Sorting and Searching Behind the Curtain

Foteini Baldimtsi[1] and Olga Ohrimenko[2]

[1] Boston University, USA and University of Athens, Greece {foteini@bu.edu}
[2] Microsoft Research, UK {oohrim@microsoft.com}

Abstract. We propose a framework where a user can 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. Sorting is achieved via a novel protocol where the server is assisted by a secure coprocessor that is required to have only 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 present an efficient construction that is based on an encryption scheme with semi-homomorphic properties.

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.

Keywords: Private sort · privacy in the cloud · ranked search on encrypted data.

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_{sorted}, back to him. However, Mr. Smith does not trust Brad and wishes to keep A, B and B_{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) [14, 28], 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) [7] 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.* [26]). 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_{sorted} which is the sorting of an encrypted B, while neither of them learns anything about the plaintext values of B nor B_{sorted}. It follows from the privacy requirement that Angelina never sees an encryption of neither B nor B_{sorted}, otherwise she could trivially decrypt them.

The Brad and Angelina model is often encountered in reality. We can see Brad as the provider of cloud storage and computation (i.e., cloud server) 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[3] or the Freescale C29x[4]) 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-collusion is justified since the cloud provider and secure coprocessor 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 additive homomorphic properties of an encryption scheme. Sorting is, arguably, one of the most common and well studied computations [18] 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 server to privately and efficiently sort encrypted data, but also allows certain computations on the

[3] http://www-03.ibm.com/security/cryptocards/pciecc/overview.shtml
[4] http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=C29x

data. Hence, it can be a useful tool for answering sophisticated queries on an encrypted database and, for example, returning top results satisfying the query.

The main component of most of the sorting algorithms is the pairwise comparison of elements. A few methods for comparison of encrypted data have been proposed in the literature, where the most well known ones either depend on homomorphic encryption or on garbled circuits. The protocol by Veugen [29] depends on homomorphic encryption and presents a private comparison protocol where the cloud server learns the result of the comparison, while the coprocessor learns nothing. To use a garbled circuits solution, as Bost *et al.* [8] suggest, one could use the comparison circuit by Kolesnikov *et al.* [19] in combination with the efficient garbled circuit implementation of Bellare *et al.* [6] and an oblivious transfer protocol like the one due to Asharov *et al.* [3]. However, this solution requires the parties to generate a fresh circuit for each pairwise comparison. For our construction we choose to use homomorphic encryption techniques due to its simplicity and efficiency.

Outsourced Ranked Text Search. We give a concrete application of our new sorting framework through the problem of outsourcing *ranked search* over encrypted data to the cloud. The goal is to rank the result according to its relevance to the query by using the standard frequency (tf) and inverse document frequency (idf) method [30]. 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 relatively 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 locates the term in the index and returns 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: (a) add the located rows of encrypted frequencies together in order to compute the score of every document w.r.t. the query, and (2) sort the resulting list of encrypted scores.

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 [25]) and then outsource them to the cloud server, S_1. S_1 is equipped with a secure coprocessor, S_2, who

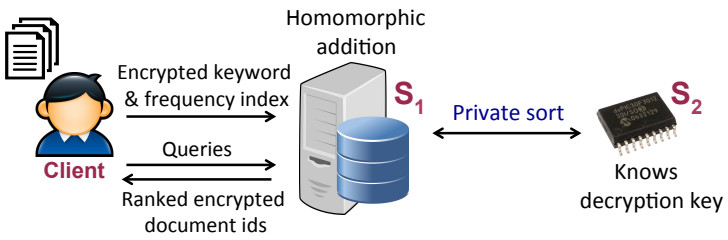

Fig. 1: Ranked Multi-keyword Searchable Encryption (MRSE) Model.

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 S_2. We refer to our proposed construction as Multi-keyword Ranked Searchable Encryption (MRSE) and give its overview in Figure 1.

Our contributions are summarized below:

- Formally define *private outsourced sorting* (Definition 1) and present a simulation based privacy definition (Definition 2).
- 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.
- In Section 4.1 we present MRSE, the first system that efficiently supports multi-keyword search and computes a ranked result based on word frequencies in a secure and private way.

2 Preliminaries

In Table 1 we summarize the notation used throughout the paper. Then, we present the building blocks used for our construction.

2.1 Homomorphic Cryptosystem Protocols

Paillier Cryptosystem The Paillier cryptosystem [25] 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 $K_{\mathsf{P}} = (\mathsf{PK}_{\mathsf{P}}, \mathsf{SK}_{\mathsf{P}})$. Paillier cryptosystem is homomorphically additive, that is, $[m_1] \cdot [m_2] = [m_1 + m_2]$.

Generalized Paillier Our construction relies on the generalization of the Paillier cryptosystem introduced by Damgård and Jurik [13] along with its special property that allows to doubly encrypt messages and use the additive homomorphism of the inner encryption layer under the same secret key [1, 20]. By $[m]$ 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.

Table 1: Notation.

