Distributed Searchable Symmetric Encryption

Christoph Bösch*, Andreas Peter*, Bram Leenders*, Hoon Wei Lim†, Qiang Tang‡,
Huaxiong Wang†, Pieter Hartel*, and Willem Jonker*
* CTIT, University of Twente, The Netherlands
{c.boesch, a.peter, pieter.hartel, willem.jonker}@utwente.nl
b.c.leenders@student.utwente.nl
† CCRG, Nanyang Technological University, Singapore
{hoonwei, hxwang}@ntu.edu.sg
‡ APSIA, SnT, University of Luxembourg
qiang.tang@uni.lu

***Abstract*—Searchable Symmetric Encryption (SSE) allows a client to store encrypted data on a storage provider in such a way, that the client is able to search and retrieve the data selectively without the storage provider learning the contents of the data or the words being searched for. *Practical* SSE schemes usually leak (sensitive) information during or after a query (e.g., the search pattern). *Secure* schemes on the other hand are not practical, namely they are neither efficient in the computational search complexity, nor scalable with large data sets. To achieve efficiency and security at the same time, we introduce the concept of *distributed* SSE (DSSE), which uses a query proxy in addition to the storage provider.**

We give a construction that combines an inverted index approach (for efficiency) with scrambling functions used in private information retrieval (PIR) (for security). The proposed scheme, which is entirely based on XOR operations and pseudo-random functions, is efficient and does not leak the search pattern. For instance, a secure search in an index over one million documents and 500 keywords is executed in less than 1 second.

Keywords*—*Searchable Encryption, Search Pattern Hiding, Practical Efficiency, Semi-Honest Model

I. Introduction

Searchable Symmetric Encryption (SSE) allows a client to outsource data in encrypted form to a semi-honest server/storage provider (like a cloud provider), such that the encrypted data remains searchable without decrypting and without the server learning the contents of the data or the words being searched for. In most cases this is achieved by introducing a searchable encrypted index, which is stored together with the encrypted data (e.g., documents) on a server. To enable the server to query the data, the client creates a trapdoor which allows the server to do the search on behalf of the client. Practical SSE schemes try to make this search process as efficient as possible, which usually comes at the cost of leaking (sensitive) information, such as the search pattern, i.e., the information if two trapdoors were generated for the same keyword [11].

Over the last decade there has been active research in SSE [6]–[8], [11], [12], [15], [16], [18], [25]–[27]. The majority of the schemes has a search complexity which is linear in the number of documents stored on the server, since one index per document has to be searched. Some schemes allow a more efficient search, e.g., by using an *inverted* index [11], [15], [16], [27], which is an index per distinct keyword in the database. This reduces the search complexity to (at least) the number of distinct keywords in the document collection. However, the reduced complexity usually comes at the cost of reduced security.

A limitation of most previous SSE schemes is the leakage of the search pattern [16]. Revealing the search pattern in SSE schemes is a serious problem, as it allows an attacker to perform statistical analysis on the occurrence frequency of each query. This allows an attacker to gain knowledge on the underlying plaintext keywords, rendering the encryption scheme less useful (as is convincingly demonstrated by Liu et al. [20]). The problem of leaking the search pattern is not only recognized in the setting of SSE, but also in the setting of predicate encryption [25].

Kantarcioğlu and Clifton [17] were the first to prove that in a *single* server setting, a cryptographically secure SSE scheme needs to process the whole database per query to protect sensitive information (including the search pattern), thus being inefficient in practice. The authors propose the use of a fully trusted party (a trusted hardware module [1], [2] in their case) to make a cryptographically secure SSE scheme efficient and sketch a construction for relational databases.

To obtain more efficient SSE schemes or to realize more complex queries than just keyword queries, a common approach is to split the server into a semi-honest storage provider and a semi-honest (query) proxy [5], [6], [10], [23], [24], [28], [29]. Unfortunately, also all of these schemes leak the search pattern.

In this paper we propose an efficient construction of an SSE scheme in the above setting that also hides the search pattern. The main idea behind our construction is to distribute the search on the encrypted data to the storage provider and the query proxy. Therefore, we call our new scheme a *distributed* Searchable Symmetric Encryption (DSSE) scheme. Our new DSSE scheme achieves its efficiency due to distributed computation and the use of efficient primitives like XOR and pseudo-random functions only. We use an inverted index, which is a common approach to reduce the search complexity

Part of this work was done while the first author was at Nanyang Technological University, Singapore. Andreas Peter is supported by the THeCS project as part of the Dutch national program COMMIT.

978-1-4799-3503-1/14/$31.00 ©2014 IEEE

in databases. The ordinary use of an inverted index directly leaks the search pattern. To hide the search pattern, we make use of techniques used in oblivious RAM [14], [21], [22] (ORAM) and private information retrieval [3], [9] (PIR), which solve this problem by continuously re-shuffling the index as it is being accessed. In this way, neither the storage provider nor the query proxy can tell which record was accessed and thus the search pattern of the scheme remains hidden.

We make the following contributions:

1) We formally define the concept of DSSE and its security (Section II).
2) We propose a simple and efficient search pattern hiding DSSE construction (Section III).
3) We prove the security of our DSSE scheme in the semi-honest model and show that it only leaks the access pattern and nothing more (Section IV).
4) We implement the core components of our scheme and analyse its performance (Section V).
5) We discuss the security implications of colluding servers, and propose a highly efficient SSE construction resulting from such a collusion (Section VI).
6) We prove adaptive semantic security for the SSE scheme, as defined by Curtmola et al. [11] (Section VI-B).
7) We give an analysis of theoretical performance of the SSE scheme.

II. Distributed Searchable Symmetric Encryption

In this section, we formally define the new notion of *distributed* searchable symmetric encryption (DSSE) and its security. As such a scheme involves a client C, a storage provider (SP) and a query proxy (QP), we formulate it as a protocol between these three parties.

Notation. Throughout the paper, we use the following notation. Let $\mathbf{D} = \{d_1, \ldots, d_n\}$ be a set of n files (e.g., documents). Let $\mathcal{W} = \{w_1, \ldots, w_m\}$ be a pre-built dictionary of m keywords. Given a document d, let $u(d)$ denote the set of distinct keywords in d. Given a tuple t, we refer to the i-th entry of t as $t[i]$. The encrypted index is denoted by $\mathcal{I} = \{I_{w_1}, \ldots, I_{w_m}\}$. $\mathcal{J}$ denotes a re-encrypted index and $\mathcal{I}'$ a re-encrypted and permuted index. The keyword used in a query we denote by $s \in \mathcal{W}$. The set of document identifiers of all documents in $\mathbf{D}$ containing the query keyword s is written as $id_s(\mathbf{D})$. An element a randomly chosen from a set A is denoted by $a \xleftarrow{\$} A$. For two distribution ensembles X and Y, we denote computational indistinguishability by $X \equiv_c Y$.

