REX: A Searchable Symmetric Encryption Scheme Supporting Range Queries

Panagiotis Rizomiliotis
Dep. of Inf. and Comm. Systems Eng.
University of the Aegean
Karlovassi, Greece
prizomil@aegean.gr

Eirini Molla
Dep. of Inf. and Comm. Systems Eng.
University of the Aegean
Karlovassi, Greece
eirinim@icsd.aegean.gr

Stefanos Gritzalis
Dep. of Inf. and Comm. Systems Eng.
University of the Aegean
Karlovassi, Greece
sgritz@aegean.gr

ABSTRACT

Searchable Symmetric Encryption is a mechanism that facilitates search over encrypted data that are outsourced to an untrusted server. SSE schemes are practical as they trade nicely security for efficiency. However, the supported functionalities are mainly limited to single keyword queries.

In this paper, we present a new efficient SSE scheme, called REX, that supports range queries. REX is a no interactive (single round) and response-hiding scheme. It has optimal communication and search computation complexity, while it is much more secure than traditional Order Preserving Encryption based range SSE schemes.

CCS CONCEPTS

• **Security and privacy → Management and querying of encrypted data**;

KEYWORDS

encrypted data, searchable encryption, secure computation, range query

1 INTRODUCTION

Searchable Symmetric Encryption (SSE) was introduced in 2000 by Song et al. [26] as a mechanism to facilitate search over encrypted data (either a database or a collection of files) that are outsourced to an untrusted server. While we have several techniques for securly querying encrypted data, like multiparty computation schemes ([27]), Oblivious RAM algorithms ([11]), or fully homomorphic encryption ([10]), all these solutions are still impractical. On the other hand, SSE schemes nicely trade security for efficiency and for that it has gained a lot of attention.

The proposed schemes are practical (or almost practical) and at the same time they are supported by a security proof. What the proof guarantees is that there is a well defined upper bound of the information leaked. The weakest schemes assume that the attacker is allowed to send a batch of queries and only once (non-adaptive security). In the most realistic model, the attacker can choose new queries after receiving the server's response to previous ones (adaptive security).

SSE schemes possess a plethora of characteristics. They can be static or dynamic, verifiable and parallelizable. The vast majority of the schemes support only single keyword queries and only a few proposals offer more complex functionalities, like conjunctive queries, wildcards, and substring.

Without doubt one of the most useful functionalities is the range query. That is, given two keywords ω_i and ω_j, the SSE must compute and return the data entries that have a keyword ω that satisfies the condition $\omega_i \leq \omega \leq \omega_j$. Very few SSE schemes, so far, have been proposed that support range queries and most of them are based on Order Preserving Encryption (OPE) schemes.

The adoption of SSE has raised some concerns regarding the level of security it can offer. While there is a good definition of the leaked information, it is still not clear how this leakage influences the scheme's security. Since Islam et al. 's work [14], several papers have been published demonstrating that the leaked information can be fatal for a SSE scheme ([28]). An interesting observation is that most of the attacks are against range query supporting schemes and especially the ones that are based on OPE algorithms ([15], [21], [22]).

In this paper, we present a new efficient SSE scheme that supports range queries. Our scheme is called REX and it does not rely on OPE algorithms. We show, somehow surprisingly, that it has almost the same complexity as a single keyword SSE scheme.

1.1 Our Contribution

We present REX, a range query SSE scheme. REX is very efficient, not interactive (single round) and response-hiding (i.e. the scheme does not reveal the response to queries). It has optimal communication and search computation complexity. Also, it is much more secure than any OPE based scheme, as it uses semantically secure encryption to protect confidentiality. We analyze its security in terms of information leakage profiles. Finally, the presented version of REX is static, however, we elaborate on the required amendments to support updates as well.

REX builds on most of the existing single keyword SSE schemes and we show that supporting range queries can come with minimum cost. We take advantage of the following observation. The vast majority of the single keyword SSE schemes are based on an inverted index, and in all these schemes the order that the identifiers are stored is random and thus we cannot exploit further the structure. In this paper, we demonstrate, somehow surprisingly, that the order of the stored identifiers can be used to support efficiently range queries.

CCSW'17, November 3, 2017, Dallas, TX,USA
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5204-8/17/11...$15.00
https://doi.org/10.1145/3140649.3140653

In a few words, REX is a sequence of simple steps. First, we introduce a new data structure called R. The client re-structures the inverted index DB and computes R. R can be seen as a matrix and each row of R is stored as a list of the file identifiers of a specific keyword ω. The client outsources R to the server using two dictionaries. At the first dictionary the rows of R are stored encrypted. It can be used to answer single keyword queries. At the second dictionary an index on R is stored. This is used to answer range queries as a partial multi-keyword search. The server uses a constant size token sent by the client. With the index stored at D_1 the server locates the adequate parts of the row lists at D_0 and sends the encrypted file identifiers. Finally, the client decrypts the received data and retrieves the answer. The scheme is response-hiding.

Finally, we have implemented and evaluated REX. Our scheme has reasonable client storage requirements comparable to secure single keyword SSE, like [3], and practical average search time per matching file identifier returned.

1.2 Related Work

Searchable Symmetric Encryption was introduced by Song et al. ([26]) in 2000 with a scheme that has linear in N search time, where N is the total number of keyword/file identifier pairs. The first sublinear and index-based solution was proposed by Curtmola et al. ([6]). This scheme was static. The first sublinear dynamic scheme was presented by Kamara et al. in [19] and later was improved in [18]. Several index based schemes have been proposed improving either the security ([25], [3]) or the functionality (for instance, [5] and [17] support Boolean queries) or the efficiency ([18], [13]).

Secure range search has gained a lot of attention by the database research community, but all these proposals lack provable security guarantees and usually they accept unreasonable information leakage.

Schemes that support range queries can be based on Order Preserving Encryption (OPE). OPE have the nice property of preserving the order of the plaintext, i.e. if $p_1 < p_2$, then $c_1 < c_2$, where c_1 and c_2 are the ciphertexts of p_1 and p_2, respectively. Thus, we do not need to modify existing database management systems, as it is possible to build traditional indexes efficiently on the encrypted data and to query in the same way as for the plaintexts.

Boldyreva et al. ([2]) have proposed the first concrete implementation of OPE. However, this first work was later proved weak, as half of the plaintext was leaked. More recent OPE by Popa et al. ([23]) and Boelter et al. ([1]) have improved OPE security. The practicality of OPE schemes was demonstrated by implementations, like CryptDb [24] (that uses Boldyreva et al.'s proposal) and the most recent ArX ([23]) that builds on [1]. However, OPE's security is still debatable. A series of recent works have shown that OPE schemes' main property, i.e. the order preservation, can lead to very efficient attacks.

Another line of work is to use SSE for exact pattern matching. A range query can be seen as multi-keyword search ([7]) by expressing the sub-ranges of the range to index nodes. A similar approach is used in [9].