Symbol	Meaning
k	security parameter
$K_{\mathsf{P}} = (\mathsf{PK}_{\mathsf{P}}, \mathsf{SK}_{\mathsf{P}})$	Paillier public/secret keys
$K_{\mathsf{QR}} = (\mathsf{PK}_{\mathsf{QR}}, \mathsf{SK}_{\mathsf{QR}})$	QR public/secret keys
$[m], [\![m]\!], \|m\|$	m encrypted using first and second layers of Paillier, and QR
Gen_{StE}, E_{StE}, SK_{StE}	StE keygen, encr. and secret key
$\mathbf{D} = \{D_1, \ldots, D_N\}$	document collection of size N
M	number of unique terms/keywords in $\mathbf{D}$
t, T	term/keyword and its StE trapdoor
$q = (t_1, \ldots, t_{l_q})$	query of l_q terms
F	frequency table
I	secure search index

The 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 [1]:

$$[\![[m_1]]\!]^{[m_2]} = [\![[m_1][m_2]]\!] = [\![[m_1 + m_2]]\!].$$

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

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 [1] (with several modifications) and define $[\![[c]]\!] \leftarrow \mathsf{EncSelect}(\mathsf{PK}_{\mathsf{P}}, \mathsf{SK}_{\mathsf{P}}, [a], [b], [\![v]\!]^5)$ where $(\mathsf{PK}_{\mathsf{P}}, \mathsf{SK}_{\mathsf{P}})$ is a pair of Paillier public, secret keys as before, a and b are the two plaintext values and v is a bit that indicates whether a or b should be returned. If v is 0, $\mathsf{EncSelect}$ returns a re-encryption of a, otherwise it returns a re-encryption of b. Hence, $\mathsf{EncSelect}$ imitates the computation $c = (1 - v) \times a + v \times b$ but over ciphertexts as follows:

$$\mathsf{EncSelect}\,([a], [b], [\![v]\!]) = ([\![1]\!][\![v]\!]^{-1})^{[a]}[\![v]\!]^{[b]} = [\![(1 - v)[a] + v[b]]\!] = [\![[c]]\!].$$

Note that the result c is doubly encrypted. For our purposes we require the output values to be encrypted using the first layer of Paillier encryption only. Simply sending $[\![[c]]\!]$ for re-encryption would be insecure since S_2 would learn the value of c. Instead, we propose a protocol $\mathsf{StripEnc}$, where S_1 randomizes the encryption of the value x he wants S_2 to re-encrypt, receives the re-encryption and removes the randomization. Hence, when S_2 decrypts the element he receives

[5] We note that v has to be encrypted using the second layer of Paillier in order to use the homomorphic properties of the cryptosystem.

Table 2: $[x] \leftarrow \mathsf{StripEnc}(\mathsf{PK_P}, \mathsf{SK_P}, [\![x]\!])$: Interactive protocol between S_1 and S_2 for stripping off one layer of Paillier encryption.

$S_1(\mathsf{PK_P}, [\![x]\!])$		$S_2(\mathsf{PK_P}, \mathsf{SK_P})$
pick $r \in \{0,1\}^{\ell+1}$		
$[\![x+r]\!] := [\![x]\!]^{[r]}$	$\xrightarrow{[\![x+r]\!]}$	
		decrypt $[\![x+r]\!]$
	$\xleftarrow{[x+r]}$	encrypt $[x+r]$
$[x] := [x+r][r]^{-1}$		

a random value and learns nothing about x. The complete protocol $\mathsf{StripEnc}$ is presented in Table 2 where we rely on the homomorphic properties of layered Paillier encryption.

We analyze the privacy guarantees of $\mathsf{StripEnc}$ and $\mathsf{EncSelect}$ in the full version of the paper [4] and show that each primitive can be simulated due to semantical properties of Paillier encryption.

Private Comparison of Encrypted Data In this work, 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 $\mathsf{SK_P}$. 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. For this purpose, we adapt the protocol of Bost *et al.* [8] and define $[\![v]\!] \leftarrow \mathsf{EncCompare}(\mathsf{PK_P}, \mathsf{SK_P}, \mathsf{PK_{QR}}, \mathsf{SK_{QR}}, [a], [b])$, an interactive comparison protocol between S_1 and S_2 that gives the above security guarantees.

The protocol is given in Table 3 and it proceeds as follows. S_2 knows the encryption and decryption keys for both Paillier and QR, $(\mathsf{PK_P}, \mathsf{SK_P}, \mathsf{PK_{QR}}, \mathsf{SK_{QR}})$, while S_1 knows the corresponding public keys $(\mathsf{PK_P}, \mathsf{PK_{QR}})$ and two values a and b encrypted under Paillier's scheme. S_1 first computes $[z] = [a] \cdot [b]^{-1} \cdot [2^\ell] \bmod n^2$ 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 $\tilde{d} = d \bmod 2^\ell$, S_1 similarly computes $\tilde{r} = r \bmod 2^\ell$ and they engage in a private input comparison protocol (we can use the DGK protocol [12] as suggested by Bost *et al.* [8]) that compares $\tilde{d}$ and $\tilde{r}$. At the end of this protocol, S_1 receives an encrypted bit λ that shows the relation between $\tilde{d}$ and $\tilde{r}$ ($\lambda = 1 \Leftrightarrow \tilde{d} < \tilde{r}$). The output λ 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 $\|v\| = \|d_\ell\| \cdot \|r_\ell\| \cdot \|\lambda\|$. 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.