Definition 1 (Distributed Searchable Symmetric Encryption Scheme)**.** *A* Distributed Searchable Symmetric Encryption (DSSE) *scheme for a set of keywords $\mathcal{W} = \{w_1, \ldots, w_m\}$ is a protocol between three parties: a client C, a storage provider SP and a query proxy QP, and consists of the following four probabilistic polynomial time (PPT) algorithms:*

- $(K_C, K_1, K_2) \longleftarrow \mathsf{Keygen}(\lambda)$*: This algorithm is run by the client C, takes a security parameter λ as input, and outputs a secret key K_C to the client C and secret keys K_1 and K_2 to SP and QP, respectively.*
- $\mathcal{I} = (\mathcal{I}_1, \mathcal{I}_2) \longleftarrow \mathsf{BuildIndex}(K_C, \mathbf{D})$*: This algorithm is run by the client C, takes a key K_C and a set of documents $\mathbf{D}$ as input, and outputs an encrypted index $\mathcal{I}_1$ to SP and $\mathcal{I}_2$ to QP.*
- $T^s = (T_1^s, T_2^s) \longleftarrow \mathsf{Trapdoor}(K_C, s)$*: This algorithm is run by the client C, takes a key K_C and a query keyword $s \in \mathcal{W}$ as input, and outputs a trapdoor T_1^s to SP and trapdoor T_2^s to QP.*
- $X \longleftarrow \mathsf{SearchIndex}(T^s, \mathcal{I} = (\mathcal{I}_1, \mathcal{I}_2), K_1, K_2)$*: This algorithm is a protocol between SP and QP. SP provides $T_1^s, \mathcal{I}_1, K_1$ and QP provides $T_2^s, \mathcal{I}_2, K_2$ as input. The algorithm has a set of document identifiers X of documents in $\mathbf{D}$ as (public) output.*

Additionally, we require a DSSE scheme to be correct*, i.e., that for the set of keywords $\mathcal{W}$, any set of documents $\mathbf{D}$, all security parameter λ, all outputs $(K_C, K_1, K_2) \longleftarrow \mathsf{Keygen}(\lambda)$, $\mathcal{I} \longleftarrow \mathsf{BuildIndex}(K_C, \mathbf{D})$, $T^s \longleftarrow \mathsf{Trapdoor}(K_C, s)$ and all keywords $s, w \in \mathcal{W}$, it holds that:*

$$\mathsf{SearchIndex}(T^s, \mathcal{I}, K_1, K_2) = id_s(\mathbf{D}),$$

where $id_s(\mathbf{D})$ denotes the set of identifiers of all documents in $\mathbf{D}$ containing the keyword s. The sequence of document identifiers $id_s(\mathbf{D})$ for consecutive keywords s is called the access pattern*.*

Suppose a client makes Q queries, while the i-th query queries for keyword $s_i \in \mathcal{W}$; so in total the client queries for $s_1, \ldots, s_Q \in \mathcal{W}$. To distinguish between the different trapdoors associated with these Q queries, we write T^{s_i} to denote a trapdoor for the i-th query (i.e., the client queries for the keyword s_i). We denote an admissible protocol run of a DSSE scheme, where the client performs Q queries, by Π^Q_{DSSE}. Formally, an admissible Q-query protocol run Π^Q_{DSSE} is defined as follows:

Definition 2 (Admissible Q-query protocol run Π^Q_{DSSE})**.** *Consider a DSSE scheme with keyword set $\mathcal{W}$, output (K_C, K_1, K_2) of $\mathsf{Keygen}(\lambda)$, and a document set $\mathbf{D}$. For a given $Q \in \mathbb{N}$, an* admissible Q-query protocol run *consists of one call of algorithm $\mathcal{I} \longleftarrow \mathsf{BuildIndex}(K_C, \mathbf{D})$, followed by Q calls of algorithm $T^{s_i} \longleftarrow \mathsf{Trapdoor}(K_C, s_i)$ for (possibly different) keywords $s_i \in \mathcal{W}$ for $i \in [1, Q]$, and another Q calls of algorithm $\mathsf{SearchIndex}(T^{s_i}, \mathcal{I}, K_1, K_2)$. We denote such a protocol run by Π^Q_{DSSE}.*

A. Security Model

Following all previous works on SSE, we treat the client as a trusted party. Concerning SP and QP, we approach the security of a DSSE scheme by following the real-vs-ideal paradigm of secure multiparty computation [13, Ch. 7] in the semi-honest model. This means that we assume SP and QP to act honest-but-curious, i.e., they will follow all protocol steps honestly but may try to infer all kinds of information on other parties inputs or intermediate results beyond what the output of the DSSE scheme reveals. Moreover, we assume secure channels between any of the parties and that SP and QP do not collude.

In particular, this implies that only admissible Q-query protocol runs Π^Q_{DSSE} are performed (for $Q \in \mathbb{N}$). Now intuitively, since the protocol Π^Q_{DSSE} only has the access pattern $(id_{s_1}(\mathbf{D}), \ldots, id_{s_Q}(\mathbf{D}))$ as public output to all participants, if a DSSE scheme is secure in the semi-honest real-vs-ideal paradigm, it leaks no information (including the search pattern) other than the access pattern. Following this paradigm [13, Ch. 7], we first define the ideal functionality of a DSSE scheme as follows:

Definition 3 (Functionality $\mathcal{F}^Q_{\mathsf{DSSE}}$)**.** *Consider a DSSE scheme with keyword set $\mathcal{W}$, output (K_C, K_1, K_2) of $\mathsf{Keygen}(\lambda)$, and a document set $\mathbf{D}$. For $Q \in \mathbb{N}$, $\mathcal{F}^Q_{\mathsf{DSSE}}$ is the functionality that takes as input*

- *K_C and keywords $s_1, \ldots, s_Q$ from the client C,*
- *K_1 from the storage provider SP, and*
- *K_2 from the query proxy QP.*

and outputs $id_Q(\mathbf{D}) := (id_{s_1}(\mathbf{D}), \ldots, id_{s_Q}(\mathbf{D}))$ to all *the parties C, SP and QP.*