Generally, the security provided by the encrypted outsourced data is poorly understood. Recent works, focusing on some practical solutions, have demonstrated that confidentiality can be compromised given auxiliary information on the data and using the system's leakage ([14], [15], [4], [8], [22], [28], [12]). Most of the attacks take advantage of well-known vulnerabilities of weak, but efficient, primitives that the systems use, like deterministic encryption and order preserving (or revealing) encryption. In [20], abstract attack models have been proposed, independent of the details of the system.

1.3 Paper Outline

The paper is organized as follows. In Section 2, some notation is defined. Also, we provide scheme and security definition. In Section 3, we introduce REX. First, we describe the main data structure and we establish some basic properties. Then, we propose a scheme that takes advantage of this data structure and we explain how it works with several examples. We also analyze its complexity, security and correctness. In Section 4, we evaluate an implementation of REX and in Section 5, we present our current line of work on extending REX to support updates. We call the new version DREX.

2 PRELIMINARIES AND DEFINITIONS

Notation. Next, we provide some notation. Let R be a $n \times m$ matrix. We use $R[:, j]$ and $R[i, :]$ to denote the j-th column and the i-th row of R respectively, and $R[i_1 : i_2, j_1 : j_2]$ is the submatrix of R that consists of the rows from i_1 to i_2 and the columns from j_1 to j_2, for $0 \le i_1 \le i_2 < n$ and $0 \le j_1 \le j_2 < m$. We define a special submatrix of R, as

$$R^{(i,j)} = R[i : j, 0 : i]$$

and we denote by $\cup R^{(i,j)}$ the union of all entries of $R^{(i,j)}$, for $0 \le i < n$ and $0 \le i < j < m$. That is,

$$\cup R^{(i,j)} = \cup_{i \le i' \le j,\ 0 \le j' \le i} R[i, j].$$

Example 2.1. Let the matrix R

$$R = \begin{pmatrix} 12 & 3 & 4 & 8 \\ 5 & 2 & 9 & 10 \\ 3 & 4 & 2 & 11 \\ 6 & 15 & 12 & 13 \\ 16 & 115 & 112 & 23 \end{pmatrix}.$$

Then, the submatrix $R^{(1,3)}$ is given by

$$R^{(1,3)} = \begin{pmatrix} 5 & 2 \\ 3 & 4 \\ 6 & 15 \end{pmatrix}$$

and

$$\begin{aligned} \cup R^{(1,3)} &= \cup_{1 \le i \le 3,\ 0 \le j \le 1} R[i, j] \\ &= \{5, 2, 3, 4, 6, 15\}. \end{aligned}$$

□

Finally, the output x of an algorithm $\mathcal{A}$ is denoted as $x \leftarrow \mathcal{A}$ and H^i is the composition i times of the function H, i.e $H^i(\cdot) = H(H(\ldots H(H(\cdot)) \cdots))$.

The Scheme's Syntax. Let λ be the security parameter. We will follow the notation that it is common in almost all the previous works on SSE. Let $\mathbf{W}$ and $\mathbf{F}$ be the set of keywords and file identifiers, respectively, and let $|\mathbf{W}|$ and $|\mathbf{F}|$ denote their cardinality, i.e. $\mathbf{F} = \cup_{i=0}^{|\mathbf{F}|-1} id_i$ and $\mathbf{W} = \cup_{i=0}^{|\mathbf{W}|-1} \omega_i$. The file identifiers and the keywords will be bit strings, and N denotes the total number of keyword/identifier pairs.

A database DB is a list of file identifier and keyword pairs (id_i, ω_j), for $id_i \in \mathbf{F}$ and $\omega_j \in \mathbf{W}$. We write $DB(\omega)$ for the set of all identifiers of files that contain ω. Similarly, we denote by

$$DB(\omega_i, \omega_j) = \cup_{\omega_i \leq \omega \leq \omega_j} DB(\omega)$$

the set of all identifiers of files that contain a keyword $\omega_i \leq \omega \leq \omega_j$, i.e. the set of identifiers that satisfies a range query. Without loss of generality, for the rest of the paper, we assume that $\omega_i \leq \omega_j$, for $0 \leq i \leq j < |\mathbf{W}|$, and for all the keywords $\omega_i, \omega_j \in \mathbf{W}$.

Cryptographic Primitives and Data Structures. We use several data structures, namely multi-maps, lists and dictionaries. Also, we use a semantically secure secret-key encryption scheme $SKE = (Gen, Enc, Dec)$, where Gen is the key generation processes, Enc is the probabilistic encryption and Dec the deterministic decryption algorithm. Finally, we use two hash functions H_1 and H_2 with outputs μ and ψ, respectively. We only need the pre-image property.

2.1 Definitions

Definition 2.2. (RS-Range SSE): A single-round response-hiding range SSE scheme $\Sigma_R = (Setup, Token, Search, Decrypt)$ consists of four polynomial-time algorithms that work as follows:

- $(S, EDR) \leftarrow Setup(1^\lambda, DB)$: is a probabilistic algorithm that takes as input a security parameter 1^λ and a database DB of keyword and file identifier pairs and outputs a secret local state S and an encrypted structure EDR.
- $tk \leftarrow Token(S, q)$: is a (possibly) probabilistic algorithm that takes as input the secret local state S and a range query q and outputs a token tk.
- $msg \leftarrow Search(tk, EDR)$: is a (possibly) probabilistic algorithm that takes as input the encrypted structure EDR and a token tk and outputs a message msg.
- $r \leftarrow Decrypt(S, msg)$: is a deterministic algorithm that takes as input a secret local state S and a message msg and outputs a response r.

Correctness. We say that a range SSE scheme Σ_R is correct if, $\forall \lambda \in \mathbb{N}$, for all poly($\lambda$)-size databases DB, for all (S, EDR) output by $Setup(1^\lambda, DB)$ and all sequences of $m = poly(\lambda)$ queries $q_1, \cdots, q_m$, for all the produced tokens tk_i by $Token(S, q_i)$, and for all messages msg_i output by $Search(tk_i, EDR)$, $Decrypt(S, msg_i)$ returns the correct response r_i with all but negligible probability.

Security. We use the standard notion of security as was first formalized by Curtmola et al. in the context of searchable encryption [6]. The range SSE scheme guarantees that no information is revealed beyond the setup leakage $\mathcal{L}_S$, and the query leakage $\mathcal{L}_Q$ due to the search algorithm. If this holds when the queries are chosen adaptively, then we have adaptive security. Otherwise, we have non-adaptive security.