Table 3: $[\![v]\!] \leftarrow \mathsf{EncCompare}(\mathsf{PK_P}, \mathsf{SK_P}, \mathsf{PK_{QR}}, \mathsf{SK_{QR}}, [a], [b])$: 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 5. This protocol is an adaptation of the comparison protocol from [8].

$S_1(\mathsf{PK_P}, \mathsf{PK_{QR}}, [a], [b])$		$S_2(\mathsf{PK_P}, \mathsf{SK_P}, \mathsf{PK_{QR}}, \mathsf{SK_{QR}})$
$[z] := [2^\ell] \cdot [a] \cdot [b]^{-1} \bmod n^2$		
pick $r \in \{0,1\}^{\ell+k}$.		
$[d] := [z] \cdot [r] \bmod n^2$	$\underrightarrow{[d]}$	
		decrypt $[d]$
$\tilde{r} := r \bmod 2^\ell$		$\tilde{d} := d \bmod 2^\ell$
	$\underleftrightarrow{\|\lambda\| \leftarrow \text{Compare } \tilde{r}, \tilde{d}}$	
	$\underleftarrow{\|d_\ell\|}$	encrypt d_ℓ
encrypt r_ℓ		
$\|v\| := \|d_\ell\| \cdot \|r_\ell\| \cdot \|\lambda\|$		
$[\![v]\!] \leftarrow \mathsf{ReEncryptBit}(\mathsf{PK_P}, \mathsf{SK_P}, \mathsf{PK_{QR}}, \mathsf{SK_{QR}}, \|v\|)$		

The above protocol returns as a result bit v encrypted using QR cryptosystem, $\|v\|$, for which only S_2 knows the secret key $\mathsf{SK_{QR}}$. 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, $[\![v]\!]$. Below we introduce $[\![v]\!] \leftarrow \mathsf{ReEncryptBit}(\mathsf{PK_P}, \mathsf{SK_P}, \mathsf{PK_{QR}}, \mathsf{SK_{QR}}, \|v\|)$ protocol to securely re-encrypt the bit v such that neither S_1 nor S_2 learns its value. The privacy guarantees of the complete EncCompare can be found in the full version of the paper [4].

The ReEncryptBit protocol consists of the following steps:

- S_1 picks a random bit r.
- S_1 computes two values $\|s_r\| := \|v\| \cdot \|0\|$ and $\|s_{1-r}\| := \|v\| \cdot \|1\|$ which are equal to $v \oplus 0$ and $v \oplus 1$, respectively.
- S_1 sends $\|s_0\|$ and $\|s_1\|$ to S_2 (i.e., using r as a secret permutation).
- S_2 decrypts them (always gets a "0" and a "1" in an order that is independent of v), 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 bit r, he outputs $[\![s_r]\!]$ which corresponds to the relation between a and b.

2.2 Searchable Encryption

Symmetric searchable encryption (SSE) allows a user that has in his possession a collection of documents $\mathbf{D} = \{D_1, \ldots, D_N\}$ to compute a "secure search index", I, over these documents and then outsource the index to the server. The

server should be able to search on I without learning anything about the actual collection $\mathbf{D}$. 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 [11]. Here we consider a generalized notion of SSE called: *structured encryption* (StE), that was given by Chase and Kamara [10] and allows SSE for arbitrarily-structured data. In particular, a data type $\mathcal{T}$ is defined by a universe $\mathcal{U} = \{\mathcal{U}_k\}_{k\in\mathbb{Z}}$ and an operation $\texttt{Query}\colon \mathcal{U}\times\mathcal{Q} \to \mathcal{R}$ with $\mathcal{Q} = \{\mathcal{Q}_k\}_{k\in\mathbb{N}}$ being the query space and $\mathcal{R} = \{\mathcal{R}_k\}_{k\in\mathbb{N}}$ being the response space, where k is the security parameter. The StE scheme then consists of the following algorithms:

$\mathsf{SK}_{\mathsf{StE}} \leftarrow \mathit{Gen}_{\mathsf{StE}}(1^k)$ run by the owner of the data. The output is owner's secret key $\mathsf{SK}_{\mathsf{StE}}$ for the security parameter k.

$I \leftarrow E_{\mathsf{StE}}(\mathsf{SK}_{\mathsf{StE}}, \delta)$ run by the owner to encrypt a data structure δ of type $\mathcal{T}$, under his secret key $\mathsf{SK}_{\mathsf{StE}}$. The output is the secure index (encrypted data structure) I sent to the server.