Then, we say that a DSSE scheme is *secure* if any admissible Q-query protocol run Π^Q_{DSSE} (for any $Q \in \mathbb{N}$) privately computes the functionality $\mathcal{F}^Q_{\mathsf{DSSE}}$. Formally, this means:

Definition 4 (Security)**.** *We say that a DSSE scheme is* secure*, if for any $Q \in \mathbb{N}$, the protocol Π^Q_{DSSE} privately computes the functionality $\mathcal{F}^Q_{\mathsf{DSSE}}$ between the three parties C, SP and QP, i.e., there exists a (PPT) simulator $\mathcal{S}$ such that*

$$\{\mathcal{S}(K_1, id_Q(\mathbf{D}))\}_{K_C, s_1, \ldots, s_Q, K_1, K_2}$$
$$\equiv_c \{\mathsf{View}_{SP}(K_C, s_1, \ldots, s_Q, K_1, K_2)\}_{K_C, s_1, \ldots, s_Q, K_1, K_2}$$

and

$$\{\mathcal{S}(K_2, id_Q(\mathbf{D}))\}_{K_C, s_1, \ldots, s_Q, K_1, K_2}$$
$$\equiv_c \{\mathsf{View}_{QP}(K_C, s_1, \ldots, s_Q, K_1, K_2)\}_{K_C, s_1, \ldots, s_Q, K_1, K_2}$$

Note that it is sufficient to simulate the views of SP and QP separately as we do not consider any form of collusion between them. Recall that the client is treated as a trusted party who only provides inputs and so the security definition does not need to take the client's view into account.

III. The Proposed Distributed Construction

Recall that a DSSE scheme consists of three parties: a client C, a storage provider SP and a query proxy QP. Our proposed scheme uses an inverted index, that is, an index per distinct keyword in the database. Each index consists of a single bit per keyword per document. A plaintext index ι_w for keyword w is a bit string of length n, where n is the number of documents in the database. Each position $\iota_w[j]$ corresponds to a unique document, where j is a unique document identifier. If a document d_j contains the keyword w, then the j-th bit of ι_w is set to 1. Otherwise the bit is set to 0. To protect the plaintext index ι_w, it is encrypted, by a bitwise XOR operation (denoted as $\oplus$) with several keyed pseudo-random functions described below. Concerning the output of $\mathsf{Keygen}(\lambda)$, the key $K_C = (K_f, K_p)$ is only known by the client C, the key K_1 is a shared key and known by C and SP. Formally, K_1 is contained in K_C as a second component which we omit here for reasons of readability and just say that C knows both K_C and K_1. The second key K_2 for QP is empty in our proposed solution. We assume that the documents are encrypted by the client with some standard symmetric encryption algorithm using a key different from K_C. Since the document encryption is independent from our scheme it is not considered further. Our construction makes use of the following cryptographic primitives:

- $f(K_C, w)$: The function $f(K_C, w)$ takes a key K_C and a keyword w as input. It outputs a pseudo-random bit-string of length n.
- $g(K_1, w, r_1)$: The function takes as input a key K_1, a keyword w and a random value r_1. It outputs a pseudo-random bit-string of length n.
- $h(K_1, r_1)$: The function takes a key K_1, and a random value r_1 as input. The output is an n-bit pseudo-random string.
- p_k: The keyed pseudo-random permutation p_k describes a permutation on the set $[1, m]$. The evaluation of the permutation p_k takes as input an element $x \in [1, m]$ and outputs its permuted position $p_k(x) \in [1, m]$.
- $\pi(\mathcal{X}, p_x)$: The function takes as input a set $\mathcal{X}$ of size $|\mathcal{W}|$ and a random permutation p_x. It outputs a permuted set according to p_x.

For ease of readability we will omit the keys K_C, K_1 and use $f_w, g_w(r_1)$ and $h(r_1)$ in the rest of the paper to denote $f(K_C, w)$, $g(K_1, w, r_1)$ and $h(K_1, r_1)$, respectively.

A. Our Construction

Next, we describe the four algorithms of our proposed scheme, namely Keygen, $\mathsf{BuildIndex}$, $\mathsf{Trapdoor}$ and $\mathsf{SearchIndex}$. The key K_2, as well as the index $\mathcal{I}_2$ are empty in our construction and are thus omitted in the description.

- $(K_C, K_1, K_2) \longleftarrow \mathsf{Keygen}(\lambda)$: Given a security parameter λ, generate a key $K = (K_C = (K_f, K_p), K_1)$ for the pseudo-random functions. The key K_C is only known by C, the key K_1 is known by C and SP.
- $\mathcal{I} = (\mathcal{I}_1, \mathcal{I}_2) \longleftarrow \mathsf{BuildIndex}(K_C, \mathbf{D})$: With the key K_C, and a document collection $\mathbf{D}$, the algorithm does the following:
  1) For all search keywords $w_i \in \mathcal{W}$:
     a) $\forall d_j \in \mathbf{D}$: set $\iota_{w_i}[j] = 1$, if $w_i \in u(d_j)$; otherwise $\iota_{w_i}[j]$ is set to 0.
     b) Encrypt the index ι_{w_i} as follows: $I_{w_i} = \iota_{w_i} \oplus f_{w_i}$.
  2) Permute the index $\mathcal{I} = \pi(\{I_{w_i}\}, p_{K_p})$ based on the client's key K_p.
  3) Output the index $\mathcal{I}$ and send to SP.
- $T^w = (T^w_1, T^w_2) \longleftarrow \mathsf{Trapdoor}(K_C, w)$: With the key K, and a query keyword $s \in \mathcal{W}$, the algorithm

selects three random values r_1, r_2, r_3 and sets $T_1^s = (r_1, r_2, r_3)$. Then, the algorithm generates the client's dictionary as $\mathcal{W}^c = \pi(\mathcal{W}, p_{K_p})$. Next, the algorithm calculates the query dictionary $\mathcal{W}^q = \pi(\mathcal{W}^c, p_{r_2})$ and looks up the current position $q_s(r_2)$ for the desired keyword s in the permuted keyword list $\mathcal{W}^q$. Generate the trapdoor $T_2^s = (q_s(r_2), k = f_s \oplus g_s(r_1) \oplus r_3)$. Output $T^s =$

$$(T_1^s = (r_1, r_2, r_3), T_2^s = (q_s(r_2), f_s \oplus g_s(r_1) \oplus r_3))$$