Definition 2.3. (Adaptive Security): Let $\Sigma_R = (Setup, Token, Search, Decrypt)$ be a response-hiding range SSE scheme and consider the probabilistic experiments where $\mathcal{A}$ is a stateful adversary, $\mathcal{S}$ is a stateful simulator, $\mathcal{L}_S$ and $\mathcal{L}_Q$ are the leakage profiles and $z \in \{0,1\}^*$:

- $\mathbf{Real}_{\Sigma,\mathcal{A}}(\lambda)$: given z the adversary $\mathcal{A}$ outputs a database DB and receives EDR from the challenger, where $(S, EDR) \leftarrow Setup(1^\lambda, DB)$. The adversary then adaptively chooses a polynomial number of queries $q_1, \cdots, q_m$. For all $1 \leq i \leq m$, the adversary receives $tk_i \leftarrow Token(S, q_i)$. Finally, $\mathcal{A}$ outputs a bit b that is output by the experiment.
- $\mathbf{Ideal}_{\Sigma,\mathcal{A},\mathcal{S}}(\lambda)$: given z the adversary $\mathcal{A}$ outputs a database DB that it sends to the challenger. Given z and the setup information leakage $\mathcal{L}_S(DB)$ from the challenger, the simulator $\mathcal{S}$ returns an encrypted data structure EDR to $\mathcal{A}$. The adversary then adaptively chooses a polynomial number of operations $q_1, \cdots, q_m$. For all $1 \leq i \leq m$, the simulator receives the search leakage $\mathcal{L}_Q(DB, q_i)$ and returns a token tk_i to $\mathcal{A}$. Finally, $\mathcal{A}$ outputs a bit b that is output by the experiment.

We say that Σ_R is adaptively $(\mathcal{L}_S, \mathcal{L}_Q)$-secure if for all polynomial time adversaries $\mathcal{A}$, there exists a polynomial time simulator $\mathcal{S}$, such that for all $z \in \{0,1\}^*$,

$$|Pr[\mathbf{Real}_{\Sigma,\mathcal{A}}(\lambda) = 1] - Pr[\mathbf{Ideal}_{\Sigma,\mathcal{A},\mathcal{S}}(\lambda) = 1] \leq negl(\lambda).$$

3 THE REX DESIGN

Without loss of generality, we assume that $\omega_i \leq \omega_j$, for $0 \leq i \leq j < |\mathbf{W}|$, and for all the keywords $\omega_i, \omega_j \in \mathbf{W}$. When only the upper bound is defined, i.e. $\omega \leq \omega_j$, we assume that the lower bound is the minimum keyword ω_0. That means that, the server must return the set $DB(\omega_0, \omega_j)$. Similarly, when only the lower bound is defined, the server returns the set $DB(\omega_i, \omega_{|\mathbf{W}|-1})$.

First, we describe a new data structure to support the range queries and then, we propose a protocol that implements the design.

3.1 The REX Data Structure

We define the following $|\mathbf{W}| \times |\mathbf{W}|$ matrix R. Each entry contains a tuple of file identifiers $f \in \mathbf{F}$ and initially the matrix is empty. Then, starting from the first column $R[:, 0]$, we fill in the matrix column by column as follows. For the j-th column, we leave the first $j-1$ entries empty, i.e.

$$R[i, j] = " - ", \ 0 \leq i \leq j - 1. \qquad (1)$$

The rest of the entries $R[i, j]$, for $i \geq j$, are completed with the maximum subset of the file identifiers $DB(\omega_i)$ that do not appear in $R^{(i,j)}$. That is, for a given column j and starting from $R[j, j]$, we fill in sequentially the entries $R[i, j]$ with all the file identifiers from $DB(\omega_i)$ that do no appear at the entries $R[i', j']$, for $j \leq i' \leq i$ and $0 \leq j' \leq j$.

Example 3.1. Let the set of file identifiers be

$$\{f_0, f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8\} \in \mathbf{F}$$

and the set of keywords be $\{\omega_0, \omega_1, \omega_2, \omega_3\} \in \mathbf{W}$. The entries of DB are $DB(\omega_0) = \{f_0, f_1, f_2, f_3\}$, $DB(\omega_1) = \{f_2, f_4, f_5\}$, $DB(\omega_2) =$

$\{f_0, f_3, f_4, f_8\}$ and $DB(\omega_3) = \{f_2, f_3, f_6, f_7\}$. The matrix R that corresponds to the above DB is:

$$R = \begin{pmatrix} f_0, f_1, f_2, f_3 & - & - & - \\ f_4, f_5 & f_2 & - & - \\ f_8 & f_0, f_3 & f_4 & - \\ f_6, f_7 & - & f_2 & f_3 \end{pmatrix} \tag{2}$$

Theorem 3.2. *The matrix R has the following properties:*

(1) *The matrix is lower triangular.*
(2) *All the entries of $R^{(i,j)}$ are distinct, for any $0 \leq i \leq j < |\mathbf{W}|$.*
(3) *The set $DB(\omega_i, \omega_j)$ equals*

$$DB(\omega_i, \omega_j) = \cup R^{(i,j)}. \tag{3}$$

The properties derive directly from the design of the matrix and we provide their proof in the Appendix.

Example 3.3. Let DB be as described in the Example 3.1. It is easy to verify that

$$\begin{aligned} DB(\omega_1, \omega_2) &= \{f_2, f_4, f_5, f_0, f_3, f_4, f_8\} \\ &= R[1,0] \cup R[1,1] \cup R[2,0] \cup R[2,1] \\ &= \cup_{1 \leq i' \leq 2,\ 0 \leq j' \leq 1} R[i,j] \\ &= \cup R^{(1,2)} \end{aligned}$$

where

$$R^{(1,2)} = \begin{pmatrix} f_4, f_5 & f_2 \\ f_8 & f_0, f_3 \end{pmatrix}.$$

All the entries of $R^{(1,2)}$ are distinct.

□

Corollary 3.4. *All the entries of the i-th row of R are distinct and their union equals the set $DB(\omega_i)$.*

Proof. This corollary derives from Theorem 3.2 by setting $\omega_j = \omega_i$. □

Example 3.5. Let DB be as described in the Example 3.1. From (2), it is easy to verify that all the entries at each row are distinct and that $DB(\omega_0) = R[0,0]$, $DB(\omega_1) = R[1,0] \cup R[1,1] = \cup R^{(1,1)}$, etc.

Corollary 3.4 states that each row of R is just a structured version of the data set $DB(\omega_i)$. Also, from Theorem 3.2 and more precisely from relation (3), it follows that due to the proposed structure, it is easy to support range queries and compute the set of file identifiers $DB(\omega_i, \omega_j)$ by computing the union of all the tuples of the submatrix $R^{(i,j)}$ of R.

3.2 The REX Protocol

Next, we present the static version of REX. The details of the scheme appear in Fig. 1 and Fig. 2. In Section 5, we describe the dynamic version.

