1 \) takes as input \( (PK_p, PK_{QR}, [a], [b]) \) and \( S_2 \) takes as input \( (PK_p, SK_p, PK_{QR}, SK_{QR}) \). When \( \Pi_{\text{EncCompare}} \) terminates \( S_1 \) receives the output \( [v] \) of \( \text{EncCompare} \). Let \( \text{VIEW}_{S_1}^{\Pi_{\text{EncCompare}}}(PK_p, SK_p, PK_{QR}, SK_{QR}, [a], [b]) \) be all the messages that \( S_i \) receives while running the protocol on inputs \( PK_p, SK_p, PK_{QR}, SK_{QR}, [a], [b] \) and \( \text{OUTPUT}_{\Pi_{\text{EncCompare}}} \) be the output of the protocol received by \( S_1 \).

We say that \( \Pi_{\text{EncCompare}} \) privately computes \( \text{EncCompare} \) if there exists a pair of probabilistic polynomial time (PPT) simulators \( (\text{Sim}_{S_1}, \text{Sim}_{S_2}) \) such that

\[
(1) \quad (\text{Sim}_{S_1}(PK_p, PK_{QR}, [a], [b]), \text{EncCompare}(K_p, K_{QR}, [a], [b])) \cong (\text{VIEW}_{S_1}^{\Pi_{\text{EncCompare}}}(K_p, K_{QR}, [a], [b]), \text{OUTPUT}_{S_1}^{\Pi_{\text{EncCompare}}}(K_p, K_{QR}, [a], [b]));
\]

\[
(2) \quad \text{Sim}_{S_1}(K_p, K_{QR}) \cong \text{VIEW}_{S_2}^{\Pi_{\text{EncCompare}}}(K_p, K_{QR}, [a], [b]),
\]

where \( K_p \) and \( K_{QR} \) denote the key pairs \( (PK_p, SK_p) \) and \( (PK_{QR}, SK_{QR}) \), correspondingly.

**Theorem 3.** The protocol in Table 5 privately computes \( \text{EncCompare} \) functionality according to Definition 2.

**Proof.** (Sketch) We can construct \( \text{Sim}_{S_1} \) and \( \text{Sim}_{S_2} \) by using corresponding simulators from [10] to simulate all but first line of \( \text{EncCompare} \) in Table 6, and then use corresponding simulators from Definition 2 to simulate the behavior of \( \text{ReEncryptBit} \).
2.4 Text Search and Ranking

We represent a document collection using an inverted index \[48\]. Each unique term, or keyword, \( t \) appearing in the collection is associated with a set of document ids \( J_t \), where each document id \( d \in J_t \) corresponds to a document containing \( t \). We refer to \( J_t \) as a posting list of term \( t \).

We consider free text queries \[35\], a common type of queries supported by today’s search engines. A free text query \( q \) is a set of terms and the result to \( q \) is a set of documents \( J_q \) that contain at least one of the terms in \( q \). We can define \( J_q \) in terms of posting lists as

\[
J_q = \bigcup_{t \in q} J_t.
\]

In this paper we use a common ranking of search results based on frequency of query terms in each document and the collection, namely tf-idf \[48\]. Let \( N \) be the number of documents in the collection and \( cf_t \) be the frequency of term \( t \) in the collection then inverse document frequency, idf, is defined as:

\[
idf_t = \log \frac{N}{cf_t}.
\]

Document frequency of term \( t \) in document \( d \) is defined as:

\[
\text{tf-idf}_{t,d} = \text{tf}_{t,d} \times idf_t
\]

where \( \text{tf}_{t,d} \) is frequency of term \( t \) in document \( d \). If a document \( d \) does not contain \( t \) we set \( \text{tf-idf}_{t,d} \) to zero.

Given a free text query \( q \) for each document \( d \in J_q \) a score based on frequencies is computed as

\[
\text{score}(q,d) = \sum_{t \in q} \text{tf-idf}_{t,d}
\]

(1)

Documents in \( J_q \) can then be sorted according to the output of the score function. We use \( F \) to denote the frequency table of all \( \text{tf-idf}_{t,d} \) entries including zero entries for a document \( d \) that does not contain a term \( t \).

Finally, we note that a conjunctive query \( q = \bigwedge_{t \in q} t \), i.e., query that returns only documents \( J_q \) that contain all the query keywords, can be expressed using the above notation as follows:

\[
J_q = \bigcap_{t \in q} J_t.
\]

2.5 Searchable Encryption of Structured Data

Symmetric searchable encryption (SSE) allows a user that has in his possession a collection of documents \( D = \{D_1, \ldots, D_N\} \) to compute a “secure search index”, \( I \), on them and then outsource the index to the server\(^6\). The server should be able to search on \( I \) but without learning anything about the actual collection \( D \). In traditional definitions of searchable encryption the user gives as input the actual document collection and his SSE secret key and receives back a secure index \( I \) and a set of ciphertexts \[15\]. However, for our construction of private ranked search in Section 4 we need a generalized version of the SSE definition, where the user uses SSE to encrypt a dictionary and some ranking meta-data instead of the document collection itself.

Such a generalized notion of SSE was given by Chase and Kamara \[14\] who introduced structured encryption (StE) that allows SSE for arbitrarily-structured data. Specifically, they consider static data types: collections of objects that support query operations on them. More formally, a data type \( T \) is defined by a universe \( \mathcal{U} = \{U_k\}_{k \in \mathbb{Z}} \) and an operation \text{Query}: \( \mathcal{U} \times \mathcal{Q} \rightarrow \mathcal{R} \) with \( \mathcal{Q} = \{Q_k\}_{k \in \mathbb{N}} \) being the query space and \( \mathcal{R} = \{R_k\}_{k \in \mathbb{N}} \) being the response space, where \( k \) is the security parameter. Below, we present the “structure-only” variant of the StE definition (in the original definition the encryption of semi-private data together with the data structure is also considered). This scheme consists of the following algorithms \[14\]:

\[
\text{SK}_{\text{StE}} \leftarrow \text{Gen}_{\text{StE}}(1^k) \text{ run by the owner of the data. The output is owner’s secret key } \text{SK}_{\text{StE}} \text{ for the security parameter } k.
\]

\(^6\) One can think of the “secure index” as a standard search index encrypted under the data owner’s encryption key.
A fresh randomness is used during re-encryption. We note that Definition 4 can be easily expanded to take as key pair $(PK, SK)$. In Definition 5, a data structure on a data type is run by the owner to generate a trapdoor for a query $t \in Q$. It outputs a trapdoor $T$ or the failure symbol $\bot$. $a \leftarrow \text{Search}(I, T)$ is run by the server $S$ to perform a search for a trapdoor $T$ and outputs an answer $a \in R$.

The SSE definition due to Curtmola et al. [15] is a special case of the definition above for the scenario of private keyword search over encrypted data. In [15] the input to $E_{\text{SSE}}$ is the whole document collection $D$ and the output is the secure index $I$ together with a sequence of ciphertexts $c$. To achieve this, constructions of [15] first build a data structure on $D$ that allows efficient search queries and then encrypt this data structure to get $I$.

The construction of ranked search presented in Section 4.2 and Section 5 use a dictionary data type for StE. In particular, in both the keys (or queries) of a dictionary are keywords of the document collection $D$. The value (or a response) that corresponds to a particular keyword in the dictionary is a sequence of pairs of document ids and encrypted frequency scores. The number of such pairs per keyword differs between the two constructions: the scheme in Section 4.2 has a pair for every document identifier in the collection, while the first scheme of Section 5 stores only pairs for document ids in which the keyword appears in.

The main security guarantee of traditional SSE is that the server does not learn anything about the documents and the queries beyond what can be inferred from the access pattern (e.g., co-occurrence of document ids in answers to queries) and search pattern (e.g., co-occurrence of keywords in queries). Similarly, the StE security definition formalizes access and search patterns as the “leakage” that is observed in the system and refers to them as intersection and query pattern, respectively.

3 Private Sort

In this section we define a new tool for secure outsourced computation: a private sort, or private outsourced sort, protocol and present its efficient construction. As mentioned above we believe that our primitive is of independent interest since it privately computes, arguably, one of the most used functionalities over data, sorting.