- $X \longleftarrow \mathsf{SearchIndex}(T^w, \mathcal{I} = (\mathcal{I}_1, \mathcal{I}_2), K_1, K_2)$: ($SP$ provides T_1^s and QP provides T_2^s) The storage provider SP re-encrypts and permutes the index $\mathcal{I}$ for all $i \in [1, m]$ as follows:

$$\mathcal{J} = \{J_{w_i}\} = \{I_{w_i} \oplus g_{w_i}(r_1) \oplus h(r_1)\},$$

$$\mathcal{I}' = \pi(\mathcal{J}, p_{r_2})$$

and sends $\mathcal{I}'$ to QP. QP stores $\mathcal{I}'$ as its current index and performs a table lookup for $q_s(r_2)$ on $\mathcal{I}'$ to obtain the right I'_s. QP then re-encrypts as follows:

$$\begin{aligned} I''_s &= I'_s \oplus k \\ &= (\iota_s \oplus f_s \oplus g_s(r_1) \oplus h(r_1)) \oplus (f_s \oplus g_s(r_1) \oplus r_3) \\ &= \iota_s \oplus h(r_1) \oplus r_3. \end{aligned}$$

I''_s is sent to SP, which can now decrypt $\iota_s = I''_s \oplus h(r_1) \oplus r_3$. The result ι_s encodes, whether a document satisfies the query keyword s or not. Depending on the client, SP sends either the matching document ids or directly the matching encrypted documents to C.

A standard work flow is as follows. A client C first runs the Keygen algorithm to generate the key K. To create a searchable index, C runs the $\mathsf{BuildIndex}$ algorithm which outputs the inverted index $\mathcal{I}$. Finally C stores the index $\mathcal{I}$ together with the encrypted documents on the storage provider SP.

Later on, when the client wants to retrieve some documents containing a search keyword $s \in \mathcal{W}$, it first runs the $\mathsf{Trapdoor}$ algorithm to generate the trapdoor $T^s = (T_1^s, T_2^s)$. C sends T_1^s to SP and T_2^s to QP. Then, SP and QP can run the $\mathsf{SearchIndex}$ algorithm. SP re-encrypts and permutes the index $\mathcal{I}$ with help of T_1^s and sends the new $\mathcal{I}'$ to QP. QP performs a table look-up and then re-encrypts the result using the key k inside T_2^s. The temporary result I''_s is sent to SP, which can now decrypt using T_1^s to obtain the plaintext index ι_s for the search keyword s. Finally, SP either sends the matching ids or the matching encrypted documents to the client.

By letting SP perform the re-encryption and permutation, QP receives a fresh index before each query. These indexes are indistinguishable from each other and also from random. Thus the next query will not leak any information. To make the scheme more efficient, the client can choose another re-encryption policy, e.g., to trigger the re-encryption before he queries the same keyword twice. In this way, SP and QP can reduce the computational and communication complexity.

B. Updates

The proposed DSSE scheme allows efficient updates of the document collection, like most of the SSE schemes. A user can update the index by adding and deleting documents without revealing information. Only the number of documents processed is leaked. To add a document $j+1$, the $\mathsf{BuildIndex}$ algorithm is run and the new indexes $\iota_{w_i}[j+1]$ are encrypted and appended to the existing indexes. To delete a document d_x from the collection, the client sets the indexes $\iota_{w_i}[x]$ to 0, encrypts and sent them to SP.

IV. Security Analysis

Theorem 1 (Security)**.** *Our proposed DSSE scheme from Section III is secure with respect to Definition 4.*

Proof: Let $Q \in \mathbb{N}$. By the Composition Theorem in the semi-honest model [13, Theorem 7.5.7], we can treat each step in protocol Π^Q_{DSSE} separately. We start by constructing a simulator $\mathcal{S}$ of SP's view in each step of protocol Π^Q_{DSSE}. We then construct a simulator $\mathcal{S}$ of QP's view of protocol Π^Q_{DSSE}.

Storage Provider SP**.** In line 2 of Figure 1, SP learns the values I_w for all keywords $w \in \mathcal{W} = \{w_1, \ldots, w_m\}$. Since this value is computed as an XOR of the plaintext index ι_w and the n-bit output of the pseudo-random function f with key K_C and keyword w, the value I_w is computationally indistinguishable from a random n-bit string (recall that $\mathbf{D}$ contains $|\mathbf{D}| = n$ documents). Therefore, $\mathcal{S}$ can simulate these values with random n-bit strings.

Now, let $s_1, \ldots, s_Q$ denote the keywords that the client queries for. In line 4, for each of these keywords s_j ($j = 1, \ldots, Q$), the storage provider SP learns the three random bit-strings r_1, r_2, and r_3. These can be trivially simulated by $\mathcal{S}$ by choosing random strings.

Finally, in line 7, SP receives the value I''_{s_j} which equals $\iota_{s_j} \oplus h(K_1, r_1) \oplus r_3$. But the simulator $\mathcal{S}$ knows the key K_1 and the overall output $id_Q(\mathbf{D}) = (id_{s_1}(\mathbf{D}), \ldots, id_{s_Q}(\mathbf{D}))$ of functionality $\mathcal{F}^Q_{\mathsf{DSSE}}$ by definition, and since he created the random values r_1 and r_3 himself, he can simulate I''_s by simply computing $\iota_s \oplus h(K_1, r_1) \oplus r_3$. This can be done for each keyword s_j and so $\mathcal{S}$ successfully simulated the view of the storage provider SP.

Query Proxy QP**.** In line 5 of Figure 1, for each keyword w_i ($i = 1, \ldots, m$), QP learns the value/index $\mathcal{I}'$. But this index is computed as a pseudo-random permutation of the re-encrypted index $\mathcal{J} = \{I_{w_i} \oplus g_{w_i}(r_1) \oplus h(r_1)\}$, while every entry in $\mathcal{J}$ is indistinguishable from a random n-bit string. Therefore, for each keyword w_j, the index $\mathcal{I}'$ is indistinguishable from a random $(m \times n)$-bit matrix, which can be simulated by $\mathcal{S}$ as such.