Main Idea. REX is based on the R matrix described in Section 3.1. Initially, the client organizes the inverted index DB as a R matrix. From (3), we have seen that it is easy to answer a range query and compute the set $DB(\omega_i, \omega_j)$ using R by returning all the tuples of the submatrix $R^{(i,j)}$, i.e. the first i columns from the i-th to the j-th row of R. Thus, the challenging part is to outsource R encrypted and construct tokens that facilitate the server to locate and return $R^{(i,j)}$ and nothing else.

```
−$Setup(1^\lambda, DB)$
(1) Initialize a multi-map $S$, two dictionaries $D_0$, and
    $D_1$, and two matrices $R$ and $tempR$;
(2) For $i = 0 : |\mathbf{W}| - 1$ do
      For $j = i : |\mathbf{W}| - 1$ do
        Set $R[i,j]$ equal to the maximum subset
          of $DB(\omega_i)$ that is not in $R^{(i,j)}$;
(3) compute $K \leftarrow SKE.Gen(1^\lambda)$;
(4) For $i = 0 : |\mathbf{W}| - 1$ do
      sample $r_i^{(0)} \leftarrow \{0,1\}^\lambda$;
      set $c = 0$
      For $j = i : 0$ do
        get $curr = c$;
        get $T \leftarrow R[i,j]$;
        if $T$ not empty
          for $z = 0 : |T| - 1$ do
            $c++$;
            $id \leftarrow T[z]$;
            $pos \leftarrow H_1^c(r_i^{(0)})$;
            $D_0[pos] \leftarrow SKE.Enc(K, id)$;
            if $z = 0$
              $tempos \leftarrow pos$;
          $tempR[i,j] \leftarrow [tempos, |DB(\omega_i)| - curr]$;
        else $tempR[i,j] \leftarrow []$;
(5) For $i = 0 : |\mathbf{W}| - 1$ do
      if $tempR[i,0]$ is empty,
        then $tempR[i,0] = [NULL, 0]$;
      For $j = 1 : i$ do
        if $tempR[i,j]$ is empty,
          then $tempR[i,j] = tempR[i,j-1]$;
(6) For $j = 0 : |\mathbf{W}| - 1$ do
      sample $[r_j^{(1)}, k_j^{(1)}] \leftarrow \{0,1\}^{\lambda \times \lambda}$;
      set $S(\omega_j) \leftarrow [r_j(1), k_j^{(2)}]$;
      For $i = |\mathbf{W}| - 1 : j$ do
        $pos \leftarrow H_1^{|\mathbf{W}|-i}(r_j^{(1)})$;
        $D_1[pos] \leftarrow tempR[i,j] \oplus H_2^{|\mathbf{W}|-i}(k_j^{(1)})$;
(7) set $EDR \leftarrow (D_0, D_1)$.
(8) Output $EDR$ to the server, and $S$, and $K$ to the
    client.
```

Figure 1: The Setup operation of the REX scheme.

The client outsources R encrypted and stored at two dictionaries D_0 and D_1. The first dictionary D_0 contains the elements of R, while the second dictionary D_1 is used as an index on D_0.

Each row of R is organized as a list $L_i^{(0)}$, for $0 \leq i < |\mathbf{W}|$. The order that the elements appear at $L_i^{(0)}$ must follow one condition: the elements from $R[i,j]$ must succeed the elements from $R[i,j']$, for $j < j'$. The relative order of appearance of the elements that come from the same tuple $R[i,j]$ can be random. Let $R[i,j_f]$ and $R[i,j_l]$ be the first and the last tuples, respectively, of the row $R[i,:]$

that are not empty. Following the above condition, the head of the list $L_i^{(0)}$ is one of the elements from $R[i, j_l]$, while the tail is one of the elements from $R[i, j_f]$. Note that since R is a diagonal matrix, it holds that $0 \leq j_f \leq j_l \leq i$.

Example 3.6. The four lists of R from (2) are given by:

$$\begin{array}{l||lllllllll} L_0^{(0)} & f_1 \leftarrow f_3 \leftarrow f_0 \leftarrow f_2 & & & & & \\ L_1^{(0)} & & f_4 \leftarrow f_5 & \leftarrow f_2 & & & \\ L_2^{(0)} & & f_8 & \leftarrow f_3 \leftarrow f_0 \leftarrow & f_4 & \\ L_3^{(0)} & & f_7 \leftarrow f_6 & \longleftarrow\!\!\!\!\longrightarrow & f_2 & \leftarrow f_3 \end{array} \quad (4)$$

Note that the elements of the first column of R appear after the elements of the second column, e.t.c. Also, the elements from the same tuple are placed in a random order. For instance, since all the elements of the first list $L_0^{(0)}$ come from the same tuple $R[0, 0]$ they are ordered randomly, without any restriction. On the other hand, regarding the third list $L_2^{(0)}$, f_8 must be its tail, f_4 its head, while f_3 and f_0 are interchangeable.

□

Each node element of the list $L_i^{(0)}$ is encrypted with a semantically secure SE scheme using the same secret key K and the ciphertext is stored at a logical location of D_0. This location $Lo_i^{(0)}(c)$ is derived from a row randomness $r_i^{(0)}$ and a counter c as

$$Lo_i^{(0)}(c) = H_1^c(r_i^{(0)}), \quad (5)$$

where H_1 is a pre-image resistant function. Thus, the head of $L_i^{(0)}$ is stored at $Lo_i^{(0)}(1) = H_1^1(r_i^{(0)})$, while the tail at $Lo_i^{(0)}(|DB(\omega_i)|) = H_1^{|DB(\omega_i)|}(r_i^{(0)})$. Since, REX is static and we do not add new data, we can use a keyed hash function to compute the locations of the elements.

From (5), it is easy to see that given the location Loc of the c'-th element of $L_i^{(0)}$, for $1 \leq c' \leq |DB(\omega_i)|$, the server can compute the logical locations of the rest of the list's nodes by successively applying H_1. That is, the locations $L_i^{(0)}(c)$, for $c' \leq c \leq |DB(\omega_i)|$, are computed as $L_i^{(0)}(c) = H_1^{c-c'}(Loc)$. The pre-image resistance of H_1 prohibits the server from traversing the list from the c'-th node towards the head.

Example 3.7. We use as an example the lists $L_1^{(0)}$ and $L_2^{(0)}$ from the Example 3.6. The row randomness is $r_1^{(0)}$ and $r_2^{(0)}$, respectively.

$L_1^{(0)}$	f_4	$\leftarrow$	f_5	$\leftarrow$	f_2
c	3		2		1
$Lo_1^{(0)}(c)$	$H_1^3(r_1^{(0)})$		$H_1^2(r_1^{(0)})$		$H_1(r_1^{(0)})$

$L_2^{(0)}$	f_8	$\leftarrow$	f_3	$\leftarrow$	f_0	$\leftarrow$	f_4
c	4		3		2		1
$Lo_2^{(0)}(c)$	$H_1^4(r_2^{(0)})$		$H_1^3(r_2^{(0)})$		$H_1^2(r_2^{(0)})$		$H_1(r_2^{(0)})$

The location $Loc = H_1^2(r_1^{(0)})$ of the second element f_5 of $L_1^{(0)}$ can be used to compute the location of f_4 as $H_1(Loc) = H_1(H_1^2(r_1^{(0)})) = H_1^3(r_1^{(0)})$. Similarly, the location $Loc = H_1^2(r_2^{(0)})$ of the second element f_0 of $L_2^{(0)}$ can be used to compute the location of the last two elements f_3 and f_8, as $H_1(Loc) = H_1(H_1^2(r_2^{(0)})) = H_1^3(r_2^{(0)})$ and $H_1^2(Loc) = H_1^2(H_1^2(r_2^{(0)})) = H_1^4(r_2^{(0)})$, respectively.