$T \leftarrow \mathit{Trpdr}(\mathsf{SK}_{\mathsf{StE}}, t)$ is a deterministic algorithm run by the owner to generate a trapdoor T for a query $t \in \mathcal{Q}$. It outputs either T or the failure symbol $\perp$.

$a \leftarrow \mathit{Search}(I, T)$ is run by the server to perform a search for a trapdoor T and outputs an answer $a \in \mathcal{R}$.

The construction of ranked search discussed in Section 4.2 uses a dictionary data type for StE. In particular, the keys (or queries) of a dictionary are keywords of the document collection $\mathbf{D}$. 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.

2.3 Text Search and Ranking

We represent a document collection using an inverted index [30]. 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 [21]. 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 $J_q = \bigcup_{\forall t\in q} J_t$.

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 [30]. 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: $\mathrm{idf}_t = \log \frac{N}{\mathrm{cf}_t}$. Document frequency of term t in document d is defined as: $\text{tf-idf}_{t,d} = \mathrm{tf}_{t,d} \times \mathrm{idf}_t$, where $\mathrm{tf}_{t,d}$ is frequency of term t in document d. If a document d does not contain t, $\text{tf-idf}_{t,d} = 0$.

Given a free text query q for each document $d \in J_q$ a score based on frequencies is computed as $\mathrm{score}(q, d) = \sum_{t\in q} \text{tf-idf}_{t,d}$. 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.

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.

3.1 Model

Private sort[6] 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 1. *(*$\mathsf{EncSort}$*) An encrypted sorting functionality* $\mathsf{EncSort}(\mathsf{PK}, \mathsf{SK}, A)$ *takes as input a public/secret key pair (*PK,SK*) of a semantically secure cryptosystem* $\{Gen_{\mathsf{SS}}, E_{\mathsf{SS}}, D_{\mathsf{SS}}\}$*, and an array* $A = [E_{\mathsf{SS}}(v_i)]_{i\in\{1,N\}}$ *of* N *elements where each element is encrypted individually using* PK*. Let* π *be a permutation of indices 1 to* N *that corresponds to the indices of* A*'s elements sorted using its unencrypted values* v_i*. Then, the output of* $\mathsf{EncSort}$ *is an array* $B = [E_{\mathsf{SS}}(v'_j)]_{j\in\{1,N\}}$ *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 $E_{\mathsf{SS}}(v'_j) \neq E_{\mathsf{SS}}(v_{\pi(i)})$ since fresh randomness is used during re-encryption. We note that Definition 1 can be easily expanded to take as input an array A that stores $(\mathsf{key}, \mathsf{value})$ 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 [15].

Definition 2. *(*$\mathsf{EncSort}$ *Privacy) Let* Π_{EncSort} *be a two party protocol for computing* $\mathsf{EncSort}$ *functionality.* S_1 *takes as input* (PK, A) *and* S_2 *takes as input* $(\mathsf{PK}, \mathsf{SK})$*. When* Π_{EncSort} *terminates* S_1 *receives the output* B *of* $\mathsf{EncSort}$*. Let* $\mathsf{VIEW}_{S_i}^{\Pi_{\mathsf{EncSort}}}(\mathsf{PK}, \mathsf{SK}, A)$ *be all the messages that* S_i *receives while running the protocol on inputs* $\mathsf{PK}, \mathsf{SK}, A$ *and* $\mathsf{OUTPUT}^{\Pi_{\mathsf{EncSort}}}$ *be the output of the protocol received by* S_1*.*

We say that Π_{EncSort} *privately computes* $\mathsf{EncSort}$*, i.e.,* Π_{EncSort} *is a private outsourced sort, if there exists a pair of probabilistic polynomial time (PPT) simulators* $(\mathsf{Sim}_{S_1}, \mathsf{Sim}_{S_2})$ *such that*

(1) $(\mathsf{Sim}_{S_2}(\mathsf{PK}, A), \mathsf{EncSort}(\mathsf{PK}, \mathsf{SK}, A)) \approxeq$
$$(\mathsf{VIEW}_{S_1}^{\Pi_{\mathsf{EncSort}}}(\mathsf{PK}, \mathsf{SK}, A), \mathsf{OUTPUT}^{\Pi_{\mathsf{EncSort}}}(\mathsf{PK}, \mathsf{SK}, A));$$

(2) $\mathsf{Sim}_{S_1}(\mathsf{PK}, \mathsf{SK}, N) \approxeq \mathsf{VIEW}_{S_2}^{\Pi_{\mathsf{EncSort}}}(\mathsf{PK}, \mathsf{SK}, A)$,

[6] We note that one should not confuse our problem with Multi-Party Computation protocols for sorting [16, 17], where every party has an input array and the goal is to output to every participating party the sorting of all inputs combined.