Let $s_1, \ldots, s_Q$ denote the Q keywords that the client queries for. In line 6, for each of the keywords s_j for $j \in [1, Q]$, the query proxy QP learns the values $q_{s_j}(r_2)$ and k. Since $q_{s_j}(r_2)$ is an index position for keyword s_j after a pseudo-random permutation with function π with input $\mathcal{J}$ and the

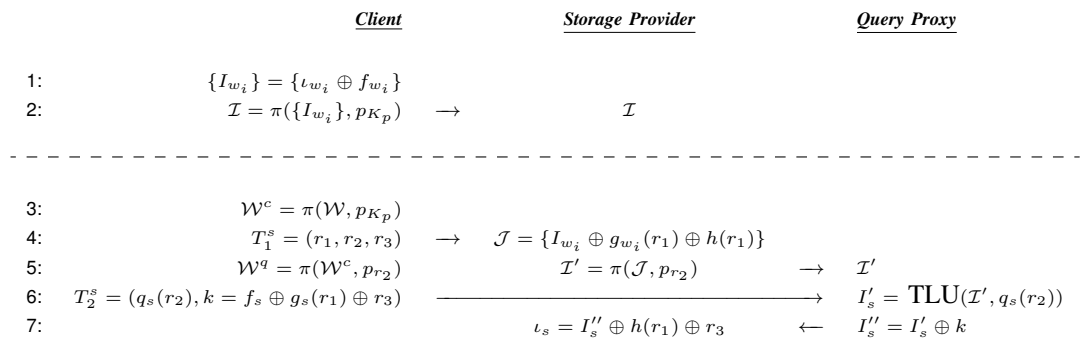

Fig. 1. Simplified upload and search processes of our DSSE scheme. TLU denotes a table look-up. The document up- and download is omitted.

pseudo-random permutation based on the random value r_2, the value can be simulated, by choosing a random value between 1 and m. The value k is computed as an XOR of the n-bit outputs of the pseudo-random functions $f(K_C, s_j)$ and $g(K_1, s_j, r_1)$ and the random n-bit string r_3. The value k is thus indistinguishable from random and can be simulated by $\mathcal{S}$ with random n-bit string. In total, this shows that $\mathcal{S}$ successfully simulates the view of the query proxy QP. ■

V. Performance Analysis

In this section, we consider the efficiency of our proposed DSSE scheme, where the efficiency is measured in terms of the computation and communication complexities.

Computation. The BuildIndex algorithm generates for all keywords $w \in \mathcal{W}$ an n-bit string (f_w). The resulting index is an $m \times n$-matrix, where m is the number of keywords and n the number of documents. The algorithm has to generate m times an n-bit string and calculate mn bitwise XOR. Thus, the index size, as well as the computation complexity is $O(mn)$.

The Trapdoor algorithm chooses two random values r_1, r_2 and a random n-bit string r_3, evaluates the permutation $\pi(\mathcal{W}, p_{r_2})$ at keyword s to find position $q_s(r_2)$, generate two n-bit strings $(f_s, g_s(r_1))$ and finally computes the two bitwise XORs on the n-bit strings. The trapdoor size and the computation complexity is $O(n)$.

In the SearchIndex algorithm, SP generates $(m+1)$ n-bit strings and computes two XORs per keyword for the re-encryption of the index. Then, SP generates and performs a random permutation on m index positions. Thus the computational complexity for SP is $O(mn)$. QP performs a simple table-lookup and calculates one XOR on a n-bit string, resulting in a complexity of $O(n)$.

Communication. Our scheme requires the index to be transferred per query. Since our index uses one bit per keyword per document (cf. Table I), the communication complexity is $O(mn)$.

The trapdoor T_1^s consists of two random values and a n-bit random string. The trapdoor T_2^s consists of an index position, i.e., a number between 1 and m, and the n-bit string k. The intermediate result I''_s of the query proxy QP that has to be transferred to SP is of size n bit.

TABLE I. Example index sizes for different document and keyword sets.

	10,000	50,000	100,000	1,000,000
100	122 kB	610 kB	1.2 MB	12 MB
250	305 kB	1.5 MB	3 MB	30 MB
500	610 kB	3 MB	6 MB	60 MB
...		...		
100,000	119 MB	596 MB	1.2 GB	11.6 GB

TABLE II. Estimated search times for a keyword search in different document/keyword sets assuming a 1 Gb/s network connection between SP and QP.

	10,000	50,000	100,000	1,000,000
100	1.6 ms	7.8 ms	16 ms	161 ms
250	3.9 ms	20 ms	39 ms	393 ms
500	7.8 ms	39 ms	79 ms	786 ms
...		...		
100,000	1.56 s	2.68 s	15.6 s	156 s

Remark. Note, that the above asymptotic complexities are similar to previous schemes with the same security guarantee [7], [25]. In practice, however, various operations make a difference for the real performance numbers. In particular, our scheme is based entirely on XOR operations and pseudo-random functions, which are orders of magnitude more efficient than other operations such as pairings. As an example, the scheme by Shen et al. [25] needs to compute $n(2m+2)$ composite order pairings per search query. For a document set of 5000 documents and 250 keywords, a search query requires 8.4 days [7]. In comparison, our scheme requires $n(2m+3)$ XOR operations and performs a search on the same dataset in less than 2 ms, assuming a 1 Gb/s network connection between SP and QP. See the example below and Table II for estimated performance numbers of different document/keyword sets.

Example. For the following example, we use a data collection of 1 million documents and a keyword list of 500 (which we consider practical in many scenarios). Then, the encrypted index is of size $500 * 1\text{M bit} = 500$ M bit or 60 MB. Using a 1 Gb/s network connection between SP and QP results in a theoretical max. transmission rate of 120 MB/s. The real max. is around 80 MB/s. To transmit an index of 60 MB takes 0.75 s at a rate of 80 MB/s. The computation on SP requires $2m+2$ XOR on n-bit strings. The query proxy QP performs one XOR on n-bit strings. An bitwise XOR on 500 million bits, takes less than 18 ms on an Intel i5 CPU M460@2.53 GHz. Per search, we require $n(2m+3)$ XORs. In our example, this results in 1,003,000,000 XOR, taking 36 ms. In total, the

search takes 786 ms. Even for a huge keyword list of 100,000 keywords and one million documents, a query takes around 2.6 minutes.

VI. Colluding servers

Recall that our security analysis assumes that the storage provider and the query proxy do not collude. In this section, we discuss the implication of SP and QP colluding.

If SP and QP collude, they can invert the permutation and encryption performed on a per-query basis (lines 4 and 5 in Figure 1). In this case, we can omit the re-encryption and permutation without further sacrificing data confidentiality. Figure 2 shows the resulting scheme, which treats SP and QP as a single server.

The original distributed scheme is reduced to a centralized scheme consisting of a client and a server. In the reduced scheme, the client sends an encrypted and permuted index to the server, and queries the server directly by sending trapdoors. Hence, the reduced scheme is in fact a "standard" SSE scheme. It is easy to see that it leaks the search and access pattern. However, we show in the next section that this reduced scheme still satisfies Curtmola et al.'s [11] definition for adaptive security.