□

At the setup phase, the client uses a $|\mathbf{W}| \times |\mathbf{W}|$ matrix *tempR* to temporary store some of these logical locations. More precisely, per tuple $R[i, j]$, the logical location of the element that appears first at the list $L_i^{(0)}$ is stored at $tempR[i, j]$. When, $R[i, 0]$ is empty, then we set $tempR[i, 0] = [NULL, 0]$ and, for $j > 0$, when $R[i, j]$ is empty, then we set $tempR[i, j] = tempR[i, j - 1]$. Together with the location we store the number of elements that can be found until the tail of the list.

Since, by design, $tempR[i, j]$ contains the logical location of the first element from the tuple $R[i, j]$ that appears in the list $L_i^{(0)}$, by continuously applying the function H_1 to this location the server can compute the locations of the rest of the list's elements. That is, using (5), the server can locate all the nodes until the tail of the list, i.e. all the elements from the tuples $R[i, j']$, for $0 \leq j' \leq j$.

Example 3.8. For the matrix R from (2) and the lists from (4), *tempR* is given by:

$$\begin{pmatrix} (Lo_0^{(0)}(r_0^{(0)}), 4) & - & - & - \\ (Lo_1^{(0)}(r_1^{(0)}), 2) & (Lo_1^{(0)}(r_1^{(0)}), 3) & - & - \\ (Lo_2^{(0)}(r_2^{(0)}), 1) & (Lo_2^{(0)}(r_2^{(0)}), 3) & (Lo_2^{(0)}(r_2^{(0)}), 4) & - \\ (Lo_3^{(0)}(r_3^{(0)}), 2) & (Lo_3^{(0)}(r_3^{(0)}), 2) & (Lo_3^{(0)}(r_3^{(0)}), 3) & (Lo_3^{(0)}(r_3^{(0)}), 4) \end{pmatrix} \quad (6)$$

For instance, the entry $tempR[0, 0]$ contains the location $Lo_0^{(0)}(r_0^{(0)})$ of the first element of $R[0, 0]$ that appears at $L_0^{(0)}$, i.e. the location of f_2 and it is the fourth element counting from the tail. Thus, the locations of the rest of the nodes at D_0 are computed as $H_1(Lo_0^{(0)}(r_0^{(0)}))$, $H_1^2(Lo_0^{(0)}(r_0^{(0)}))$ and $H_1^3(Lo_0^{(0)}(r_0^{(0)}))$.

Similarly, the entry $tempR[2, 1]$ contains the logical location $Lo_2^{(0)}(r_2^{(0)})$ of f_0, the element of $R[2, 1]$ that appears first at the list $L_2^{(0)}$ and the number 3 as this is the third element counting from the tail. The locations of the other two elements are computed as $H_1(Lo_2^{(0)}(r_2^{(0)}))$ and $H_1^2(Lo_2^{(0)}(r_2^{(0)}))$. The server cannot calculate the location of f_4, the list's head.

Note that since $R[3, 1]$ is empty, $tempR[3, 1]$ is set equal to the previous non zero value, i.e. $tempR[3, 0]$.

□

The matrix *tempR* is also outsourced to the server. Each column of the matrix is organized as a list $L_i^{(1)}$, and each list has $|\mathbf{W}| - i$ nodes, for $0 \leq i < |\mathbf{W}|$, while the head of the list is located at the last row of *tempR*. Again, each node element of the list $L_i^{(1)}$ is encrypted and the ciphertext is stored at a logical location, this time of D_1. The logical location of the c-th node $Lo_i^{(1)}(c)$ derives from a column randomness $r_i^{(1)}$ and a counter c, as $Lo_i^{(1)}(c) = H_1^c(r_i^{(1)})$. Thus, the element at the head node of $L_i^{(1)}$ is stored at $Lo_i^{(1)}(1) = H_1^1(r_i^{(1)})$, and the tail at $Lo_i^{(1)}(i) = H_1^i(r_i^{(1)})$. The plaintext is masked by a value that is derived from the column key $k_i^{(1)}$ and the counter c, as $H_2^c(k_i^{(1)})$.

Example 3.9. For the matrix *tempR* from (6), the second column's list and the corresponding locations at D_1 are given by:

$L_1^{(1)}$	$(Lo_1(r_1^{(0)}), 3)$	$\leftarrow$	$(Lo_2(r_2^{(0)}), 3)$	$\leftarrow$	$(Lo_3(r_3^{(0)}), 2)$
c	3		2		1
$Lo_1^{(1)}(c)$	$H_1^3(r_1^{(1)})$		$H_1^2(r_1^{(1)})$		$H_1(r_1^{(1)})$

□

The client stores locally the randomness per column $r_i^{(1)}$ and the secret key per column $k_i^{(1)}$, as well as the secret key K used to encrypt the entries of D_0.

Given a range query of keywords $\omega_i \leq \omega \leq \omega_j$ the server computes the set $DB(\omega_i, \omega_j)$ by locating and returning all the entries of $R^{(i,j)}$. The client retrieves locally the randomness $r_i^{(1)}$ and the secret key $k_i^{(1)}$ of the list $L_i^{(1)}$. Then, the location and the secret key of $tempR[i, j]$ are computed and sent to the server as a search token. Using the token the server locates the rest of the elements of the list $L_i^{(1)}$ until its tail and reveals a node location at D_0 for each one of the lists $L_m^{(0)}$, for $i \leq m \leq j$. Then, the server locates all the elements of each of these lists from the revealed node location to the tail. The server returns all these encrypted elements of D_0 to the client and the client decrypts them locally to compile the set $DB(\omega_i, \omega_j)$.

Example 3.10. The client wants to submit the range query for $\omega_1 \leq \omega \leq \omega_2$ and compute the set $DB(\omega_1, \omega_2)$. First, she retrieves the randomness $r_1^{(1)}$ and the key $k_1^{(1)}$ of the list $L_1^{(1)}$. Then, from $r_1^{(1)}$, she computes $Loc = H_1^2(r_1^{(1)})$ for the location of the second element (see also Example 3.9) and the corresponding decryption key as $Key = H_2^2(k_1^{(1)})$ (we will elaborate on that later). The location Loc and the key Key are sent to the server as the search token.

The server uses Loc and the key Key to retrieve from D_1 the pair $(Lo_2(r_2^{(0)}), 3)$. Also, by applying H_1 to Loc and H_2 to Key, it computes $H_1^3(r_1^{(1)})$, the location of the next element of the list $L_1^{(1)}$ from D_1, decrypts it with the key $H_2(Key)$ and retrieves the next pair $(Lo_1(r_1^{(0)}), 3)$.