3.1 Model

Private sort\(^7\) is executed between two parties $S_1$ and $S_2$ as follows. $S_1$ has an array $A$ encrypted using a secret key $SK$ that is known to $S_2$ but not $S_1$. The goal of private sort is for $S_1$ to obtain $B$, a re-encryption of a sorted array $A$, such that neither $S_1$ nor $S_2$ learn anything about the plaintext values of $A$ (e.g., their initial order, frequency of the values) while running the protocol. We consider the honest-but-curious model: our servers honestly follow the protocol but might try to analyze the protocol transcript to infer more information about the data in the array.

We formally capture the definition of private sort below.

Definition 4. (EncSort) An encrypted sorting functionality $\text{EncSort}(PK, SK, A)$ takes as input a public/secret key pair $(PK, SK)$ of a semantically secure cryptosystem $\{\text{GenSS}, E_{\text{SS}}, D_{\text{SS}}\}$, and an array $A = [E_{\text{SS}}(v_i)]_{i \in \{1, N\}}$ of $N$ elements where each element is encrypted individually using $PK$. Let $\pi$ be a permutation of indices 1 to $N$ that corresponds to the indices of $A$’s elements sorted using its unencrypted values $v_1$. Then, the output of $\text{EncSort}$ is an array $B = [E_{\text{SS}}(v'_j)]_{j \in \{1, N\}}$ where $v'_j = v_{\pi(i)}$ and $i \in \{1, N\}$.

In the definition above, though $v'_j = v_{\pi(i)}$, it holds with very high probability that $E_{\text{SS}}(v'_j) \neq E_{\text{SS}}(v_{\pi(i)})$ since fresh randomness is used during re-encryption. We note that Definition 4 can be easily expanded to take as input an array $A$ that stores (key, value) pairs and the output is required to be sorted using values.

We describe the privacy property of the encrypted sorting functionality stated above using the paradigm for defining privacy in the semi-honest model given by Goldreich [21].

Definition 5. (EncSort Privacy) Let $\Pi_{\text{EncSort}}$ be a two party protocol for computing $\text{EncSort}$ functionality. $S_1$ takes as input $(PK, A)$ and $S_2$ takes as input $(PK, SK)$. When $\Pi_{\text{EncSort}}$ terminates $S_1$ receives the output $B$ of $\text{EncSort}$. Let $\text{VIEW}^\Pi_{S_1}(PK, SK, A)$ be all the messages that $S_1$ receives while running the protocol on inputs $PK, SK, A$ and $\text{OUTPUT}^\Pi_{S_1}$ be the output of the protocol received by $S_1$.

\(^7\) We note that one should not confuse our problem with Multi-Party Computation protocols for sorting [26, 28], where every party has an input array and the goal is to output to every participating party the sorting of all inputs combined.
We say that $\Pi_{\text{EncSort}}$ privately computes $\text{EncSort}$, i.e., $\Pi_{\text{EncSort}}$ is a private outsourced sort, if there exists a pair of probabilistic polynomial time (PPT) simulators $(\text{Sim}_{S_1}, \text{Sim}_{S_2})$ such that

\((1) (\text{Sim}_{S_2}(PK, A), \text{EncSort}(PK, SK, A)) \cong \left(\text{VIEW}_{S_1}^{\Pi_{\text{EncSort}}}(PK, SK, A), \text{OUTPUT}^{\Pi_{\text{EncSort}}}(PK, SK, A)\right);
\)

\((2) \text{Sim}_{S_1}(PK, SK, N) \cong \text{VIEW}_{S_2}^{\Pi_{\text{EncSort}}}(PK, SK, A),\)

where $N$ is the size of the array $A$ and $\cong$ denotes computational indistinguishability for all tuples $PK, SK, A$.

The intuition behind the privacy definition of $\text{EncSort}$ is as follows. $S_1$ has an array $A$ encrypted using a semantically secure encryption and by the end of the protocol he receives an array $B$ which contains the values of $A$ sorted and encrypted using fresh randomness, i.e., a property of semantic security. $S_2$ has the corresponding secret key $SK$ and receives nothing as an output. $\text{VIEW}_{S_1}$ captures messages that $S_1$ receives while participating in $\Pi_{\text{EncSort}}$. In order to capture that $S_1$ does not learn anything about $SK$, and plaintext of $A$ or $B$ as a consequence, one has to show that there exists a simulator of $S_2$, $\text{Sim}_{S_2}$. $\text{Sim}_{S_2}$ knows exactly what is known to $S_1$ and nothing more. The main property of $\text{Sim}_{S_2}$ is that $S_1$ should not be able to distinguish if he is interacting with $\text{Sim}_{S_2}$ or with $S_2$ who knows the secret key of the encryption scheme. Hence, $S_1$ learns nothing more than he knew already. The privacy guarantee for $S_1$ is similar. One shows that there is a simulator $\text{Sim}_{S_1}$ that knows the key pair of the cryptosystem and only the size of $A$.

### 3.2 Construction

In this section we develop a construction for the private sort functionality $\text{EncSort}(PK, SK, A)$ presented in Definition 4. From now on we assume that the array $A$ is encrypted using the first layer of Paillier cryptosystem (Section 2.1), however, the system can be adapted to higher levels with corresponding adjustment to the protocols.

Our private sort protocol relies on (a) homomorphic properties of the generalized Paillier cryptosystem from Section 2.1 to allow $S_1$ and $S_2$ to privately compare and swap pairs of ciphertexts, and (b) a data independent sorting network, Batcher’s sort [5], which allows to sort the data such that comparisons alone do not reveal the order of the encrypted elements. We first describe a protocol for sorting just two elements and then use it as a blackbox for general sorting. Finally, we show how to extend the protocol to sort an array where an element is not a single ciphertext value but a (key, value) pair where key and value are individually encrypted and sorting has to be performed on value.

**Two Element Sort** We develop a protocol between two parties $S_1$ and $S_2$ to blindly sort two encrypted values. In particular, $S_1$ possesses encryptions of $a$ and $b$, $[a]$ and $[b]$, while $S_2$ has the corresponding decryption key $SK_P$. $S_1$ and $S_2$ engage in an interactive protocol, $\text{EncPairSort}$, by the end of which $S_1$ has a pair of values $([c], [d])$ such that $c = a < b$ and $d = (b, a)$, otherwise. Informally, $\text{EncPairSort}$ has the following privacy guarantees. $S_1$ nor $S_2$ should learn nothing about values $a$ and $b$ nor their sorted order. The formal definition of $\text{EncPairSort}$ is a special case of $\text{EncSort}$ in Definition 4 with $N = 2$.

The $\text{EncPairSort}$ makes use of the comparison protocol $\text{EncCompare}$ from Section 2.3 to help $S_1$ to acquire an encryption of the bit $v$ that denotes whether $a \geq b$ or not. QR cryptosystem. Given a Paillier encryption of $v$ we can then use a ciphertext selection $\text{EncSelect}$ from Section 2.1 to blindly swap $a$ and $b$ according to $v$, i.e., their sorted order. The complete protocol $\text{EncPairSort}$ is shown in Table 7. To simplify the notation in Table 7 we denote public and secret key pair for Paillier cryptosystem as $K_P = (PK_P, SK_P)$. We omit the QR keys when calling $\text{EncCompare}$ protocol to simplify notation.

**Theorem 4.** The $\text{EncPairSort}$ protocol in Table 7 is a private outsourced sorting protocol according to Definition 5 for the case $N = 2$.

**Proof.** (Sketch) In order to show that $\text{EncPairSort}$ in Table 7 is secure according to Definition 5 we need to construct two simulators $\text{Sim}_{S_1}$ and $\text{Sim}_{S_2}$ that show that behavior of $S_1$ and $S_2$ can be simulated without their corresponding private inputs and hence cannot reveal any information about these inputs to $S_2$ and $S_1$, correspondingly.