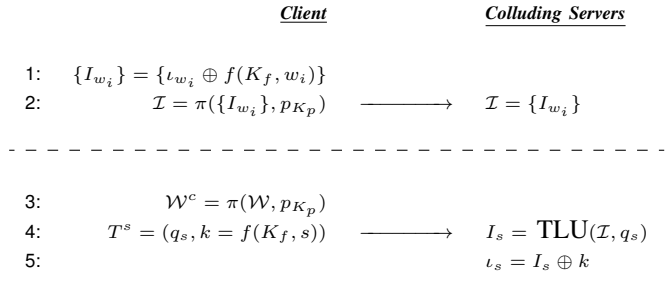

Fig. 2. SSE scheme with colluding servers.

A. The reduced scheme

As mentioned before, the reduced scheme is a plain SSE scheme which does not fall under the DSSE definition given in Section II. Therefore, we redefine this scheme in the standard SSE terminology as introduced by Curtmola et al. [11].

- $K \longleftarrow \mathsf{Gen}(1^\lambda)$: the client generates a set of three secret keys $K = (K_d, K_f, K_p)$, for the document encryption, the row encryption and the table permutation respectively. Remark that this construction does not share keys with the search provider.
- $(I, \mathbf{c}) \longleftarrow \mathsf{Enc}(K, \mathbf{D})$: the client encrypts every document in $\mathbf{D}$ with the key K_d using a PCPA secure symmetric encryption scheme, e.g. AES in CTR mode [19]. An encrypted and permuted index I is calculated as follows:
  - For every keyword w_i in $\mathcal{W}$:
    - For all documents $d_j \in \mathbf{D}$; let $\mathcal{I}[i]_j = 1$ if w_i matches d_j, otherwise set $\mathcal{I}[i]_j = 0$.
    - Reassign $\mathcal{I}[i] \longleftarrow \mathcal{I}[i] \oplus f(K_f, w_i)$, which encrypts the row $\mathcal{I}[i]$.
  - Generate a permuted index I by applying σ to the encrypted rows, such that $I = \pi(I, p_{K_p})$. Thus, for all $1 \leq i \leq m$: $I[p_{K_p}(i)] = \mathcal{I}[i]$.

  Output the encrypted and permuted index I.
- $t \longleftarrow \mathsf{Trpdr}(K, w_i)$: using the key K_p, the client can calculate the trapdoor $t = (p_{K_p}(i), f(K_f, w_i))$. The trapdoor contains the position of the row in the permuted index corresponding to w_i and the encryption/decryption key for the row.
- $X \longleftarrow \mathsf{Search}(I, t)$: given an index and a trapdoor, the server does the following:
  - it finds and decrypts the row $r = f(K_f, w_i) \oplus I[K_p(i)]$.
  - from the decrypted row, the server deduces the set of document identifiers $\{id(d_i) | d_i \in \mathbf{D} \wedge r[i] = 1\}$. Note that the server only has to know what document identifier corresponds to the i-th bit.

B. Security Analysis

In this section, we prove our reduced SSE scheme to be semantically secure against an adaptive adversary. We use the simulation-based definition for adaptive semantic security as provided in [11] (Definition 4.13).

Recall that a *history* $H = (\mathbf{D}, \mathbf{s})$ over q queries, is a tuple including the document collection and the queried keywords. We denote the keywords queried for by $\mathbf{s} = (s_1, \ldots, s_q)$ where s_i is the keyword asked for in the i-th query, and every $s_i \in \mathcal{W}$. Note there may exist a pair s_i, s_j where $i \neq j$ but $s_i = s_j$.

An *access pattern* $\alpha(H)$ from a history $H = (\mathbf{D}, \mathbf{s})$ contains the results of each query in H. Thus $\alpha(H) = (id_{s_1}(\mathbf{D}), \ldots, id_{s_q}(\mathbf{D}))$ is a vector containing the sets of document identifiers of the matched documents.

The *search pattern* $\sigma(H)$ induced from a q-query history is a $q \times q$ binary matrix such that $s_i = s_j \Leftrightarrow \sigma(H)[i][j] = 1$. If the setting is unambiguous, we write α (resp. σ) for $\alpha(H)$ (resp. $\sigma(H)$).

A *trace* $\tau(H) = (|d_1|, \ldots, |d_n|, \alpha(H), \sigma(H))$ contains the lengths of all documents in $\mathbf{D}$ and the access and search pattern induced by the input history H.

In the simulation-based definition of adaptive security by Curtmola et al. [11], the basic idea is to build a simulator which is given only the trace, and can simulate an index, ciphertexts and trapdoors that are indistinguishable from the real index, ciphertexts and trapdoors. We allow the adversary to build the history linked to the trace adaptively; the adversary can query for a keyword, receive a trapdoor and query again polynomially many times.

Theorem 2 (Security)**.** *Our reduced SSE scheme from Section VI-A is secure with respect to Curtmola et al.'s [11] definition for adaptive semantic security for SSE.*

Proof: We will first define the q-query simulator $\mathcal{S} = (\mathcal{S}_0, \ldots, \mathcal{S}_q)$ that, given a trace $\tau(H)$, generates $v^* = (I^*, c^*, t^*)$ and a state $st_{\mathcal{A}}$. The simulator $\mathcal{S}_0$ only creates an index and document ciphertexts, as no keywords have been

queried for at this stage. The i-th simulator $\mathcal{S}_i$ returns trapdoors up until the i-th query. We will then prove that no polynomial-size distinguisher D can distinguish between the distributions of v^* and the outputs of an adaptive adversary that runs the real algorithm.