Then, from $(Lo_2(r_2^{(0)}), 3)$ we have that starting from the location $Lo_2(r_2^{(0)})$ at D_0 the server will return the content for 3 locations, i.e. for $Lo_2(r_2^{(0)})$, $H_1(Lo_2(r_2^{(0)}))$ and $H_1^2(Lo_2(r_2^{(0)}))$ (see also Example 3.7). Similarly, starting from $Lo_1(r_1^{(0)})$ the server computes the 3 logical locations of D_0 and retrieves their content. Then, it sends to the client the six ciphertexts, the client decrypts them with the symmetric key K and reveals the plaintexts f_4, f_5, f_2, f_8, f_0, and f_3 (see also Example 3.3). □

3.3 The REX Operations

Setup. The Setup algorithm takes the database DB and the security parameter λ as input. First, it computes the matrix R and generates the symmetric key K for the encryption of the file identifiers with the symmetric key semantically secure encryption scheme *SKE*. Then, it stores each row of R as a list at D_0 and produces the matrix *tempR*. Afterwards, it stores each column of R as a list at D_1, while the elements of the lists are masked by the output of H_2. The location and the masking value are computed using different randomness per columns. This randomness is stored at a multi-map S. The algorithm outputs the dictionaries D_0 and D_1 as the encrypted structure *EDR* and the multi-map S and the key K as the local state.

Token. The Token algorithm takes as input the local state of the client, the lower bound ω_i and the upper bound ω_j of the range. When there is no lower bound, it is assumed that the lower bound is the smallest of all the keywords ω_0. Similarly, in the absence of an upper bound, the maximum keyword $\omega_{|\mathbf{W}|-1}$ is used. The algorithm retrieves the i-th entry of the internal state S, i.e. the column randomness $r_i^{(1)}$ and the key $k_i^{(1)}$ of the list $L_i^{(1)}$ and computes the location Loc of the $(|\mathbf{W}| - j)$-th node of the list at D_1. It also computes the corresponding key needed to decrypt the ciphertext. The token tk consists of the location Loc, the key Key and the number of elements from this location until the end of the list.

Search. The Search algorithm takes as input the token tk and the two dictionaries stored at the server. The element of the list $L_j^{(1)}$ is retrieved from the location Loc at D_1 and it is decrypted with the key Key. By applying $counter$ times the function H_1 to the value Loc and H_2 to Key, the server can traverse the rest of the list $L_j^{(1)}$ and decrypt all the nodes. Each revealed plaintext has one location at D_0 that corresponds to a node from a different list. Again, by applying recursively the function H_1 to these locations the algorithm traverses these lists and finds the ciphertexts of the file identifiers. The algorithm returns these ciphertexts as its output message.

Decryption. The Decryption algorithm takes as input a set of ciphertexts from the server and the symmetric secret key K of the client. It decrypts the ciphertexts with the key and returns the plaintexts, i.e. the file identifiers, as its response.

3.4 Complexity and Correctness

Complexity. The asymptotic complexities of REX appear in Table 1. We distinguish three main types of complexity, the communication, the computation and the storage complexity.

The size of the token is the same for all the range queries. It is equal to the length of the sum of the counter $\log(|\mathbf{W}|)$ and the outputs of the two hash functions $\mu + \psi$. On the other hand, the server's reply message is exactly the set $DB(\omega_i, \omega_j)$ and nothing more. That is, the communication complexity is optimal and the Search operation is single round.

In order to compute the set $DB(\omega_i, \omega_j)$, the server accesses $j-i+1$ locations of D_1 to retrieve the rest of the list $l_i^{(1)}$, i.e. $j-i+1$ indexes (logical locations) of list nodes stored at D_0. Then, it accesses exactly $|DB(\omega_i, \omega_j)|$ locations of D_0, at least one from each of the $j - i + 1$ lists $l_z^{(0)}$, for $i \leq z \leq j$. That is, the search computation complexity is $O(DB(\omega_i, \omega_j))$ and it is also optimal.

The client must store $|\mathbf{W}|$ pairs of μ and ψ bits for the randomness and the column key, respectively. Finally, the server requires $O(\log(|\mathbf{F}|))$ bits for each file identifier's ciphertext and in total N such entries. Thus, D_0 requires $O(N \log(|\mathbf{F}|))$. Finally, for the dictionary D_1 the server has storage overhead $O(|\mathbf{W}|^2(\mu + \log(|\mathbf{F}|)))$.

Correctness. The correctness of REX is straightforward and derives from Theorem 3.2. Regarding the probability of collision among the D_0 and D_1 locations generated by H_1, it can be easily

```
−Token(S, ω_i, ω_j)
    (1) If $\omega_i$ is empty, then $i = 0$;
    (2) If $\omega_j$ is empty, then $j = |\mathbf{W}| - 1$
    (3) get $[r^{(1)}, k^{(1)}] \leftarrow S[\omega_i]$;
    (4) compute $pos \leftarrow H_1^{|\mathbf{W}|-1-j}(r^{(1)})$;
            $key \leftarrow H_2^{|\mathbf{W}|-j-1}(k^{(1)})$;
            $counter \leftarrow j - i + 1$;
    (5) set $Tk \leftarrow [pos, key, counter]$;
    (6) Output $Tk$;

−Search(Tk, EDR)
    (1) parse $[pos, key, counter] \leftarrow Tk$;
    (2) empty $msg$;
    (3) For $j = 0 : counter - 1$ do
            $[loc, c] \leftarrow D_1[H_1^j[pos]] + H_2^j(key)$;
            for $i = 0 : c - 1$ do
                $msg = msg \cup D_0[H_1^i[loc]]$;
    (4) Output $msg$;

−Decryption(K, msg)
    (1) empty $r$;
    (2) For $j = 0 : |msg| - 1$ do
            $r \leftarrow msg \cup De_K(msg[j])$;
    (3) Output $r$;
```

Figure 2: The Token, Search and Decryption operations of the REX scheme.

Token Comm. Comp.	$O(1)$
Reply Comm. Compl.	$O(\|DB(\omega_i, \omega_j)\|)$
Search Comp. Compl.	$O(\|DB(\omega_i, \omega_j)\|)$
Client Storage Compl.	$O(\|\mathbf{W}\|(\mu + \psi))$
Server Storage Compl.	$O(\|\mathbf{W}\|^2(\mu + \log(\|\mathbf{F}\|)))+O(N\log(\|\mathbf{F}\|))$

Table 1: The asymptotic communication, search computation and storage complexities of REX.

become negligible by choosing sufficiently large dictionaries (with respect to $\mu^{-1}N^2$ for D_0 and $\mu^{-1}|W|^4$ for D_1). Also, since the scheme is static, a collision can be easily resolved at the setup phase by choosing new row or column randomness, $r_i^{(0)}$ and $r_i^{(1)}$, respectively.

3.5 Security

Intuitively, the client must compute a token that provides sufficient information to the server to locate the file identifiers and nothing else. We assume that the server is curious but honest, i.e. it executes the operations properly, but at the same time it tries to extract further knowledge.

The setup leakage is only due to the size of the two dictionaries. From that the adversary can deduce the number of pairs N and the size of the keywords set $\mathbf{W}$. Of course, there are techniques for hiding them, if it is necessary, at the expense of some redundancy (we can use padding or even use one hash table for both dictionaries). Thus, the setup leakage is at most:

$$\mathcal{L}_S = (N, |\mathbf{W}|).$$

Note that, even in the worst case described above, the setup leakage of REX is much smaller than the information leaked from the OPE based schemes. Due to the order preservation property and the deterministic nature of these schemes, in most of the proposed attacks the adversary needs just a snapshot of the encrypted data and it is not necessary to eavesdrop query transactions. REX uses semantically secure encryption to limit that leakage.