We construct $\text{Sim}_{S_2}$ as follows. $\text{Sim}_{S_2}$ has access to private inputs of $S_1$ in the protocol. The $\text{VIEW}$ of $S_1$ consists of $\text{VIEW}$’s from $\text{EncCompare}$ and two invocations of $\text{EncSelect}$ protocols. In Theorems 1 and 3 we showed
that there exist simulators for each of these functionalities. Let us refer to Sim_{S_2} from Theorem 1 as Sim_{EncSelect}^{EncCompare}, and to Sim_{S_2} from Theorem 3 as Sim_{S_2}^{EncSelect}. Then Sim_{S_2}^{EncPairSort} simply invokes Sim_{S_2}^{EncSelect} and Sim_{S_2}^{EncSelect} twice. The construction of Sim_{S_1} is symmetrical and uses Sim_{S_1}’s from Theorems 1 and 3.

**General Sort** In the previous section we developed an interactive method EncPairSort for blindly sorting two elements (Table 7). In this section, we use EncPairSort as a blackbox to build a protocol EncSort for privately sorting N elements according to Definition 5. Recall that EncSort is an interactive protocol between S_1 and S_2. S_1 has an encrypted array A that he wishes to sort and S_2 has a secret key of the underlying encryption scheme. In the end of the protocol, S_1 obtains a re-encryption of his array A with S_2’s help while neither of them learn anything about A nor its sorting.

Privacy properties of two element sorting EncPairSort guarantee that S_1 does not learn the result of the comparison of two encrypted elements nor anything about the elements being compared. Hence, sorting algorithms that make calls to a comparison function depending on the data are not applicable in our scheme (e.g., quick sort performs a different sequence of comparisons depending on the layout of the data it is sorting, giving O(N log N) comparisons on average). For our purposes we require a sorting network that performs comparisons in a data-independent manner and guarantees that after performing a deterministic sequence of comparisons the result is sorted. We pick Batcher’s sorting [5] for our purposes. Even though asymptotically AKS [3] is more efficient, it has high hidden constants that in practice make it inferior to Batcher’s sorting network.

Batcher’s sorting network sorts an array of N elements using O(N log^2 N) data independent calls to a comparator function (i.e., the number of rounds is the same for a fixed N independent of the data). One can view the network in O((log N)^2) consecutive levels where O(N) pairs of elements are compared and swapped at every level. In particular, let A_i be an array of elements at ith level such that A_i is the input array, where A_i[1] denotes the jth element of array A_i. Each level i takes as input array A_i and produces A_{i+1} where the pairs scheduled to be sorted at level i are in sorted order in A_{i+1}. For example, A_2[0] and A_2[1] contain A_2[0], A_2[1] in sorted order, A_2[2] and A_2[3] contain A_2[2], A_2[3] in sorted order and so on. We use pairs_i to denote an iterator over pairs that need to be sorted in the ith level and pairs_{i+1} next returns the next pair to be sorted.

In Table 8 we present our protocol EncSort where S_1 performs Batcher’s sorting network using S_2 to help him sort the elements of pairs at every level of the network. To sort every pair, S_1 and S_2 run EncPairSort. Recall that the output of EncPairSort is encrypted using the first layer of Paillier cryptosystem, hence, the result of pairwise sorting at level i can be used as input for calls to EncPairSort in the next level i + 1. (See Figure 2 for an illustration of EncSort on an example array of size 4.) Recall that S_1 and S_2 are two non-colluding honest but curious adversaries and hence will execute their side of the protocol faithfully.

**Theorem 5 (EncSort Privacy).** The EncSort protocol in Table 8 is a private outsourced sorting protocol according to Definition 5.

**Proof.** (Sketch) The protocol EncSort in Table 8 makes O(N log^2 N) calls to EncPairSort protocol in Table 7. In Theorem 4 we showed that there exist simulators Sim_{S_1} and Sim_{S_2} for EncPairSort. Hence, the simulators for EncSort can be trivially constructed by calling corresponding simulators of EncPairSort.

**Theorem 6 (EncSort Performance).** The EncSort protocol in Table 8 has the following performance guarantees:

- The storage requirement of S_1 is O(N);
- The total computation required by S_1 and S_2 is O(N log^2 N);
- The communication complexity between S_1 and S_2 consists of O(N log^2 N) rounds;
Table 8: \( B \leftarrow \text{EncSort}(PK_p, SK_p, A) \): Interactive protocol between \( S_1 \) and \( S_2 \) for privately sorting an array \( A \) of \( N \) elements encrypted using Paillier encryption such that only \( S_1 \) acquires the sorted result \( B \) (see Definition 5). Paillier key pair is denoted using \( K_p = (PK_p, SK_p) \). See Figure 2 for an illustration for the case when \( N = 4 \).

\[
\begin{align*}
\text{S}_1(PK_p, A) & \quad \text{S}_2(PK_p, SK_p) \\
A_1 \leftarrow A & \quad \% k = O((\log N)^2), N = |A| \\
\text{for } i \in \{1, \ldots, k - 1\} & \quad \% \text{i}th \text{ level of Batcher’s sort} \\
(x, y) \leftarrow \text{pairs} \_\text{next} & \quad \% \text{Sort } x, y \text{ entries of } A_i \\
\text{while } ((x, y) \neq \bot) & \quad \% \text{Next pair of } A_i \text{ to sort} \\
(A_{i+1}[x], A_{i+1}[y]) \leftarrow \text{EncPairSort}(PK_p, A_i[x], A_i[y]) & \\
(x, y) \leftarrow \text{pairs} \_\text{next} & \\
B \leftarrow A_k & \\
\end{align*}
\]

\( \text{A}_{1} \) \[
\begin{bmatrix}
(5) \\
(1) \\
(2) \\
(9)
\end{bmatrix}
\]
\( \text{A}_{2} \) \[
\begin{bmatrix}
(1) \\
(2) \\
(9)
\end{bmatrix}
\]
\( \text{A}_{3} \) \[
\begin{bmatrix}
(1) \\
(2) \\
(9)
\end{bmatrix}
\]
\( \text{A}_{4} \) \[
\begin{bmatrix}
(1) \\
(2) \\
(9)
\end{bmatrix}
\]

\begin{align*}
\text{pairs}_1 = \{(1,2), \{3,4\}\} & \\
\text{pairs}_2 = \{(1,3), \{2,4\}\} & \\
\text{pairs}_3 = \{(2,3)\} &
\end{align*}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{Fig2-EncSort.png}
\caption{Example of \texttt{EncSort} protocol in Table 8 for sorting an array of four elements \(5, 1, 2, 9\) where \([m]\) denotes a Paillier encryption of message \( m \). Note that only \( S_1 \) stores values in the arrays \( A_i \) while \( S_2 \) blindly assists \( S_1 \) in sorting the values.}
\end{figure}

- \( S_2 \) has \( O(1) \) storage, the time overhead of the protocol is \( O(N(\log N)^2) \);
- \( S_2 \) has \( O(N) \) storage, the time overhead of the protocol is \( O((\log N)^2) \).

Proof. (Sketch) \( S_1 \) needs to have \( O(N) \) storage space in order to store the original array \( A \) of size \( N \) along with the intermediate sorting results. The intermediate storage is at most two arrays \( A_1 \) and \( A_{i+1} \) since after finishing the \( i \)th level of sorting \( S_1 \) can safely discard array \( A_i \). \( S_2 \), on the other hand, is only required to store the keys of the encryption schemes used and perform field arithmetic to run encryption and decryption algorithms on constant number of elements.

The protocol requires \( O(N(\log N)^2) \) roundtrips between \( S_1 \) and \( S_2 \) where \( S_1 \) and \( S_2 \) perform a constant computation after every round.

If \( S_2 \) has \( O(N) \) memory then a highly parallelizable nature of the Batcher’s sorting network can be exploited. It allows all invocations of \texttt{EncPairSort} during a single round \( i \) to be run in parallel since they operate on different pairs of the array \( A_i \).