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

The intuition behind the privacy definition of $\mathsf{EncSort}$ 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. VIEW_{S_i} captures messages that S_i receives while participating in Π^{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, Sim_{S_2}. Sim_{S_2} knows exactly what is known to S_1 and nothing more. The main property of Sim_{S_2} is that S_1 should not be able to distinguish if he is interacting with 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 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 $\mathsf{EncSort}(\mathsf{PK}, \mathsf{SK}, A)$ presented in Definition 1. 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 $(\mathsf{key}, \mathsf{value})$ 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, $\mathsf{EncPairSort}$, by the end of which S_1 has a pair of values $([c], [d])$ such that $(c, d) = (a, b)$ if $a \leq b$ and $(c, d) = (b, a)$, otherwise. Informally, $\mathsf{EncPairSort}$ has the following privacy guarantees. S_1 and S_2 should learn nothing about values a and b nor their sorted order. The formal definition of $\mathsf{EncPairSort}$ is a special case of $\mathsf{EncSort}$ in Definition 1 with $N = 2$.

The $\mathsf{EncPairSort}$ makes use of the comparison protocol $\mathsf{EncCompare}$ from Section 2.1 to help S_1 to acquire an encryption of the bit v that denotes whether

Table 4: $([c],[d]) \leftarrow \mathsf{EncPairSort}(\mathsf{PK_P}, \mathsf{SK_P}, [a], [b])$: Interactive protocol between S_1 and S_2 for sorting two encrypted elements such that only S_1 receives the result. The key pair for Paillier cryptosystem is denoted as $K_\mathsf{P} = (\mathsf{PK_P}, \mathsf{SK_P})$.

$S_1(\mathsf{PK_P}, [a], [b])$	$S_2(\mathsf{PK_P}, \mathsf{SK_P})$	
$[\![v]\!] \leftarrow \mathsf{EncCompare}(K_\mathsf{P}, [a], [b])$		% Compare a, b: $v := a \geq b$.
$[\![[c]]\!] \leftarrow \mathsf{EncSelect}(K_\mathsf{P}, [a], [b], [\![v]\!])$		% $c := (1-v)a + vb$.
$[\![[d]]\!] \leftarrow \mathsf{EncSelect}(K_\mathsf{P}, [a], [b], [\![1]\!][\![v]\!]^{-1})$		% $d := va + (1-v)b$.
$[c] \leftarrow \mathsf{StripEnc}(K_\mathsf{P}, [\![[c]]\!])$		% Strip a layer of encryption
$[d] \leftarrow \mathsf{StripEnc}(K_\mathsf{P}, [\![[d]]\!])$		% for c and d.

$a \geq b$ or not. Given a Paillier encryption of v we can then use a ciphertext selection EncSelect from Section 2.1 to blindly swap a and b according to v, i.e., their sorted order. The last step of the protocol brings the encryption of swapped a and b back to the first layer of Paillier. The complete protocol EncPairSort is shown in Table 4.

Theorem 1. *The* EncPairSort *protocol in Table 4 is a private outsourced sorting protocol according to Definition 2 for the case* $N = 2$.

Proof. (Sketch) In order to show that EncPairSort in Table 4 is secure according to Definition 2 we need to construct two simulators Sim_{S_1} and 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 Sim_{S_2} as follows. Sim_{S_2} has access to private inputs of S_1 in the protocol. The VIEW of S_1 consists of VIEW's from EncCompare and two invocations of EncSelect and StripEnc protocols. In the full version [4] we show that there exist simulators for each of these functionalities. Then Sim_{S_2} for EncPairSort simply invokes EncCompare simulator once, and EncSelect and StripEnc simulators twice each. The construction of Sim_{S_1} is symmetrical.

General Sort In the previous section we developed an interactive method EncPairSort for blindly sorting two elements (Table 4). In this section, we use EncPairSort as a blackbox to build a protocol EncSort for privately sorting N elements according to Definition 2. 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

Table 5: $B \leftarrow \mathsf{EncSort}(\mathsf{PK_P}, \mathsf{SK_P}, A)$: 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 2). Paillier key pair is denoted using $K_\mathsf{P} = (\mathsf{PK_P}, \mathsf{SK_P})$. See Figure 2 for an illustration for the case when $N = 4$.

$S_1(\mathsf{PK_P}, A)$	$S_2(\mathsf{PK_P}, \mathsf{SK_P})$	
$A_1 \leftarrow A$		
for $i \in \{1, \ldots, k-1\}$		% $k = O((\log N)^2), N = \lvert A \rvert$.
$(x, y) \leftarrow \mathsf{pairs}_i.\mathsf{next}$		% ith level of Batcher's sort.
while$((x, y) \neq \perp)$		
$(A_{i+1}\{x\}, A_{i+1}\{y\}) \leftarrow \mathsf{EncPairSort}(K_\mathsf{P}, A_i\{x\}, A_i\{y\})$	$\longrightarrow$	% Sort x, y entries of A_i.
	$\longleftarrow$	
$(x, y) \leftarrow \mathsf{pairs}_i.\mathsf{next}$		% Next pair of A_i to sort.
$B \leftarrow A_k$		

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 [2] 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 $O(N(\log N)^2)$ 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_1 is the input array, where $A_i\{j\}$ 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 $A_1\{0\}$, $A_1\{1\}$ in sorted order, $A_2\{2\}$ and $A_2\{3\}$ contain $A_1\{2\}$, $A_1\{3\}$ 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 $\mathsf{pairs}_i.\mathsf{next}$ returns the next pair to be sorted.