Next, we compute the leakage due to the token and search operations. Let $q_{i,j}$ be the query for the pair of keywords (ω_i, ω_j). Since the token generation is deterministic, it leaks the history of queries $\{Hist(q_{i,j})\}$. That is, it is leaked the moment when the same query was submitted in the past. This leakage, as well as the access pattern that we will describe next, are common in practically all the proposed SSE schemes and they are considered acceptable.

More over, due to the nature of the query, i.e. it covers a range of keywords, it is also common to leak the history of any sub-range that has been queried in the past as part of another query or as a separate query. Thus, the leakage includes also $Hist(q_{i',j'})\}_{i \le i' \le j' \le j}$.

The access pattern $accp(q_{i,j})$ is also leaked, as expected, as a sequence of ciphertexts $\{c_1, \cdots, c_{|DB(\omega_i,\omega_j)|}\}$, i.e. the response of the server. At the same time, this access pattern is divided into $len(q_{i,j})$ lists of ciphertexts $accp(q_{i,j})^{(r)}$, for $1 \le r \le len(q_{i,j})$. The value $len(q_{i,j}) = j - i + 1$ is the number of keywords in the range and it is leaked as part of the token. The lists have a specific order corresponding to the ordered sequence of keywords in the query range.

The information leaked per keyword in the range is not the entire set of ciphertexts for the specific keyword. Each list $accp(q_{i,j})^{(r)}$ contains only the ciphertexts of the files that have not appeared in the previous lists $accp(q_{i,j})^{(r')}$, $i \le r' < r$. This leakage, is much smaller than the usual access pattern leakage that we encounter in other range SSE schemes. That is, while the adversary has also learned all the sets $DB(\omega_i, \omega_{j'})$, for $i \le j' \le j$, only a subset of the sets $DB(\omega_{i'}, \omega_{j'})$, for $i' \le j' \le j$ and $i < i' \le j$ is leaked.

Finally, the scheme is response-hiding and the ciphertexts of the same file identifier are indistinguishable up to the security advantage of the *SKE* scheme.

The search leakage of the scheme is given by:

$$\begin{aligned}\mathcal{L}_Q(\omega_i, \omega_j) &= (accp(q_{i,j}), \{accp(q_{i,j})^{(r)}\}_{i \le r \le j}, \\ &\quad \{Hist(q_{i',j'})\}_{i \le i' \le j' \le j}).\end{aligned}$$

Next, we give a sketch of the proof for non-adaptive security, which is rather straightforward. The proof of the adaptive case presents some technical challenges and we are going to present it with the dynamic version of REX that we are currently working on.

We need to show the existence of a simulator $\mathcal{S}$ who has access to the leakage profiles $\mathcal{L}_S$ and $\mathcal{L}_Q$, and produces indistinguishable views to an adversary $\mathcal{A}$ (according to Definition 2.3). The only difference from the definition is that the adversary sends the queries all together. Again, the attacker $\mathcal{A}$, i.e. the server, is semi-honest. The simulator $\mathcal{S}$ must create m tokens for each query. Since the

$\|\mathbf{W}\|$	Client Storage
$1k$	$31kB$
$10k$	$320kB$
$100k$	$3MB$

Table 2: Local (client) storage size.

queries are known beforehand, $\mathcal{S}$ creates an ordered sequence of the keywords and stores in a multi-map the randomness per keyword. The number of keywords $|\mathbf{W}|$ is leaked and the value j can be deduced from the order. Then, the simulator just applies the token operation. The value of $counter$ is the range length and it is known from the leakage function. The full proof will appear at the extended version of the paper.

4 IMPLEMENTATION AND EXPERIMENTS

We implemented REX in Java. From the Clusion framework, we have used the functions for file parsing and extraction of keywords to multi-maps. Clusion is an open source encrypted search framework provided by Kamara and Moataz ([16]).

For the symmetric encryption scheme SKE we used AES in CBC mode with key size 128 bits. At the setup phase the matrix R was implemented as an array of arraylists and $tempR$ as an array of generic objects. The local storage S is a multi-map and the outsourced D_0 and D_1 are HashMaps (with support of multiple collisions). We have chosen to use $SHA-1$ as the underlying hash function.

We ran our experiments on a desktop computer with an Intel Xeon E3-1241 3.GHz CPU (with 8 logical cores), 8GB of RAM, a 250 GB SSD, running on Windows 10. Our code is Open Source.

4.1 Evaluation

We have computed the client storage size for different keyword sets (randomly produced) and we present some indicative results in Table 2. We optimized the load of S. It seems that the requirements are reasonable (similar results have been provided in [3]).

Regarding the search time, we have computed the average per matching entry time. We have performed the computations locally, thus there is no communication latency (see Table 3). We used for our experiments around $10k$ randomly produced files with an average of 20 keywords per file. The results indicate that as the number of output keywords per query increases, REX performs better, i.e. the required time per file is decreasing. Definitely, our implementation can be significantly improved. However, even these early results are very promising.

We are planning to accelerate the search operation by visiting the lists $L_i^{(0)}$ in parallel and use a more efficient hash function like Blake2b. We are currently investigating the dynamic version of REX (see also Section 5) and we are planning to enrich our measurements.

5 FUTURE WORK: THE DYNAMIC REX

In this section, we describe our work in progress, the dynamic version of REX. However, static SSE schemes have an interest of their own and they are used in many applications (for instance archiving).

Number of matching files	Average Time per matching file (ms)	Range $O(j-i)$
750	0.025	800
550	0.028	800
350	0.03	150
150	0.03	100

Table 3: Average computation time per matching file. No communication overhead is included.

Let's assume we want to add a new file $\hat{f}$. Thus, we have a new file identifier $\hat{id}$ and a list of z keywords $\{\omega_{j_1}, \omega_{j_2}, \cdots, \omega_{j_z}\}$. In classical single keyword SSE schemes, we have to update the sets $DB(\omega_{j_i})$, $1 \leq i \leq z$. In our case we will update the structure R. We will assume, without loss of generality, that $\omega_{j_1} < \omega_{j_2} < \cdots < \omega_{j_z}$.

The updates are based on the following observation. The tuple $R[i, 0]$ contains the identifiers of the files that have the keyword ω_i and that do not have any of the keywords $\{\omega_j, \cdots, \omega_{i-1}\}$. Thus, since the first keyword of $\hat{f}$ is ω_{j_1}, the tuple $R[j_1, 0]$ must be updated with $\hat{id}$.

By design, a file identifier can appear only once in any $R^{(i,j)}$. Thus, $\hat{id}$ cannot belong in any other tuple of the columns $R[:, j]$, for $0 \leq j \leq j_1$. Following the same reasoning, $R[j_2, j_1+1]$ is updated with $\hat{id}$ and so on. In other words, we insert the new file identifier $\hat{id}$ in the tuples $R[j_1, 0]$ and $R[j_{r+1}, j_r + 1]$, for $1 \leq r < z$. The nice thing is that we can use this update approach to improve our setup phase, as well. The implementation performance so far is very promising.