Key-Value Sort In the previous section we described how to sort an array where every element of an array is an encrypted plaintext used for comparison. However, the protocol is easily expandable to work on arrays where every element is a pair of ciphertexts representing a \((\text{key, value})\) pair and \text{\texttt{value}}\ is used to sort the array. The main alternations happen in \texttt{EncPairSort} protocol where the input is not two ciphertexts as in Table 7 but two pairs of ciphertexts: \((k_1, [v_1])\) and \((k_2, [v_2])\), and similarly in the output. Since comparison is performed only on \texttt{values \texttt{EncCompare} \ is called only on \([v_1]\) and \([v_2]\). Once the \texttt{bit} representing the result of the comparison is computed, \texttt{EncSelect} is used not only on the ciphertexts of the values but also on the keys. That is, if values have to be swapped so do their corresponding keys. This functionality is used in Section 4.2 when sorting document identifiers using their query score where the \texttt{key} is an encrypted document id and \texttt{value} is an encryption of the corresponding score.
4 Private Ranked Search (MRSE)

We now describe our Multi-keyword Ranked Searchable Encryption (MRSE) framework that allows an owner of a data collection to outsource his documents to a server $S_1$ and then search on them and receive ranked results. $S_1$ is able to sort the results by running the private sort (Section 3) with server $S_2$.

4.1 MRSE Security Model

MRSE is essentially an augmented structured searchable encryption system that also supports ranking: MRSE consists of $Gen_{SE}$, $ES_{SE}$, $Trapdr$, $Search$, as defined in Section 2.5, plus a ranking mechanism. To describe the security properties of MRSE we start by defining the ranking mechanism.

We assume ranking mechanisms that can be described by two algorithms, $PrepareRankStruct$ and $RankSortFunc$. $PrepareRankStruct$ takes as input the document collection $D$ and returns a score data structure $ScT$ that allows one to evaluate a score for every document given a set of keywords for a specific ranking method. Then, $RankSortFunc$, given as input the result of $Search$ and the secure index $I$ (which is the output of $ES_{SE}$ on input $ScT$), returns the result sorted according to the ranking.

Definition 6. (Ranking mechanism: $PrepareRankStruct$, $RankSortFunc$) Let $D$ be a document collection and $(PK, SK)$ be a public/secret key pair of a semantically secure cryptosystem $\{Gen_{SS}, E_{SS}, D_{SS}\}$. A ranking mechanism is represented by $PrepareRankStruct$ and $RankSortFunc$ as follows. $PrepareRankStruct$ takes as input $(PK, SK)$, and $D$, and returns the score data structure $ScT$. $A$ rank and sort functionality $RankSortFunc$ takes as input $(PK, SK)$, a set of keyword tags $X$ and the encrypted secure (score) data structure $I$. The outputs of $RankSortFunc$ are: (1) $B$, the result of a query consisting of keywords with tags in $X$ ranked and sorted according to the underlying ranking mechanism; (2) $\text{VIEW}_{S_1}$, the view of an involved party $S_1$.

In searchable encryption, a server, given an encrypted document collection or encrypted data structure and the corresponding secure search index, while answering multiple search queries, should not be able to deduce anything regarding the data collection or the corresponding search index apart from the access and search patterns. By the term access pattern we denote the identifiers of the documents that contain a keyword\(^8\), while the search pattern refers to whether the search was for the same term or not. Our MRSE setting though consists of two servers $S_1$ and $S_2$. The user queries only $S_1$ which is required to return the document id’s that match the search query sorted by relevance criteria (i.e., tf-idf) by running private sort with $S_2$. Thus, we examine the privacy of the client against $S_1$ and $S_2$. This is sufficient for the overall privacy given that $S_1$ and $S_2$ are non-colluding and the privacy of the interaction between them is limited to using private sort as specified in Definition 5.

Informally, MRSE privacy definitions (Definitions 9 and 10 below) capture the following: $S_1$ learns the number of documents and unique keywords in the collection (he can infer this from the size of the encrypted index), as well as the search pattern of client queries since he observes the “encrypted” queries of the user. However, $S_1$ learns nothing about the access pattern. $S_2$, on the other hand, only learns the number of documents in the collection and knows when the client is querying the system. However, $S_2$ learns nothing about the access and search patterns, and hence does not know anything about the content of the queries. We note that $S_1$’s capabilities are similar to those of the server in the original SSE definition [15] while $S_2$ learns much less.

Let $\Delta = (t_1, \ldots, t_M)$ denote a dictionary of $M$ unique keywords in lexicographic order, and $2^\Delta$ denote the set of all possible documents with words in $\Delta$. The $i$th search query is defined as $q_i = (t_{i,1}, \ldots, t_{i,l_i})$, $t_{i,j} \in \Delta$ and $l_i \leq M$ denotes the number of terms on the query. Without loss of generality we assume that all terms in a search query are unique. Recall that for every term the client wishes to query, SSE generates a trapdoor. Since we allow multi-term queries, we use $T_1$ to denote the sequence of trapdoors that corresponds to the terms in the $i$th query. By $\sigma(H)$ we denote the search pattern induced by an $n_q$-query history $H$ and by $\tau(H) = (N, M, \sigma(H))$ we denote corresponding $\text{trace}$, that is allowed to be leaked during searching and is related to the search pattern. The interaction between the server $S_1$ and the client is captured under the notion of history and trace.

Definition 7. (Trace [15]) Let $\Delta$ be a dictionary and $D \subseteq 2^\Delta$ be a document collection over $\Delta$. Let the $n_q$-query history over $D$ be a tuple $H = (D, Q)$ where $Q$ is the set of $n_q$ multi-term queries, i.e., $Q = (q_1, \ldots, q_{n_q})$. The $\text{trace}$ of $\sigma(H)$ is $\tau(H) = (N, M, \sigma(H))$. The $\text{history}$ of $\sigma(H)$ is $\text{HIST}(\sigma(H)) = \cup_{i=1}^{n_q} \sigma(q_i)$.

\(^8\) It would be possible to hide the access pattern by utilizing Oblivious RAM techniques [22]. However, this would lead to a significantly less efficient protocol and it is out of the scope of this work.
trace induced by an \( n_q \)-query history \( H \) is the total number of documents in the collection \( N \), the total number of unique terms \( M \) and the search pattern \( \sigma(H) \), i.e., \( \tau(H) = (N, M, \sigma(H)) \).

We extend the definition of the search pattern \( \sigma(H) \) given in [15] in order to capture the search pattern of multi-keyword queries.

**Definition 8.** (Search Pattern) Let \( \Delta \) be a dictionary and \( D \subseteq 2^\Delta \) a document collection over \( \Delta \). Let \( l_{max} \) be the length of the longest query in the set of queries \( Q \). The search pattern induced by an \( n_q \)-query history \( H = (D, Q) \) is a binary, four dimensional matrix \( \sigma(H) \) of size \( n_q \times n_q \times l_{max} \times l_{max} \) such that the element in position \( i, j, y, x \) is 1 if in \( q \), the \( y \)th term was the same as \( x \)th term of query \( q \) and 0 otherwise.

Note that we omitted the lengths of the documents from the definition of the trace since our construction does not reveal document sizes and behaves the same for all collections that contain the same total number of documents \( N \) and same number of unique terms \( M \).

We are now ready to present our privacy definitions for each MRSE server. Our definitions are inspired by the game based indistinguishability definition of searchable encryption due to Curtmola et al. [15] for adaptive adversaries. Note that adaptivity is more natural for searchable encryption and moreover, as shown in the same work the corresponding simulation based definition is equivalent. Remember that the definitions in [15] are a special case of [14] for document collections instead of data structures – for our model where we want to capture security against the ranking mechanism as well which takes as input the whole document collection the definitions of [15] are more compatible as a starting point.

As mentioned before, our MRSE definitions extend the SSE model in order to support ranking and sorting by making use of the PrepareRankStruct and RankSortFunc functionalities. If RankSortFunc can be executed by a single party then our \( S_1 \) privacy definition below would be sufficient to describe the privacy of the resulting framework against this party. Finally note that, the MRSE model assumes that it is sufficient to return to the client only document identifiers and not the actual documents. This is consistent with related work on searchable encryption with ranking, e.g., [11] (see discussion in Section 6).