- $\mathcal{S}_0(1^k, \tau(H))$: given $(|d_1|, \ldots, |d_n|)$, choose $I^* \xleftarrow{\$} \{0,1\}^{m \times n}$. Recall that m is public as it is the size of the dictionary $\mathcal{W}$, and that n is included in the trace as the number of $|d_i|$'s.
  The ciphertexts are simulated by creating random strings of the same lengths as the documents; $c_i^* \xleftarrow{\$} \{0,1\}^{|d_i|}$, where $|d_i|$ is included in the trace. Also, a random permutation $p^* : [1,m] \to [1,m]$ is generated. The simulator stores I^*, a counter $c = 0$ and a random permutation $\sigma^* : [1,m] \to [1,m]$ in the state $st_{\mathcal{S}}$, and outputs $v^* = (I^*, c^*, st_{\mathcal{S}})$.
- $\mathcal{S}_i(st_{\mathcal{S}}, \tau(H, s_1, \ldots, s_i))$ for any $1 \leq i \leq q$: given the state (which includes I^* and any previous trapdoors) and the access pattern α, the simulator can generate a trapdoor t_i^* as follows:
  Check if the keyword has been queried before; if there is a $j \neq i$ such that $\sigma[i][j] = 1$, set $t_i^* = t_j^*$. Otherwise:
  - Increase the counter by one and generate a unique row index $\sigma^*(c)$, using the counter and the random permutation. Note that the counter will never exceed m, as there are only m unique keywords.
  - Calculate a bit string $r \in \{0,1\}^n$ such that for $1 \leq j \leq n$: $r[j] = 1 \Leftrightarrow id(d_j) \in \alpha[i]$. We now have what should be the unencrypted row of the index corresponding to the keyword queried for.
  - Calculate $k^* = r \oplus I^*[\sigma^*(c)]$. We now have a dummy key which satisfies the property $k^* \oplus I^*[\sigma^*(c)] = r$.
  - Let $t_i^* = (\sigma^*(c), k^*)$

  Include t_i^* in $st_{\mathcal{S}}$, and output $(t_i^*, st_{\mathcal{S}})$.

We will now show that the outputs of $\mathbf{Real}_{SSE,\mathcal{A}}$ and $\mathbf{Sim}_{SSE,\mathcal{A},\mathcal{S}}$, being v and v^*, can not be distinguished by a distinguisher D that is given $st_{\mathcal{A}}$. Recall that $v = (I, \mathbf{c}, t_1, \ldots, t_q)$ and $v^* = (I^*, \mathbf{c}^*, t_1^*, \ldots, t_q^*)$.

- (Indistinguishability of I and I^*) The output of $f(K_f, w_i)$ is indistinguishable from random by definition of f. Therefore, the XOR of entries of the index and the output of f is indistinguishable from random bit strings of the same length [4]. Since I^* is a random bit string of the same length as I, and with all but negligible probability $st_{\mathcal{A}}$ does not contain the key, we conclude that I and I^* are indistinguishable.
- (Indistinguishability of c_i and c_i^*) Since c_i is PCPA-secure encrypted, c_i cannot be distinguished from a random string. Since every c_i^* is random and of the same length as c_i, and with all but negligible probability $st_{\mathcal{A}}$ does not contain the encryption key, c_i is distinguishable from c_i^*.
- (Indistinguishability of $t_i = (K_p(i), k = f(K_f, w_i))$ and $t_i^* = (\sigma^*(c), k^*)$) With all but negligible probability, $st_{\mathcal{A}}$ will not contain the key K_p, so the pseudo-randomness of π guarantees that each $\sigma^*(c)$ is computationally indistinguishable from $\pi(K_p, i)$.
  As stated above, I and I^* are indistinguishable, thus $I[i]$ and $I^*[i]$ are indistinguishable, thus $I[i] \oplus r = k$ and $I^*[i] \oplus r = k^*$ are indistinguishable. Thus, t_i and t_i^* are indistinguishable.

This indistinguishability shows that $\mathcal{S}$ successfully simulates the view of the adversary, which concludes the proof. ■

C. Performance Analysis

Computation and storage. The Enc algorithm is similar to the BuildIndex algorithm of our scheme in Section III: it generates an inverted index of the dictionary $\mathcal{W}$ over the documents $\mathbf{D}$. Thus, the index size and the computation complexity are $O(m \cdot n)$. Table I shows example index sizes for various document and keyword sets.

The Trapdoor algorithm calculates the position of a row and its decryption key by evaluating p_{K_p} and f. Since the decryption key is as long as a row, the trapdoor size is $O(n + \log(m))$. The computational complexity depends on the chosen pseudo-random function f.

Given a trapdoor, the server evaluates the SearchIndex algorithm by doing a table lookup and XOR'ing the resulting row with the given decryption key. The computational complexity is $O(n)$.

Communication. The trapdoor contains a row id ($O(\log m)$ bits) and the row decryption key ($O(n)$ bits). Thus, the communication complexity is $O(n + \log(m))$.

Remark. As with the DSSE scheme, the above functions are based entirely on XOR operations and pseudo-random functions.

Comparison. To demonstrate the efficiency of our scheme, we will compare it to Curtmola et al.'s [11] adaptively secure SSE scheme. Since both schemes use negligibly little computational resources (lookups and XOR's only), we focus on the sizes of trapdoors instead.

For details on Curtmola et al.'s scheme, we refer to [11] and only state here that their scheme stores document identifiers of matching documents, rather than a single bit encoding of whether a document matches a keyword. To hide the actual number of matches a document has, every document id is stored once for every possible keyword/document match. The number of possible matches equals the number of keywords a document can contain. This value, referred to as $\mathtt{max}$, is limited by two factors: the number of distinct keywords in $\mathcal{W}$ and the size of a document. The following algorithm can be used to determine $\mathtt{max}$:

- Let $i = 0$, $\mathtt{max} = 0$ and S be the document size in bytes.
- While $S > 0$:
  - If $2^{8 \cdot i} \cdot i \leq S$, set $i = i + 1$, $\mathtt{max} = \mathtt{max} + 2^{8 \cdot i}$ and $S = S - 2^{8 \cdot i} \cdot i$
  - Otherwise, set $\mathtt{max} = \mathtt{max} + \frac{S}{i}$ and $S = 0$.

- Let $\texttt{max} = \min(\texttt{max}, |\mathcal{W}|)$: if there are not enough keywords to fill the entire document, use the size of the dictionary as max value.

In [11], a trapdoor is $n \cdot \log(\texttt{max} \cdot n)$ bit. Our scheme uses trapdoors of size $\log(m) + n$ bit; a $\log(m)$ bit row id and an n bit key to decrypt it. Notice that the number of keywords hardly affects the size of a trapdoor.

We compare document sets with documents of 25 kB (i.e., $\texttt{max} \leq 12628$), to demonstrate the effect of the document size on the performance of the schemes.

TABLE III. COMPARISON OF TRAPDOOR SIZES.

Doc. size		Curtmola et al.			Our scheme
25 kB	m =100	1000	15,000	100,000	100,000
n =1,000	2.1 kB	2.5 kB	2.9 kB	2.9 kB	141 B
10,000	24.9 kB	29.0 kB	33.6 kB	33.6 kB	1.25 kB
100,000	290 kB	332 kB	378 kB	378 kB	12.5 kB
10,000,000	37.3 MB	41.5 MB	46.1 MB	46.1 MB	1.25 MB

The comparison in table III indicates that our scheme outperforms Curtmola et al.'s scheme in terms of trapdoor sizes in the given setting. We believe that our scheme is of interest even outside the context of this paper due to its conceptual simplicity and high efficiency.