In Table 5 we present our protocol $\mathsf{EncSort}$ 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 $\mathsf{EncPairSort}$. Recall that the output of $\mathsf{EncPairSort}$ 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 $\mathsf{EncPairSort}$ in the next level $i+1$. (See Figure 2 for an illustration of $\mathsf{EncSort}$ 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.

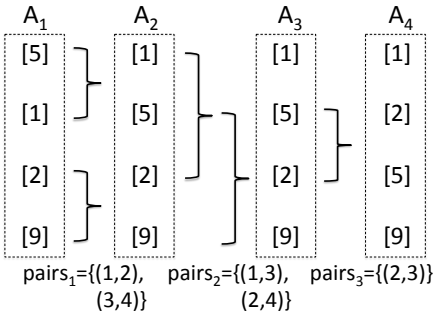

Fig. 2: Example of EncSort protocol in Table 5 for sorting an encrypted array of four elements $5, 1, 2, 9$ where $[m]$ denotes a Paillier encryption of message m and pairs_i denotes a pair of elements to be sorted. Note that only S_1 stores values in the arrays A_i while S_2 blindly assists S_1 in sorting the values.

Theorem 2 (EncSort Privacy). *The* EncSort *protocol in Table 5 is a private outsourced sorting protocol according to Definition 2.*

Proof. (Sketch) The protocol EncSort in Table 5 makes $O(N(\log N)^2)$ calls to EncPairSort protocol in Table 4. In Theorem 1 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 3 (EncSort Performance). *The* EncSort *protocol in Table 5 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 $O(N(\log N)^2)$;*
- *The communication complexity between S_1 and S_2 consists of $O(N(\log N)^2)$ rounds;*
- *If S_2 has $O(1)$ storage, the time overhead of the protocol is $O(N(\log N)^2)$;*
- *If 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_i and A_{i+1} since after finishing the ith 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 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 (key, value) pair and value is used to sort the array. The main alternations happen in EncPairSort protocol where the input is not two ciphertexts as in Table 4 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 values EncCompare is called only on $[v_1]$ and $[v_2]$. Once the bit representing the result of the comparison is computed, 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. There, the key is an encrypted document id and value is an encryption of the corresponding score.

4 Private Ranked Search (MRSE)

We now give an overview 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, using our *private sort* protocol.

4.1 MRSE Security Model

MRSE builds on the SSE model [11] where a server, given an encrypted document collection and the corresponding secure search index, while answering multiple search queries, should not be able to deduct 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 query keyword, while the *search pattern* refers to any connection between queries that the server may derive (e.g., if specific query terms have been queried before and how many times). The MRSE model assumes that it is sufficient to return to the client only document identifiers and not the actual document. This is consistent with related work on searchable encryption with ranking, e.g., [9].

Our setting consists of two servers S_1 and S_2, where the user queries only S_1. S_1 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. We examine the privacy of the client against S_1 and S_2 separately. 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 2.

Informally, MRSE privacy definitions 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 [11] while S_2 learns much less. The formal definitions can be found in [4].

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, $\mathsf{SK}_{\mathsf{StE}}$, and a public/secret key pair for Paillier cryptosystem (PK_{P}, SK_{P}). Then, shares PK_{P} with S_1 and ($\mathsf{PK}_{\mathsf{P}}, \mathsf{SK}_{\mathsf{P}}$) 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 sort protocol from Section 3.2.

The client first extracts all M unique terms[7] from his collection of documents $\mathbf{D}$ 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., tf-idf$_{t,d}$ as defined in Section 2.3. Note that tf-idf$_{t,d}$ is zero if the term t does not appear in document d. Given all tf-idf$_{t,d}$ entries the client obtains the frequency table F where the number of rows is M (number of unique terms in $\mathbf{D}$) and the number of columns is N (number of documents in $\mathbf{D}$). 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 $[F_t] = \{[\text{tf-idf}_{t,d}] \mid \forall d \in \{1, N\}\}$.

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 2.2. 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 $[F_t]$ 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 [10]. Let $\mathsf{SK}_{\mathsf{StE}}$ 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 π and creates an auxiliary index of pairs $(t_i, \pi(i))$ $\forall i \in \{1, M\}$. He also appends $\pi(i)$ to the corresponding $[F_{t_i}]$ and permutes the pairs $(\pi(i), [F_{t_i}])$, i.e., creates a dictionary that maps a keyword t_i to a list of encrypted scores

[7] Stemming and removal of stop words is outside of the scope of our paper.

for all the documents in the collection (not only the documents in which the keyword appears at).