**\( S_1 \) Privacy Definition** We now define privacy against an adaptive adversary that acts as a curious server \( S_1 \). The client flips a secret coin \( b \) that the adversary has to guess. The adversary (acting like a malicious \( S_1 \)) submits two document collections \( (D_0, D_1) \) that have the same number of documents \( N \) and same number of unique terms \( M \). \( A \) receives the index of one of the collections \( D_b \) along with the secure score data structure ScT and is allowed to query the system polynomially many times \( n_q \). At each round \( i \) he picks two queries \( q^{(i)}_0, q^{(i)}_1 \) with the same number of terms, \( |q^{(i)}_0| = |q^{(i)}_1| \), and receives trapdoors \( T_b^{(i)} \) for \( q_b \). Then, he uses the StE Search algorithm to find the entries in the search index that match the query. Given these entries he invokes RankSortFunc functionality and receives its output \( B \) as well as the view of the first server VIEW\( S_1 \). \( S_1 \) wins if he manages to correctly guess the bit \( b \).

In the experiments below, running the SSE Trpdr algorithm on a multi-term query is equivalent to running the algorithm for each term separately.

**Definition 9.** (Adaptive indist. for \( S_1 \)) Let \( StE = \{ \text{GenStE}, \text{EStE}, \text{Trpdr}, \text{Search.} \} \) be an StE scheme with security parameter \( k \) and let PrepareRankStruct and RankSortFunc be a ranking mechanism. Let \( A = (A_0, \ldots, A_{t+1}) \) be a polynomial adversary that for each query receives \( B \) and VIEW\( S_1 \) from RankSortFunc’s output. Then consider the experiment in Figure 3 for \( n_q = \text{poly}(k) \).

Let \( Q_b = (q^{(1)}_0, \ldots, q^{(n_q)}_b) \). We say that MRSE is secure against \( S_1 \) as long as \( \forall i \in \{1, \ldots, n_q\} \) \( |q^{(i)}_0| = |q^{(i)}_1| \) and histories \( H_0 = (D_0, Q_0) \) and \( H_1 = (D_1, Q_1) \) that have the same trace \( \tau(D_0, Q_0) = \tau(D_1, Q_1) \), then for all PPT \( A \) it holds that
\[
\Pr[\text{Expt}_{A}^{\text{IndS}1}(1^k) = 1] \leq 1/2 + \text{negl}(k).
\]

In our MRSE model we assume that it is sufficient for server \( S_1 \) to return to the client only document identifiers and not the actual documents. Thus, \( E_{StE} \) algorithm of the SSE scheme can return only the index \( I \) and omit the ciphertexts \( c \). However, \( S_1 \) can only query the system with the same search pattern as defined in Definition 8 which requires \( |q^{(i)}_1| = |q^{(i)}_1| \) as well as co-appearance of terms across queries. Note that \( S_1 \)'s capabilities are similar to those of the server in the original SSE definition [15] but as we will see below \( S_2 \) learns much less.
Stemming and removal of stop words is outside of the scope of our paper. The client sets up the system by generating a secret key for StE, the security parameter $k$, and let PrepareRankStruct and RankSortFunc be a ranking mechanism. Let $A = \{A_0, \ldots, A_{t+1}\}$ be a polynomial adversary who for every query receives $\text{VIEW}_{S_2}$, one of the outputs of $\text{RankSortFunc}$ functionality. Then consider the experiment in Figure 4 for $n_q = \text{poly}(k)$.

We say that MRSE is secure against $S_2$ if for all PPT $A$ and (D₀, D₁) having the same number of documents $N$ it holds that:

$$\Pr[\text{Expt}_{A}^{\text{Indp-S}_{2}}(1^k)] = 1 \leq 1/2 + \text{negl}(k).$$

### 4.2 MRSE Construction

We are now ready to present the details of our MRSE construction. We split our description into two phases: the setup phase and the query phase.

**Setup and Initialization** The client sets up the system by generating a secret key for StE, $\text{SK}_{\text{StE}}$, and a public/secret key pair for Paillier cryptosystem ($\text{PK}_{p}$, $\text{SK}_{p}$). Then, shares $\text{PK}_{p}$ with $S_1$ and (PK₀, SK₀) with $S_2$. We omit exact details of how the client sends the secret key to $S_2$ but any efficient key wrapping algorithm suffices for our purposes. $S_1$ and $S_2$ are honest but curious and interact with each other faithfully only using the private protocol from Section 3.2.

The client first extracts all M unique terms from his collection of documents D and associates every unique term $t$ with an array $F_t$ of size $N$. An element in position $d$ of list $F_t$ corresponds to the frequency of term $t$ in document with id $d$, i.e., $\text{tf-idf}_{t,d}$ as defined in Section 2.4. Note that $\text{tf-idf}_{t,d}$ is zero if the term $t$ does not appear in document $d$. Given all tf-idf, the client obtains the frequency table $F$ where the number of rows

\[ \text{Stemming and removal of stop words is outside of the scope of our paper.} \]
is $M$ (number of unique terms in $D$) and the number of columns is $N$ (number of documents in $D$). The client then maps every frequency score $tf-idf_{t,d}$ to an integer and encrypts it using the first layer of Paillier encryption (Section 2.1). The mapping to integers ensures that we can use Paillier cryptosystem whose plaintext space is defined over $\mathbb{Z}_n$. Note that, once encrypted, the table representing frequencies of terms does not reveal the number of documents that every term appears in, i.e., the length of the posting list. We overload the notation and define $[F_i] = \{[tf-idf_{t,d}] \mid \forall d \in \{1,N\}\}$. 

The client then wishes to upload encrypted term and frequency index to $S_1$ and query it later. For this purpose, he uses a structured encryption scheme as defined in Section 2.5. Since the frequencies are already in an encrypted form, it is sufficient to create a searchable index for all the terms and allow $S_1$ to find the corresponding frequency array $[F_i]$ only if he is given a trapdoor for $t$. To do so, we consider a simplified version of the labeled data structured encryption scheme described in [14]. Let $SK_{SE}$ consist of two random $k$-bit strings $K_1, K_2$ and let $G_{K_1}$ and $G'_{K_2}$ be two different pseudo-random functions (PRF) with keys $K_1$ and $K_2$, respectively.

The client first sorts the terms using the lexicographic order and numbers each term in this order as $\{t_1, t_2, \ldots, t_M\}$. Then, he picks a pseudo-random permutation $\pi$ and creates an auxiliary index of pairs $(t_i, \pi(i)) \forall i \in \{1,M\}$. He also appends $\pi(i)$ to the corresponding $[F_i]$ and permutes the pairs $(\pi(i), [F_i])$, i.e., creates a dictionary that maps a keyword $t_i$ to a list of encrypted scores for all the documents in the collection (not only the documents in which the keyword appears at).

Then, the encryption algorithm of $StE, E_{SE}$, works as follows. For every $i \in \{1,M\}$, the search key $k_i = G'_{K_2}(t_i)$ and the value $(\pi(i), [F_i]) \oplus G_{K_1}(t_i)$ are computed. Both are stored together in the secure index $I$ which is sent as an input to $S_1$. We do not give to $S_1$ the encryption of the document collection since this is outside of our model.

**Multi-user setting** Our scheme could potentially be extended to a multi-user setting where the data owner allows authorized users to search on the data stored in the cloud server. This can be done by using a multi-user SSE scheme (MSSE) like the one proposed by Curtmola et al. [15]. In order to authorize a user, the data owner creates a unique secret key for him which the user can later use to generate keyword trapdoors for search. MSSE also allows the data owner user to revoke users and forbid them accessing the collection.

**Query Phase** During the query phase, the client computes the trapdoor $T \leftarrow Trpdr(SK_{SE}, t)$ for each keyword $t$ in the query $q$. In our scheme, $Trpdr$ sets $T$ to $(G_{K_1}(t_i), G'_{K_2}(t_i))$. The client then sends the trapdoors of all the query terms (i.e., an “encrypted” representation of the query) $T = \{T \mid \forall t \in q\}$ to $S_1$. Server $S_1$, upon receiving client’s query $T$, can locate each encrypted keyword $t \in q$ using the corresponding trapdoors by running $Search(I, T) \forall T \in T$. The $Search$ algorithm parses $T$ as $(\alpha, \beta)$ and computes the answer as $I(\beta) \oplus \alpha$, where $I(\beta)$ is the value stored in $I$ under the search key $\beta$. The answer is a vector $[F_i] = \{[tf-idf_{t,d}] \mid \forall d \in \{1,N\}\}$ for every term $t$ in the query.

