Zachary A. Kissel* and Jie Wang

Generic Adaptively Secure Searchable Phrase Encryption

Abstract: In recent years searchable symmetric encryption has seen a rapid increase in query expressiveness including keyword, phrase, Boolean, and fuzzy queries. With this expressiveness came increasingly complex constructions. Having these facts in mind, we present an efficient and generic searchable symmetric encryption construction for phrase queries. Our construction is straightforward to implement, and is proven secure under adaptively chosen query attacks (CQA2) in the random oracle model with an honest-but-curious adversary. To our knowledge, this is the first encrypted phrase search system that achieves CQA2 security. Moreover, we demonstrate that our document collection preprocessing algorithm allows us to extend a dynamic SSE construction so that it supports phrase queries. We also provide a compiler theorem which transforms any CQA2-secure SSE construction for keyword queries into a CQA2-secure SSE construction that supports phrase queries.

Keywords: Privacy, Searchable Encryption, Phrase Queries, Dynamic Searchable Encryption, Clouds

DOI 10.1515/popets-2017-0002
Received 2016-05-31; revised 2016-09-01; accepted 2016-09-02.

1 Introduction

Searchable Symmetric Encryption (SSE) is a mechanism that allows untrusted storage providers, such as clouds, to execute search queries over encrypted data. In particular, an SSE system involves a client issuing encrypted search queries to the cloud, and the cloud returning the results of the search query without learning the query itself.

SSE has been studied extensively since its initial construction by Song, Wagner, and Perrig in 2000 [18]. Publications that followed tended to focus on keyword search [6, 7, 9, 18, 22]. These early works quickly converged on an inverted index approach. An inverted index is a data structure that stores what keywords are present in which documents. This index is encrypted in a special way so that encrypted queries, known as *tokens* or *trapdoors*, may be evaluated over the encrypted index. As a result of the query, the SSE system returns either identifiers for the matching documents or the encrypted documents themselves.

There is no reason to limit SSE queries to just keywords. Recent work has provided the primitive with more expressive queries. There are now SSE systems that support Boolean queries [5], fuzzy keyword queries [14], and phrase queries [13, 21] (our focus).

SSE systems that support phrase queries provide several enhancements and advantages over traditional SSE. First, phrase queries allow for higher precision in queries than keywords or Boolean queries. For example, phrase queries can be used to retrieve only documents that contain “red bicycles” instead of having to search for all documents that contain “red” and “bicycles”. Second, phrase queries provide the ability to perform certain context-sensitive searches. By this we mean that phrase queries allow for context to be added to keywords. For example, a law enforcement agency may be concerned with searching for a “red Kona mountain bike” in encrypted files containing theft reports. In a keyword based system or Boolean query system this query could return many additional results depending on the definition of keywords. Third, unlike keyword queries, phrase queries allow for arbitrarily long queries, and thus arbitrarily specific queries to be executed over the encrypted documents.

In this paper, we focus on searchable encryption for phrase queries. We seek to reduce the complexity of phrase query systems through the introduction of a generic construction. In particular, we construct an efficient and easy to implement phrase search system based on the work of Cash et al. [4] (Section 6). Building on insight developed from this construction, we introduce an encrypted phrase search mechanism that makes black-box use of any underlying keyword encrypted search mechanism (Section 10). Black-box constructions for

***Corresponding Author: Zachary A. Kissel:** Merrimack College Department of Computer Science, E-mail: kisselz@merrimack.edu
Jie Wang: University of Massachusetts Lowell Department of Computer Science, E-mail: wang@cs.uml.edu

cc BY-NC-ND

phrase SSE benefit from any improvements to the underlying building blocks.

Our construction has the following contributions:

– It is CQA2 secure which, to our knowledge, is the first such construction for phrase queries.
– It operates using a single round of communication between the client and cloud to retrieve matching documents. This is a substantial improvement over the existing multi-round systems of Tang et al. [21] and Kissel and Wang [13].
– It supports parallel processing of queries.
– It is straight forward to implement.

In addition to our basic construction, we also offer two additional contributions important in their own right; they are

1. A CQA2 dynamic SSE construction that supports phrase queries;
2. A compiler theorem which translates *any* SSE system for keyword queries to an SSE system for phrase queries.