Finally, we want to support modifications of the keyword set $\mathbf{W}$. Mainly, we want to add new keywords to REX. This has led us to modify the local storage S. Instead of using a multi-map, we organize S as a search tree. This adds some logarithmic overhead to the client and it is somehow close to the work of [1]. Unfortunately, it seems difficult to avoid multiple rounds without communication overhead (using and updating garbled circuits to maintain single round).

6 CONCLUSIONS

In this paper, we have presented REX, a new efficient SSE scheme that supports range queries. REX is single round and has optimal communication and search computation complexity. It is using semantically secure encryption algorithms and it offers more security than traditional Order Preserving Encryption based range SSE schemes. We have implemented REX and the first experiments are very promising.

Currently, we are working on an extended version of REX, the dynamic REX. The enhanced REX will support updates as well, i.e. the client will be able to add new files or delete existing ones. At the same time, we improve our implementation as several tweaks can significantly improve its performance.

A APPENDIX

A.1 Proof of Theorem 3.2

(1) This is a straightforward result since the first $i-1$ entries of the i-th column are left empty by design.

(2) Let's assume that there are two tuples of $R^{(i,j)}$ that have at least one element in common and let $R[i_1, j_1]$ be the first one and $R[i_2, j_2]$ the second one, where $0 \leq j_1 \leq j_2 \leq i$ and $i \leq i_1, i_2 \leq j$. We distinguish two cases: $i_1 \leq i_2$ and $i_1 > i_2$.
In the first case, $R[i_1, j_1]$ belongs to $R^{(i_2,j_2)}$, and by design $R[i_2, j_2]$ has no common elements with the other tuples of $R^{(i_2,j_2)}$. Thus, we have a contradiction.
In the second case, $i_1 > i_2$. Let u be the element that appears at both tuples. Since, $u \in R[i_1, j_1]$, then by design $u \in DB(\omega_{i_1})$ and $u \notin DB(\omega_{i_2})$. Again, a contradiction.

(3) By design the tuple $R[j, i]$ contains all the elements of the set $DB(\omega_j)$ that are not in $R^{(i,j)}$, i.e. $R^{(i,j)}$ contains the entire $DB(\omega_j)$. Since, $R^{(i,j')}$ is a submatrix of $R^{(i,j)}$, that means that $R^{(i,j)}$ contains also $DB(\omega_{j'})$, for $i \leq j' \leq j$.

ACKNOWLEDGMENTS

REFERENCES

[1] Tobias Boelter, Rishabh Poddar, and Raluca Ada Popa. 2016. A Secure One-Roundtrip Index for Range Queries. *IACR Cryptology ePrint Archive* 2016 (2016). http://eprint.iacr.org/2016/568

[2] A. Boldyreva, N. Chenette, and A. O Neill. 2009. Order-Preserving Symmetric Encryption. In *Advances in Cryptology - EUROCRYPT 2009 - 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques.*

[3] Raphael Bost. 2016. $\sum o\varphi o\varsigma$: Forward Secure Searchable Encryption. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016.* 1143–1154.

[4] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-Abuse Attacks Against Searchable Encryption. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015.* 668–679.

[5] David Cash, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. 2013. Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries. In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I.* 353–373.

[6] Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. 2006. Searchable symmetric encryption: improved definitions and efficient constructions. In *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, Ioctober 30 - November 3, 2006.* 79–88.

[7] Ioannis Demertzis, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, and Minos Garofalakis. 2016. Practical Private Range Search Revisited. In *2016 ACM SIGMOD, San Francisco, CA, USA, Jun 26 - July 01, 2016.*

[8] F. Betül Durak, Thomas M. DuBuisson, and David Cash. What Else is Revealed by Order-Revealing Encryption?. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016.* 1155–1166.

[9] Sky Faber, Stanislaw Jarecki, Hugo Krawczyk, Quan Nguyen, Marcel-Catalin Rosu, and Michael Steiner. 2015. Rich Queries on Encrypted Data: Beyond Exact Matches. In *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part II.* 123–145.

[10] Craig Gentry. 2009. Fully Homomorphic Encryption Using Ideal Lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing (STOC '09).* 169–178.

[11] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* 43, 3 (1996), 431–473.

[12] Paul Grubbs, Kevin Sekniqi, Vincent Bindschaedler, Muhammad Naveed, and Thomas Ristenpart. Leakage-Abuse Attacks against Order-Revealing Encryption. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017.* 655–672.

[13] Florian Hahn and Florian Kerschbaum. 2014. Searchable Encryption with Secure and Efficient Updates. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014.* 310–320.

[14] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012.*

[15] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2014. Inference attack against encrypted range queries on outsourced databases. In *Fourth ACM Conference on Data and Application Security and Privacy, CODASPY'14, San Antonio, TX, USA - March 03 - 05, 2014.* 235–246.

[16] Seny Kamara and Tarik Moataz. 2016. Clusion. *https://github.com/orochi89/Clusion* (2016).

[17] Seny Kamara and Tarik Moataz. 2017. Boolean Searchable Symmetric Encryption with Worst-Case Sub-linear Complexity. In *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part III.* 94–124.

[18] Seny Kamara and Charalampos Papamanthou. 2013. Parallel and Dynamic Searchable Symmetric Encryption. (2013), 258–274.

[19] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. 2012. Dynamic searchable symmetric encryption. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012.* 965–976.

[20] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. Generic Attacks on Secure Outsourced Databases. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016.* 1329–1340.

[21] Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. 2017. Improved Reconstruction Attacks on Encrypted Data Using Range Query Leakage. *IACR Cryptology ePrint Archive* 2017 (2017), 701. http://eprint.iacr.org/2017/701

[22] Muhammad Naveed, Seny Kamara, and Charles V. Wright. Inference Attacks on Property-Preserving Encrypted Databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015.* 644–655.

[23] Raluca A. Popa, Frank H. Li, and Nickolai Zeldovich. 2013. An Ideal-Security Protocol for Order-Preserving Encoding. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013.* 463–477.

[24] R. A. Popa, N. Zeldovich, C.M.S. Redfield, and H. Balakrishnan. 2011. CryptDB: protecting confidentiality with encrypted query processing. In *2011 ACM Symposium on Operating Systems Principles, SOSP 2011.*

[25] Panagiotis Rizomiliotis and Stefanos Gritzalis. 2015. ORAM Based Forward Privacy Preserving Dynamic Searchable Symmetric Encryption Schemes. In *Proceedings of the 2015 ACM Workshop on Cloud Computing Security Workshop, CCSW 2015, Denver, Colorado, USA, October 16, 2015.* 65–76.

[26] Dawn Xiaodong Song, David A. Wagner, and Adrian Perrig. 2000. Practical Techniques for Searches on Encrypted Data. In *2000 IEEE Symposium on Security and Privacy, Berkeley, California, USA, May 14-17, 2000.* 44–55.

[27] Andrew C. Yao. 1982. Protocols for Secure Computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science (FOCS '82).* 160–164.

[28] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.* 707–720.