**Computing Document Scores:** Recall that $[F_i]$ is an array of individually encrypted $tf-idf_{t,d}$ scores for $d \in \{1,N\}$. In order to compute the document scores, $S_1$ uses the additive property of the homomorphic encryption scheme and for every document $d$ computes an encrypted score

$$e_d = [score(q,d)] = \sum_{t \in q}[tf-idf_{t,d}].$$

Note that $e_d$ is simply an encryption of $score(q,d)$. $S_1$ then creates an array $A$ of $(key, value)$ pairs where a $key$ is an encryption of a document id and value is the corresponding encrypted score:

$$A = \{(1, e_1), (2, e_2), \ldots, (N, e_N)\}.$$

**Sorting Document Scores:** The server $S_1$ has acquired the final scores for every document identifier, however, these scores are encrypted which prohibits $S_1$ from sorting them and returning the document identifiers sorted by their relevance to the query $q$. To sort the documents $S_1$ engages with $S_2$ in the private sorting protocol $EncSort$ defined in Table 8 and its extension to $(key,value)$ pairs in Section 3.2. The protocol returns to $S_1$ an array

$$B = \{(d_1, e_{d_1}), (d_2, e_{d_2}), \ldots, (d_N, e_{d_N})\}$$

which corresponds to a re-encryption of array $A$ sorted using document scores, that is

$$D(e_{d_1}) \leq D(e_{d_2}) \leq \ldots \leq D(e_{d_N})$$
where $D$ is a decryption algorithm of Paillier cryptosystem and $d_i$ are document identifiers. Moreover, as we showed in Section 3.2 the protocol maintains privacy guarantees against $S_1$ and $S_2$. In particular, $S_1$ learns nothing about the scores, their relative order in neither $A$ nor $B$, while the privacy guarantees against $S_2$ are even stronger. $S_2$ also learns nothing about $A$ nor $B$ as well as nothing about the search pattern. That is $S_2$ only learns that a query happened but learns nothing about it including the number of keywords nor whether the same query or keyword has been queried before or not.

Finally, $S_1$ sends to the client array $B$. If the client is not interested in document scores, server $S_1$ can send only the ordered encrypted document identifiers $[d_i]$ from the array $B$, i.e., the key from every pair in $B$. Note that our construction can be easily adapted for the case where the user only asks for the top $k$ documents.

Theorem 7 (MRSE Privacy). Let $\text{EncSort}$ be a secure private sorting protocol as defined in Definition 5. Then, the MRSE protocol described in Section 4.2 is secure against $S_1$ and $S_2$ according to Definitions 9 and 10.

Proof (Sketch). We first show how $\text{PrepareRankStruct}$ and $\text{RankSortFunc}$ are defined in MRSE. The ranking mechanism supported by MRSE is the tf-idf scoring system and, hence, the output of $\text{PrepareRankStruct}$ is a frequency table $[F_i]$ for each term $i$, as well as the search pattern. On a query $q$, $\text{RankSortFunc}$ takes as input tags $t \in T$ returned by $\text{StE Search}$ on a query $q$, finds the corresponding rows in $[F_i]$, adds them to get an array $A$, and then performs $\text{EncSort}$ on $A$ to receive the sorted result $B$.

Security against $S_1$ Definition 9 guarantees that $S_1$ could not possibly distinguish between two different document collections as long as they have the same trace. Recall from Definition 7 that the trace of a collection includes the number of documents, $N$, and the number of unique keywords, $M$, as well as the search pattern. According to the indistinguishability game (Figure 3) assume that an adversarial $S_1$ picks two document collections with the same trace. He is then given an index $I$ encrypted under $\text{StE}$ and a frequency table $[F_i]$ encrypted under Paillier for one of them, say $b$, chosen uniformly at random. By only seeing $I_b$ and $[F_i]$, $S_1$ cannot guess $b$ with probability greater than $1/2 + \text{negl}(k)$. $[F_i]$ gives no information to $S_1$ since it is constructed in such a way that the number of keywords per document is hidden (padded with 0s). Also, $I_b$ will be of the same size for $b = 0, 1$.

In the next phase, $S_1$ picks queries with the same answer size in both collections and receives the corresponding set of query trapdoors for one of the collections. The trapdoors are generated using the $\text{Trpdr}$ algorithm of the $\text{StE}$ scheme which is deterministic. Once the trapdoors are generated, $S_1$ is given the trapdoors, the sorted array $B$ as well as the view of the first server of $\text{EncSort}$. Given that the instantiation of the $\text{StE}$ scheme for labeled data [14] is secure against adaptive chosen query attacks, $S_1$ cannot possibly infer anything about which are the actual terms being queried, thus the privacy of the data collection is preserved. As in the $\text{StE}$ instantiation of [14], the search pattern is also leaked to $S_1$. Note that the keywords in $I_b$ and the corresponding rows of $[F_i]$ are permuted, thus, an adversarial $S_1$ cannot use lexicographical order of queried keywords to distinguish the collections. Finally, by $\text{EncSort}$ privacy we guarantee that $S_1$ learns nothing about the sorting of the documents thus cannot use any information to distinguish between the two collections.

Security against $S_2$ In the MRSE construction $S_2$ behaves in exactly the same way as $S_2$ does in $\text{EncSort}$, thus, inherits the security properties of private sort second server. The only information leaked to $S_2$ is the number of documents included on the collection since he can infer that by the number of pairs he will be given to compare for one query. However, this is not enough to distinguish given that the two challenge documents are of the same size $N$.

Overall privacy As long as the servers are honest and non colluding the MRSE scheme presented in Section 4.1 is secure. As noted above, given that the servers are non colluding, it is sufficient to study security against each one of them independently. \hfill $\square$

The performance of MRSE is summarized in the following theorem.

Theorem 8 (MRSE Performance). MRSE protocol presented in Section 4.2 gives the following performance guarantees:

- The client takes $O(|D|)$ time and space to setup the system, and $O(|q|)$ time to generate a query;
- The communication cost between the client and $S_1$ during the query phase is $O(|q| + N)$;
- The space requirements for $S_1$ and $S_2$ are $O(N \times M)$ and $O(1)$, respectively;
The query phase takes \(O(N(\log N)^2)\) for both \(S_1\) and \(S_2\); where \(|D|\) is the size of the data collection \(D\), \(N\) is the number of documents and \(M\) is the number of unique terms in \(D\), and \(|q|\) is the query size.

**Proof (Sketch).** The initialization cost of the system for the client is linear in the size of the document collection since term frequencies can be computed while parsing the collection and maintaining an efficient index of unique terms for updating the corresponding occurrence count. Hence, the client performs \(O(N \times M)\) work before uploading the search and frequency indices to \(S_1\).

During the query phase the client sends to \(S_1\) a trapdoor for every keyword in the query \(q\) and as a result obtains \(1 \leq l \leq N\), ordered document identifiers where \(l\) is the parameter specified by the client. Hence, query generation time for the client is \(|q|\) and overall communication cost between the client and \(S_1\) in the worst case is \(O(|q| + N)\).

We now analyze the query phase for \(S_1\) and \(S_2\). \(S_1\) queries \(\text{StE}\) in \(O(|q|)\) time \([15]\) and sums the corresponding frequency score vectors in \(|q| \times N\) time, i.e., to obtain \(e_d\) for every document. Private sort of encrypted scores takes \(O(N(\log N)^2)\) rounds of communication between \(S_1\) and \(S_2\) since Batcher’s sorting network gives the dominating cost. Hence, the total work for \(S_1\) and \(S_2\) is \(O(N(\log N)^2)\).

Note that the non-secure counterpart of the same ranking and sorting functionality also takes \(|q| \times N\) time to add the scores for every document, while sorting in the clear takes \(O(N \log N)\). Hence, our privacy preserving construction over encrypted data gives only a \(\log N\) multiplicative overhead over a non-secure construction. If \(S_1\) and \(S_2\) operate in parallel the runtime overhead can be dropped to \(O((\log N)^2)\).