VII. CONCLUSION

In this paper, we have explored SSE in a distributed setting and proposed the concept of distributed searchable symmetric encryption (DSSE) for outsourcing encrypted data. Compared with standard SSE, a DSSE scheme can potentially provide more efficiency and better security guarantees. We described a security model that in addition to previous models also protects the search pattern. We proposed a construction for DSSE (based entirely on binary XOR operations and pseudo-random functions) which is highly efficient, despite the additional security. The scheme uses an inverted index approach and borrows re-shuffling techniques from private information retrieval. The main idea is, that the query proxy gets a fresh (i.e., re-encrypted and shuffled) index per query. Thus, the query can be realized by a simple table look-up, without revealing the search pattern.

We have also shown that even if the storage provider and query proxy collude, the scheme is still secure under Curtmola et al.'s definition for adaptive semantic security for SSE. The resulting SSE scheme when the two servers collude is very efficient and outperforms Curtmola et al.'s scheme in terms of trapdoor sizes.

REFERENCES

[1] Todd W. Arnold, Carl U. Buscaglia, F. Chan, Vincenzo Condorelli, John C. Dayka, W. Santiago-Fernandez, Nihad Hadzic, Michael D. Hocker, M. Jordan, T. E. Morris, and Klaus Werner. IBM 4765 Cryptographic Coprocessor. *IBM Journal of Research and Development*, 56(1):10, 2012.

[2] Todd W. Arnold and Leendert van Doorn. The IBM PCIXCC: A New Cryptographic Coprocessor for the IBM eServer. *IBM Journal of Research and Development*, 48(3-4):475–488, 2004.

[3] Dmitri Asonov. *Querying Databases Privately: A New Approach to Private Information Retrieval*, volume 3128 of *Lecture Notes in Computer Science*. Springer, 2004.

[4] Mihir Bellare, Anand Desai, Eron Jokipii, and Phillip Rogaway. A Concrete Security Treatment of Symmetric Encryption. In *Foundations of Computer Science*, pages 394–403. IEEE, 1997.

[5] Steven M. Bellovin and William R. Cheswick. Privacy-Enhanced Searches Using Encrypted Bloom Filters. *IACR Cryptology ePrint Archive*, 2004:22, 2004.

[6] Dan Boneh, Craig Gentry, Shai Halevi, Frank Wang, and David J. Wu. Private Database Queries Using Somewhat Homomorphic Encryption. In *ACNS*, pages 102–118, 2013.

[7] Christoph Bösch, Qiang Tang, Pieter H. Hartel, and Willem Jonker. Selective Document Retrieval from Encrypted Database. In *ISC*, pages 224–241, 2012.

[8] Yan-Cheng Chang and Michael Mitzenmacher. Privacy Preserving Keyword Searches on Remote Encrypted Data. In *ACNS*, pages 442–455, 2005.

[9] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private Information Retrieval. *Journal of the ACM*, 45(6):965–981, November 1998.

[10] Emiliano De Cristofaro, Yanbin Lu, and Gene Tsudik. Efficient Techniques for Privacy-Preserving Sharing of Sensitive Information. In *TRUST*, pages 239–253, 2011.

[11] Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions. *Journal of Computer Security*, 19(5):895–934, 2011.

[12] Eu-Jin Goh. Secure Indexes. Cryptology ePrint Archive, Report 2003/216, 2003.

[13] Oded Goldreich. *The Foundations of Cryptography - Volume 2, Basic Applications*. Cambridge University Press, 2004.

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

[15] Seny Kamara and Charalampos Papamanthou. Parallel and Dynamic Searchable Symmetric Encryption. In *Financial Cryptography*, pages 258–274, 2013.

[16] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic searchable symmetric encryption. In *ACM Conference on Computer and Communications Security*, pages 965–976, 2012.

[17] Murat Kantarcioglu and Chris Clifton. Security Issues in Querying Encrypted Data. In *DBSec*, pages 325–337, 2005.

[18] Kaoru Kurosawa and Yasuhiro Ohtaki. UC-Secure Searchable Symmetric Encryption. In *Financial Cryptography*, pages 285–298, 2012.

[19] Helger Lipmaa, David Wagner, and Phillip Rogaway. Comments to nist concerning aes modes of operation: Ctr-mode encryption. 2000.

[20] Chang Liu, Liehuang Zhu, Mingzhong Wang, and Yu an Tan. Search Pattern Leakage in Searchable Encryption: Attacks and New Construction. *Inf. Sci.*, 265:176–188, 2014.

[21] Rafail Ostrovsky. Efficient Computation on Oblivious RAMs. In *STOC*, pages 514–523, 1990.

[22] Rafail Ostrovsky. *Software Protection and Simulations on Oblivious RAMs*. PhD thesis, MIT, 1992.

[23] Andreas Peter, Erik Tews, and Stefan Katzenbeisser. Efficiently outsourcing multiparty computation under multiple keys. *IEEE Transactions on Information Forensics and Security*, 8(12):2046–2058, 2013.

[24] Mariana Raykova, Binh Vo, Steven M. Bellovin, and Tal Malkin. Secure Anonymous Database Search. In *CCSW*, pages 115–126, 2009.

[25] Emily Shen, Elaine Shi, and Brent Waters. Predicate Privacy in Encryption Systems. In *TCC*, pages 457–473, 2009.

[26] Dawn Song, David Wagner, and Adrian Perrig. Practical Techniques for Searches on Encrypted Data. In *IEEE Symposium on Security and Privacy*, pages 44–55, 2000.

[27] Peter van Liesdonk, Saeed Sedghi, Jeroen Doumen, Pieter H. Hartel, and Willem Jonker. Computationally Efficient Searchable Symmetric Encryption. In *Secure Data Management*, pages 87–100, 2010.

[28] Peishun Wang, Huaxiong Wang, and Josef Pieprzyk. An Efficient Scheme of Common Secure Indices for Conjunctive Keyword-Based Retrieval on Encrypted Data. In *WISA*, pages 145–159, 2008.

[29] Peishun Wang, Huaxiong Wang, and Josef Pieprzyk. Keyword Field-Free Conjunctive Keyword Searches on Encrypted Data and Extension for Dynamic Groups. In *CANS*, pages 178–195, 2008.