This paper is organized as follows. In Section 2 we will discuss previous work on SSE. In Section 3 and 5 we will set up the fundamental model of SSE and define our notations. In Section 4 we will discuss the necessary background information to understand our construction presented in Section 6. In Section 7 we will discuss the security issues in our construction. In Section 9 we will show how to use our preprocessing mechanism in a different classic keyword-based system. We present our compiler theorem in Section 10 and conclude with final remarks in Section 11.

2 Previous Work

Encrypted search can be based on either symmetric encryption or asymmetric encryption, first studied by, respectively, Song et al. [18] and Boneh et al. [1]. We will only consider symmetric searchable encryption in this paper. For a timely survey of the state of searchable encryption research we recommend Bösch et al.'s article [2].

Security for SSE was first formalized by Curtmola et al. [8]. They introduced two notions of security chosen-query attack 1 (CQA1) and chosen-query attack 2 (CQA2). In the CQA1 setting the attacker is not allowed to make adaptive queries that are based on observing previous results. In the CQA2 setting the attacker is allowed to make adaptive queries; thus CQA2 is a strictly stronger form of security. In addition to formalizing these two notions of security Curtmola et al. provided constructions based on linked lists that satisfied both CQA1 and CQA2.

An early construction of phrase queries is due to Tang et al [21]. Their construction operates as a two-phase protocol. In the first phase, the cloud retrieves the document identifiers for documents that contain all the words in the phrase provided by the client, and returns the identifiers to the client. This phase relies on a global index shared among all documents in the cloud. In the second phase, the client sends a query and a list of document identifiers to the cloud. The cloud searches for an exact phrase match for each document in the per-document index and returns to the client the actual encrypted documents that match the phrase. Their protocol, however, only provides security under the honest-but-curious (HBC) adversarial model. Tang et al. further formulated, a definition of CQA1 security for phrase encryption based on the work of Curtmola et al. [8] and showed that their construction satisfies this definition of CQA1 security.

Kissel and Wang [13] devised an encrypted phrase search scheme for the semi-honest-but-curious adversarial model. In this model the adversary is very similar to an HBC adversary except that the adversary (cloud) may return incomplete or incorrect results [6]. Often times a system that provides security in the presence of adversaries that lie about the number of results are called verifiable. Loosely speaking, their construction combines ideas from [21], which uses essentially a hybrid Merkle tree [15], and [6]. In particular, they replaced the first phase of the scheme of [21] with the verifiable keyword index of [6], and modified the second phase of [21] to include verification. These changes make both phases of the scheme of [21] verifiable. They showed that the resulting construction is CQA1 secure.

Recent advances greatly simplified and improved the IO efficiency of SSE constructions for the HBC model. Notably, the work Cash et al. [4] simplified constructions of SSE while achieving IO efficiency. Their method allows any arbitrary history independent dictionary data structure to be used as a black box in constructing an SSE system. In addition, their system is clean, easily implemented, and has a form that is CQA2 secure. We will further elaborate on their contributions in Section 4.

Contemporary work is moving towards dynamic SSE. In a dynamic form of SSE, the collection is allowed to be modified. In indexed based constructions this in-

volves securely modifying the index. A trivial solution is to download the collection, modify, and then reencrypt. These solutions are mentioned in several early papers and newer papers provide better results. For example, Kamara et al. [12] presented a dynamic encrypted linked list based index and Kamara and Papamanthou [11] presented a dynamic encrypted red-black tree. While efficient, the constructions in [12] and [11] offer weak security around updates. Non-indexed based approaches have also been tried. Specifically, Stefanov, Papamanthou, and Shi created a non-index based solution [19, 20]. Their solution offers what is known as forward security. Loosely speaking, forward security means that insertion of new data into the collection does not allow an attacker to infer if the new data would match a previously issued query.

3 Searchable Phrase Encryption Model and Notations

We extend the notations of Curtmola et al. [8] to obtain notations for encrypted phrase search.