## 5 MRSE-X Protocol

In this section we present a scheme that is more efficient than the scheme in the previous section but achieves weaker privacy guarantees in return. Recall that MRSE hides the access pattern of SSE by padding the posting lists to be of the same length and inserting zero scores for a document-keyword pair in cases when a keyword is not present in a given document. In this section, we use the single keyword SSE scheme of \([15]\) and extend it to support ranking for multiple keyword queries while having same security guarantees as \([15]\). We also discuss how to include our ranking mechanism in a scheme by Cash et al. \([13]\) that addresses conjunctive queries directly as opposed to combining results of multiple single keyword queries.

### 5.1 Ranking for Curtmola et al. \([15]\) Scheme

**Setup and Initialization** The main difference between MRSE-X and MRSE is in the construction of the frequency table \(F\). In particular, for every term \(t\) we compute \(tf-idf_{t,d}\) only if \(t\) appears in document \(d\). We then set \(F_t\) to contain pairs \((d, tf-idf_{t,d})\). We encrypt entries \(tf-idf_{t,d}\) using the first layer of Paillier and leave document identifiers open. Hence, now the frequency mapping is \(F_t = \{(d, [tf-idf_{t,d}]) \mid \forall d \in J_t\}\), where \(J_t\) is the posting list of \(t\).

The client sets up \(\text{StE}\) scheme by simply feeding (key, value) pairs \((t, F_t), \forall t\), as the dictionary \(\delta\) to \(E_{\text{StE}}\) (similar to MRSE). The output of \(E_{\text{StE}}\) is again the secure index \(I\) that is sent to \(S_1\).

**Query Phase** For a query \(q\) the client proceeds as before. He creates a trapdoor \(T_t\) by running \(\text{StE} \ Trpdr\) for every term \(t\) in the query \(q\), and sends \(T_q = \{T_t \mid \forall t \in q\}\) to \(S_1\). \(S_1\) then runs \(\text{Search}\) procedure for every trapdoor \(T_t\) and as a response receives pairs \((d, [tf-idf_{t,d}]\)), where document identifiers \(d\) are in the clear. His next task is to compute the sum of the encrypted scores. Let \(J_q = \bigcup_{t \in q} J_t\), if \(q\) is a disjunctive query, or \(J_q = \bigcap_{t \in q} J_t\) for a conjunctive query. Then \(\forall d \in J_q:\)

\[
e_d = [\text{score}_d] = \sum_{\forall t \in q \land d \in J_t} [tf-idf_{t,d}].
\]

\(S_1\) constructs \(A = \{(d, e_d) \mid \forall d \in J_q\}\) by encrypting \(d\)’s in \(J_q\) himself. The rest of the protocol is the same as for MRSE: \(S_1\) proceeds by calling \(S_2\) to get a sorting of \(A\). \(B\). It is important to note that the size of query response \(J_q\), \(N\), is the size of the union of posting lists of keywords that appeared in the query and, hence, \(N \leq N\). In practice, however, \(N\) should be much less than \(N\).
Performance and Security. We use SSE-2 construction of [15] to instantiate our StE scheme\textsuperscript{12} since it has optimal search time and is adaptively secure. The performance of the resulting MRSE-X is summarized in the following theorem.

**Theorem 9 (MRSE-X Performance).** MRSE-X protocol presented in Section 5 gives the following performance guarantees:

- The client takes $O(|\mathbf{D}|)$ time and space to setup the system, and $O(|q|)$ time to generate a query;
- The communication cost between the client and $S_1$ during the query phase is $O(|q| + \bar{N})$;
- The space requirements for $S_1$ and $S_2$ are $O(L)$ and $O(1)$, respectively;
- The query phase takes $O(N(\log \bar{N})^2)$ for both $S_1$ and $S_2$;

where $|\mathbf{D}|$ is the size of the data collection $\mathbf{D}$, $N$ is the number of documents in $\mathbf{D}$, $L$ is the size of the index (sum of lengths of posting lists for unique terms in $\mathbf{D}$), and $|q|$ is the query size and $\bar{N}$ is the size of a query reply where $\bar{N} < N$.

The space requirement at $S_1$ is reduced due to the size of the frequency table. In particular, it is $N \times M$ for MRSE and $L$ for MRSE-X where $L = \sum_{t \in q} J_t$. In practice, $L$ is much smaller than $N \times M$ but in the worst case, when every term appears in every document, $L = N \times M$.

In return for the performance improvement in MRSE-X, the security for $S_1$ and $S_2$ is weakened compared to MRSE. $S_1$ learns $L$, the sum of posting list sizes for $\mathbf{D}$, and the access pattern [15]. Informally, the access pattern reveals which documents were accessed for every keyword in the query. The security definition for $S_1$ can be trivially extended from Definition 9 to accommodate for this leakage as follows. We set $L$ and the access pattern to be part of the trace that is enforced on the document collection and query sequence allowed to be queried by the adversary during $\text{Inds}_{S_1}$ game in Figure 3 (we refer the reader to [15] for more details).

The definition of security for $S_2$ extends $\text{Inds}_{S_2}$ game in Figure 4, since $S_2$ learns $\bar{N}$ every time he assists in private sort. Hence, $\mathcal{A}_t$ is given the size of the query response after every query in $\text{Inds}_{S_2}$ game.

### 5.2 Ranking for Cash et al. [13] Scheme

The ranking mechanism can be also applied to a recent construction by Cash et al. [13]. This construction handles conjunctive queries over multiple terms directly. Hence, it has different guarantees in terms of performance and leakage of access pattern, if compared to our naive extension of [15] in previous section. Below we apply our ranking mechanism to [13].

We first give an overview of the construction by Cash et al. [13] and refer the reader to the original paper for details. The construction considers a client who sends Boolean queries to a server who stores a search index and replies client’s queries. The main intuition behind the method is to let the server open only a small subset of document ids for every keyword in the query, instead of opening document ids for all keywords in the query.

In particular, the client picks the first keyword in query $q$, $t_1$, and creates trapdoor-like tags for every document where $t_1$ appears, i.e., a tag for every document in the posting list of $t_1$, $J_{t_1}$. The client also creates secure tags for every remaining keyword in $q$. The server then combines all the received tags and verifies the presence of keywords in $q$ only for a subset of documents in $J_{t_1}$. Note that for conjunctive queries it is enough to check document ids in $J_{t_1}$. Then the server returns to the client only the document ids for which he could verify the presence of all keywords in the query. We note that as a result the server stores a tag for every $(t, d)$ pair where term $t$ appears in document $d$.

The scheme of [13] has the following performance. For a query $q$, the client sends $|q| + |J_{t_1}|$ objects to the server, where $|J_{t_1}|$ is the size of the posting list for the first term in the query. The running time at the server is $|q| \times |J_{t_1}|$ since he has to build all possible keyword-document pairs to verify their presence. This scheme is more efficient than [15] when $\sum_{t \in q} |J_t| > |q||J_{t_1}|$, e.g., when $t_1$ is a very rare word while $t_2 \in q$ is a very frequent word.

We extend this method with our ranking functionality as follows. Since the server already stores a tag for every $(t, d)$ pair, we can simply append $[\text{tf-idf}_{t,d}]$ to every tag. The rest follows trivially from MRSE-X construction for [15] in Section 5.1, where $S_1$ behaves as the server of [13] when searching for keyword-document match, adds the encrypted tf-idf scores for the documents in the query reply, and engages in private sort with $S_2$ to sort these documents according to encrypted tf-idf scores.

\textsuperscript{12} Though the encryption method in [15] takes as input a document collection $\mathbf{D}$, it first produces an intermediate dictionary structure and only then encrypts it. Hence, this method can be easily extended to take our dictionary $\delta$ as its input instead.
6 Discussion

MRSE and MRSE-X return to the client only document identifiers and not documents themselves. Here, we argue that with presently known techniques a system that returns documents ranked according to a search query using tf-idf is either not secure or very expensive.

Searchable symmetric encryption schemes are known to reveal search and access patterns where access pattern refers to the documents that are returned as a result for each query (recall that MRSE reveals only the search pattern). Hence, the server learns co-occurrence of queried terms in documents. Indeed, Islam et al. [27] show how one can reconstruct single word queries made to an SSE scheme by observing search and access patterns. Now consider a SSE scheme that indeed allows one to return documents sorted based on keyword frequency and allows for multi-term queries. This scenario gives even more possibilities for an attack since now the server learns word frequencies in documents and can potentially reconstruct the document collection. 