Then, the encryption algorithm of StE, E_{StE}, works as follows. For every $i \in \{1, M\}$, the *search key* $k_{t_i} = G'_{K_2}(t_i)$ and the value $(\pi(i), [F_{t_i}]) \oplus G_{K_1}(t_i)$ 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.

Query Phase During the query phase, the client computes the trapdoor $T \leftarrow$ *Trpdr*$(\mathsf{SK}_{\mathsf{StE}}, t)$ for each keyword t in the query q. In our scheme, *Trpdr* sets T to $(G_{K_1}(t), G'_{K_2}(t))$. The client then sends the trapdoors of all the query terms (i.e., an "encrypted" representation of the query) $\mathbf{T} = \{T \mid \forall t \in q\}$ to S_1. Server S_1, upon receiving client's query $\mathbf{T}$, can locate each encrypted keyword $t \in q$ using the corresponding trapdoors by running *Search*(I, T) $\forall T \in \mathbf{T}$. The *Search* algorithm parses T as (α, β) and computes the answer as $I(\beta) \oplus \alpha$, where $I(\beta)$ is the value stored in I under the *search key* β. The answer is a vector $[F_t] = \{[\text{tf-idf}_{t,d}] \mid \forall d \in \{1, N\}\}$ for every term t in the query.
Computing Document Scores: Recall that $[F_t]$ 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 $e_d = [\text{score}(q, d)] = \sum_{t \in q} [\text{tf-idf}_{t,d}]$. Note that e_d is simply an encryption of score(q, d). S_1 then creates an array A of $(\mathsf{key}, \mathsf{value})$ 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 $\mathsf{EncSort}$ defined in Table 5 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.

Finally, S_1 sends to the client array B. According to client preference, S_1 can send document identifiers with scores, omit scores, or send only the top k results. The client has the Paillier decryption key and can easily decrypt the ordered sequence of document identifiers (and scores) received from S_1.

4.3 MRSE Analysis

Here, we give an informal analysis of why MRSE is secure against S_1 and S_2 and refer the reader to [4] for a proof. The client's document collection $\mathbf{D}$ and the scores are represented as the encrypted index I which is stored semantically encrypted with S_1 only. The client encrypts frequency scores for all documents and unique terms in F, including zeroes, hence, S_1 does not learn anything

about the collection except the number of documents N and the number of unique terms M in $\mathbf{D}$. The client sends his queries to S_1 and, hence, S_1 learns the search pattern, i.e., if the keywords were queried before or not. Note that the search pattern is also leaked in the original StE scheme since StE generates a deterministic trapdoor for the same term.

The security properties of private sort in Definition 2 guarantee that as long as S_1 and S_2 behave honestly neither one learns anything about the array of document scores they are sorting. Since S_2 is invoked to participate in private sort he learns N, the number of documents in the collection, and that a client has queried S_1 but learns nothing more about query keywords or query length. The performance of MRSE is summarized in the following theorem.

Theorem 4 (MRSE Performance). *MRSE protocol presented in Section 4.2 gives the following performance guarantees:*

- *The client takes $O(N \times M)$ 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 N is the number of documents and M is the number of unique terms in the collection, and $|q|$ is the query size.

A proof of Theorem 4 can be found in the full version [4].

4.4 Comparison with Related Work

In this section we compare MRSE with other multi-keyword searchable encryption schemes with ranked results. Cao *et al.* [9] 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 [9] 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 [23, 24] also propose protocols for ranked multi-keyword search. Their ranking is loosely based on frequency of a word in the document

Table 6: Comparison of MRSE with multi-keyword searchable encryption schemes returning ranked results in terms of **sound**ness of the result, the **ranking** technique, the **client** query generation time, **server(s)** time to compute the result and the **privacy** guarantees. We note that schemes [9] and [27] are single server solutions. Inner product similarity is denoted as ips, N is the number of documents and M is the number of unique terms in the collection, $|q|$ is the query length, * denotes the use of FHE techniques, CCA-2 is security against chosen ciphertext attack for SSE schemes [11]. All the time complexities are asymptotic.

Scheme	**Sound**	**Ranking**	**Client**	**Server(s)**	**Privacy**
Cao *et al.* [9]		ips	M^2	$N \times M^2$	Precision-privacy tradeoff
Örencik *et al.* [22]	✓	tf-idf	$\|q\|$	$N \log N$	CCA-2 v.s. S_1, but not S_2
Strizhov-Ray [27] *	✓	ips†	$\|q\|N$	$\|q\|N + M^2$	CCA-2
Our scheme MRSE	✓	tf-idf	$\|q\|$	$N(\log N)^2$	CCA-2 v.s. S_1 and $S_2^\ddagger$

† 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).

where fake keywords and documents are added, hence, their scheme also may return false negatives and positives. Recent proposal by Örencik *et al.* [22] 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 [27]. 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 6 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 [9] is harder to compare with since a heuristic is used to hide access and search patterns.