3.1 Notations

Let $\mathsf{DB} = (\mathsf{id}_i, W_i, L_i)_{i=1}^d$ denote a corpus of d documents, where $\mathsf{id}_i \in \{0,1\}^\lambda$ is a unique document identifier, $W_i \subset \{0,1\}^*$ a set of keywords in document id_i, and

$$L_i : W_i \to \mathcal{P}(\mathbb{N}),$$

where $\mathcal{P}(\cdot)$ denotes the power set. L_i is a mapping from a word to a set of location information for document i (the meanings of L_i will be made clear later). Let W be the set of all words, namely, $W = \bigcup_i^d W_i$. Let

$$\mathsf{DB}(w) = \{(\mathsf{id}_i, u) : w \in W \text{ and } u \in L_i(w)\}.$$

Note that in the notations for keyword-based SSE there is no word-location mapping L_i.

We denote by $w_1 \parallel w_2$ the concatenation of words w_1 and w_2, which is often denoted by $w_1 w_2$ when there is no confusion.

We denote a semantically secure symmetric encryption algorithm by (G, E, D), where G is a key generation algorithm that takes a security parameter as input and generates a key of the desired size, E a symmetric key encryption algorithm that takes a key and a message as input and returns a ciphertext, and D a decryption algorithm that takes a key and ciphertext as input and returns the original message. In addition, we will use a family pseudo-random functions F that are indexed by a secret key. Each member of the family will take a word as input and produce a binary output.

3.2 Model

An SSE scheme is a collection of four polynomial-time algorithms

$$\mathsf{SSE} = (\mathsf{Setup}, \mathsf{BuildIndex}, \mathsf{Token}, \mathsf{Search}).$$

Formally we have:

- $\mathsf{Setup}(1^\lambda)$. This is a probabilistic key generation algorithm run by the data owner, who is also the client. It takes a unary notation of λ, a security parameter, as input and returns a secret key K such that the length of K is polynomially bounded in λ.
- $\mathsf{BuildIndex}(K, \mathsf{DB})$. This is a (possibly probabilistic) algorithm run by the data owner. It takes as input the secret key K and a document collection DB that is polynomially bounded in λ, and returns an encrypted index EDB such that the length of EDB is polynomially bounded in λ.
- $\mathsf{Token}(K, w)$. This is run by the data owner. It takes the secret key K and a word w as input, and returns a token T_w.
- $\mathsf{Search}(\mathsf{EDB}, T_w)$. This is run by the cloud C. It takes an index EDB for a collection DB and the token T_w for word w as inputs, and returns $\mathsf{DB}(w)$, the set of identifiers of documents containing w.

In our construction, we will slightly modify Token and Search such that we operate on a phrase instead of a keyword. Specifically Token will be given a sequence of words as input and produce a sequence of tokens. Similarly, Search will be provided with a sequence of tokens previously produced by a single call to Token.

We say that a SSE scheme is *correct* if executing Search with a token T_w for word $w \in W$, returns $\mathsf{DB}(w)$ with negligible probability of failure.

Intuitively we say that a phrase system is secure, if a polynomial (in the security parameter) time adversary can not learn anything meaningful about the document collection from queries and their associated results. The adversary should also not learn anything meaningful about the query. A more formal discussion of security is deferred to Section 7. For our purposes we will constrain our work to HBC adversaries.

4 Background on Generic Keyword SSE

Cash et al. [4] recently presented an elegant SSE construction that uses an generic history-independent dictionary data structure.

A dictionary data structure T is any data structure that supports three operations: Insert, Lookup, and Delete. We define these operations as follows:

- Insert (T, k, v). This function inserts a value v with key k into the dictionary T.
- Lookup (T, k). This function returns the values in T that are associated with key k. If there are no values then $\perp$ is returned.
- Delete (T, k). This function removes the value in dictionary T associated with key k. It further removes the key k if there is no other associated data.

We say that a dictionary is *history independent* [16, 17] if any two sequences of Insert and Delete operations will create the same content and generate the same distribution over the memory representation of the dictionary [17]. Essentially, the dictionary only depends on what is inserted and deleted, *not* the order in which these operations occur. As mentioned in [4], a simple way to achieve history independence for dictionaries is to sort all the key-value pairs *before* insertion into the dictionary. If one wishes to remove dependence on history-independent data structures this approach could be used. Provided all inserts were done at one time.