Example: consider three queries \((t_1), (t_2)\) and \((t_1, t_2)\), where each keyword is hidden using SSE trapdoor, and their corresponding response with ranked documents are \((d_1, d_2, d_3)\), \((d_3, d_2, d_1)\) and \((d_1, d_2, d_3)\). Knowing the basic principle behind tf-idf ranking system, it is clear to the attacker that \(t_1\) is a more “important” word in the system (e.g., rare) and that \(t_1\) occurs more often in \(d_1\) than \(d_2\). Hence, revealing the order of the document based on a ranking significantly increases the leakage of a SSE scheme.

Techniques for mitigating the above attacks by hiding search and access patterns are expensive. Oblivious RAM [22, 25] requires poly-log number of rounds of communication between the client and the server, where every round would require sending back and forward encrypted documents. Private Information Retrieval (PIR) [31], on the other hand, for every query requires that the server touches every document he stores.

Finally we note that previous work [11] on ranked multi-keyword search also returns only document ids and not the documents. While the construction of [36] does return the documents but reveals the access and search patterns as well as the ranking to one of the servers.

7 Related Work

In recent years the topic of searching over encrypted data has received a lot of attention. Many models have been proposed, each one of them trying to solve the problem under a different prism or trying to achieve different properties. We start by giving an overview of the literature on searchable encryption and then we discuss and compare our scheme with multi-keyword searchable encryption schemes that return ranked results.

7.1 Searchable encryption

In the standard searchable encryption setting one wishes to perform a boolean search on an encrypted document (potentially stored on a remote server) to decide whether a keyword exists in the document without having to decrypt it. In the symmetric scenario the owner of the data encrypts them himself so that he can first organize it in a certain data structure to make search more efficient. The encrypted data and the corresponding data structure can be stored on a server so that only the data owner (or a user authorized by him) can search on them. Song et al. [42] are the first to introduce symmetric searchable encryption for a single user. Their scheme requires two layers of encryption and search time by the server is linear in the length of the document collection for every query.

Curtmola et al. [15] formally defined symmetric searchable encryption (SSE) and gave efficient constructions. Moreover, they proposed multi-user SSE, where even though a single user owns the data, there exists an arbitrary group of users (authorized by the owner) that can submit search queries to the cloud by computing a trapdoor for the term to be queried. Another notable work is the one due to Kamara et al. [29] who propose a solution in the symmetric setting that not only allows single keyword search over encrypted data but also allows a user to verify the integrity of the returned documents and verify that the server has all the files, i.e., proof of ownership protocols are integrated into their solution. Recently Cash et al. [12] proposed a symmetric scheme that allows for efficient multi-term conjunctive queries.

Boneh et al. [9] were the first to introduce the idea of asymmetric encryption with keyword search (PEKS) and their work inspired a series of other proposals [1, 6, 24, 46]. In this public key setting users who encrypt the data and send them to the server may be different from the user who owns the decryption key and is able to perform search queries (e.g., consider mail server scenario).

Schemes that do not require a precise match to a search query have also been proposed [33, 32]. They use a similarity measurement to find documents that match the query.
### 7.2 Comparison with Related Work

In this section we compare MRSE with other multi-keyword searchable encryption schemes with ranked results. Cao et al. [11] provide one of the first schemes that allow ranked multi-keyword search. The scheme sorts documents using the score based on “inner product similarity” (ips) where a document score is simply the number of matches of query keywords in each document. This ranking is not as standard in information retrieval as tf-idf since it loses information about keyword importance to the document collection w.r.t document lengths and other keywords (e.g., documents which contain all query keywords are ranked equally). The scheme of [11] also proposes a heuristic to hide the search and access patterns by adding dummy keywords and noise. As a result, the returned document list may contain false negatives and false positives. Query phase of the scheme is expensive for the client since query generation time is \(O(M^2)\), i.e., quadratic in the number of unique keywords in the original collection, \(M\), and the length of the trapdoor for every query is \(O(M)\). To answer the query, the server has to perform \(O(N \times M^2)\) computation, where \(N\) is the number of documents in the collection. In comparison, the client of our scheme is required to generate only a constant size trapdoor for every term in the query which is likely to be much smaller than \(M\). Also, the work for the server in MRSE is \(O(N \log N)^2\).

Örencik and Savaş [37, 38] also propose protocols for ranked multi-keyword search. Their ranking is loosely based on frequency of a word in the document where fake keywords and documents are added, hence, their scheme also may return false negatives and positives. Recent proposal by Örencik et al. [36] is a solution with two non-colluding servers. Their first server works similar to our \(S_1\), however, the interaction between the two servers is very different and gives much weaker privacy guarantees than our system. In particular, the second server has access to the result of every query in the clear, revealing information about user’s data collection as well as the search and access patterns. Recall that in our scheme \(S_2\) is merely assisting \(S_1\) during sorting and never sees neither the queries nor the data. Finally, storage requirement of the second server is linear in the size of the collection, while it is constant for \(S_2\) in MRSE.

Another recent work that uses tf-idf and inner product similarity based ranking is the one due to Strizhov and Ray [43]. Their model assumes a single server that performs only the search functionality and not the sorting of the results. In particular, the client generates \(N\) trapdoors for every term in the query, the server finds the required encrypted documents and scores, returns them to the client who performs the sorting based on tf-idf himself. Moreover, the frequency table has to be encrypted under a fully homomorphic encryption (FHE) scheme in order for the server to be able to perform ranking. Using FHE in such a setting is a direct solution but unfortunately is very inefficient.

In Table 9 we present a comparison of our MRSE scheme with the schemes discussed above. We compare them in terms of soundness of the returned result (e.g., if the result contains false positives), ranking method, client query generation time and search complexity for the server(s). The last column of the table presents privacy guarantees of the schemes. We note that privacy of [11] is harder to compare with since a heuristic is used to hide access and search patterns.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Sound</th>
<th>Ranking</th>
<th>Server(s)</th>
<th>Privacy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cao et al. [11]</td>
<td>ips</td>
<td>tf-idf</td>
<td>(M^2) (N \times M^2)</td>
<td>Precision-privacy tradeoff</td>
</tr>
<tr>
<td>Örencik et al. [36]</td>
<td>✓</td>
<td>tf-idf</td>
<td>(</td>
<td>q</td>
</tr>
<tr>
<td>Strizhov-Ray [43]</td>
<td>✓</td>
<td>ips†</td>
<td>(</td>
<td>q</td>
</tr>
<tr>
<td>Our scheme MRSE</td>
<td>✓</td>
<td>tf-idf</td>
<td>(</td>
<td>q</td>
</tr>
</tbody>
</table>

† the client receives document scores and sorts them himself.
‡ security against \(S_1\), \(S_2\) is in fact stronger than CCA-2 (see Section 4.1).
8 Future Work

In this paper we presented ranked text search as one appealing application of our new private sorting functionality. In the future we wish to find other cloud computing scenarios that rely on sorting. One immediate application is database queries that require a join on sorted tables or to return top $k$ results satisfying a query.

Our construction, as well as previous work that addresses ranking, returns as a result to the user only document identifiers and not documents themselves. If one does otherwise, he reveals the documents being queried for and information about word frequencies in each document (due to the tf-idf ranking). An obvious solution is to use techniques that hide search and access patterns, i.e., Oblivious RAM [22] and Private Information Retrieval [31]. However, such techniques are known to give high overhead. We also propose to consider a system that can support ranked search as well as efficient secure updates to the data collection. Another appealing scenario is to consider a scheme that is secure against malicious servers and allows users to verify the integrity of returned results.

Acknowledgments

The authors would like to thank Seny Kamara, Markulf Kohlweiss and Roberto Tamassia for useful discussions and suggestions on how to improve the results and the write-up in hand. Olga Ohrimenko worked on this project in part while at Brown University, where her research was supported in part by the National Science Foundation under grants CNS–1012060 and CNS–1228485. Foteini Baldimtsi was supported in part by grants CNS–1012910 (Boston University), FINER project by Greek Secretariat of Research Technology (University of Athens) and CNS–0964379 (Brown University).
References