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 by the Center for Reliable Information Systems and Cyber Security (RISCS) and grant CNS–1012910 (Boston University), and also in part by the FINER project by Greek Secretariat of Research Technology (University of Athens) and CNS–0964379 (Brown University).

References

1. Ben Adida and Douglas Wikström. How to shuffle in public. In *Conference on Theory of Cryptography*, TCC'07, pages 555–574. Springer-Verlag, 2007.
2. Miklós Ajtai, János Komlós, and Endre Szemerédi. An O(n log n) sorting network. In *ACM Symposium on Theory of Computing*, STOC'83, pages 1–9. ACM, 1983.
3. Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *ACM conference on Computer and communications security*, CCS '13, pages 535–548. ACM, 2013.
4. Foteini Baldimtsi and Olga Ohrimenko. Sorting and searching behind the curtain: Private outsourced sort and frequency-based ranking of search results over encrypted data. Cryptology ePrint Archive, Report 2014/1017, 2014.
5. Kenneth E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, 1968.
6. Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *IEEE Symposium on Security and Privacy*, SP '13, pages 478–492. IEEE, 2013.
7. Alexandra Boldyreva, Nathan Chenette, and Adam O'Neill. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *Conference on Advances in Cryptology*, CRYPTO'11, pages 578–595. Springer-Verlag, 2011.
8. Raphael Bost, Raluca Ada Popa, Stephen Tu, and Shafi Goldwasser. Machine learning classification over encrypted data. Cryptology ePrint Archive, Report 2014/331, 2014.
9. Ning Cao, Cong Wang, Ming Li, Kui Ren, and Wenjing Lou. Privacy-preserving multi-keyword ranked search over encrypted cloud data. In *Conference on Information Communications*, INFOCOM'11, pages 829–837. IEEE, 2011.
10. Melissa Chase and Seny Kamara. Structured encryption and controlled disclosure. In *Advances in Cryptology - ASIACRYPT 2010*, volume 6477 of *Lecture Notes in Computer Science*, pages 577–594. Springer Berlin Heidelberg, 2010.
11. Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *ACM conference on Computer and communications security*, CCS '06, pages 79–88. ACM, 2006.
12. Ivan Damgard, Martin Geisler, and Mikkel Kroigard. A correction to efficient and secure comparison for on-line auctions. *Int. J. Appl. Cryptol.*, 1(4):323–324, 2009.

13. Ivan Damgård and Mats Jurik. A generalisation, a simplification and some applications of paillier's probabilistic public-key system. In *International Workshop on Practice and Theory in Public Key Cryptography*, PKC '01, pages 119–136. Springer-Verlag, 2001.
14. Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. `crypto.stanford.edu/craig`.
15. Oded Goldreich. *Foundations of Cryptography, vol. 2*. Cambridge University Press, 2001.
16. Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS*, 2012.
17. Kristján Valur Jónsson, Gunnar Kreitz, and Misbah Uddin. Secure multi-party sorting and applications. In *Applied Cryptography and Network Security*, ACNS'11, 2011.
18. Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., 1998.
19. Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. Improved garbled circuit building blocks and applications to auctions and computing minima. In *Cryptology and Network Security*, volume 5888 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin Heidelberg, 2009.
20. Helger Lipmaa. An oblivious transfer protocol with log-squared communication. In *Proceedings of the 8th International Conference on Information Security*, ISC'05, pages 314–328. Springer-Verlag, 2005.
21. Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*. Cambridge University Press, 2008.
22. Cengiz Örencik, Murat Kantarcioglu, and Erkay Savaş. A practical and secure multi-keyword search method over encrypted cloud data. In *International Conference on Cloud Computing*, CLOUD '13, pages 390–397. IEEE, 2013.
23. Cengiz Örencik and Erkay Savaş. Efficient and secure ranked multi-keyword search on encrypted cloud data. In *Proceedings of the 2012 Joint EDBT/ICDT Workshops*, EDBT-ICDT '12, pages 186–195. ACM, 2012.
24. Cengiz Örencik and Erkay Savaş. An efficient privacy-preserving multi-keyword search over encrypted cloud data with ranking. *Distributed and Parallel Databases*, 2014.
25. Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *International Conference on Theory and Applications of Cryptographic Techniques*, EUROCRYPT'99, pages 223–238. Springer-Verlag, 1999.
26. Ronald L. Rivest, Len Adleman, and Michael L. Dertouzos. On data banks and privacy homomorphisms. In *Foundations of Secure Computation*, pages 169–177. Academic Press, 1978.
27. Mikhail Strizhov and Indrajit Ray. Multi-keyword similarity search over encrypted cloud data. In *ICT Systems Security and Privacy Protection*, volume 428, pages 52–65, 2014.
28. Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In *International Conference on Theory and Applications of Cryptographic Techniques*, EUROCRYPT'10, pages 24–43. Springer-Verlag, 2010.
29. Thijs Veugen. Comparing encrypted data. Manuscript, 2010. `http://isplab.tudelft.nl/sites/default/files/Comparingencrypteddata.pdf`.
30. Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), 2006.