In its basic form, known as Π_{Bas} in [4], the system of Cash et al. begins by building a list of key-value pairs for every $w \in W$:

$$(F(K_1, c), E_{K_2}(\mathsf{id})),$$

where the value c denotes the c-th occurrence of w in DB, $\mathsf{id} \in \mathsf{DB}(w)$,

$$\begin{aligned} K_1 &= F(K, 1 \parallel w), \\ K_2 &= F(K, 2 \parallel w), \end{aligned}$$

K is a secret key generated by $\mathsf{Setup}\left(1^\lambda\right)$, and F is a pseudo-random function. The value $F(K_1, c)$ is also referred to as a pseudo-random label. The system proceeds to insert the sorted key-value pairs into a history-independent dictionary (e.g., the hash table of [17]). To search for a keyword w, we first generate a search token pair: $(K_1 = F(K, 1 \parallel w),\ K_2 = F(K, 2 \parallel w))$. The cloud uses these tokens to execute Search queries on the dictionary. The resulting document identifiers $\mathsf{DB}(w)$ are returned to the client.

Any SSE construction will leak some information about a client's interaction with the cloud. Any adversary, including the cloud, that observes this leakage will be able to exploit the leakage to learn information about the data collection. We break the leakage of any construction into the following two types:

1. What is leaked about the data collection from the existence of the encrypted index EDB.
2. What is leaked by a query and its processing.

Formally, we will denote the leakage of any SSE construction using two *stateful* leakage functions $\mathfrak{L}_1$ and $\mathfrak{L}_2$, where $\mathfrak{L}_1$ captures what is leaked about DB, and $\mathfrak{L}_2$ captures what is leaked as a result of issuing queries. By assumption, both $\mathfrak{L}_1$ and $\mathfrak{L}_2$ implicitly share state.

In Π_{Bas} the leakage stemming from the encrypted index, $\mathfrak{L}_1(\mathsf{DB})$, is the size of the index. The leakage from the processing of a query, $\mathfrak{L}_2(q)$, is: the number of documents that match an issued query q for word w (i.e., $|\mathsf{DB}(w)|$); the document identifiers that match an issued query (i.e., $\mathsf{DB}(w)$); and if a query has been repeated.

5 A Trivial Phrase System

Using Π_{Bas}, with little modification, we can obtain a generic SSE system that supports phrase search. The idea is to tag every entry in EDB with location information. The location information for a word is given by the function L_i which provides the position(s) of the word in document i. Specifically, for $w \in W$ we insert the key-value pair

$$\left(F\left(K_1, c\right), E_{K_2}\left(\mathsf{id}, \text{curr}, \text{next}\right)\right),$$

where $(\text{curr}, \text{next}) \in L'_{\mathsf{id}}(w)$. The function L'_{id} is defined as the set of pseudo-random labels for every entry in L_{id}. Specifically,

$$L'_i(w) = \left\{\left(F\left(K_3, x\right), F\left(K_3, x+1\right)\right) : x \in L_i(w)\right\},$$

where K_3 is a key sampled uniformly at random from $\{0,1\}^\lambda$ during Setup. We note that key K_3 is *not* present in the original construction due to Cash et al and is used for the entire document collection.

To generate a search token for a phrase $p = w_1 w_2 \cdots w_k$, we must generate k search tokens. The Search operation proceeds to follow the general Search operation of Π_{Bas}. Once all the results are collected, the curr and next information is used, by the storage provider, to determine documents that contain the

phrase p. The storage provider performs this operation as follows:

1. Collect all document identifiers and the associated location information present in results for w_1 into a set S. In other words, compute $\mathsf{DB}(w_1)$ and add the results to S.
2. Processing in succession for every $i > 1$ and word $w_i \in p$:
   (a) Collect into set S' all document identifiers where the value of curr in the w_i results match the value of next in the w_{i-1} results. Formally,
   $$S' = \{(\mathsf{id}, \text{curr}, \text{next}) \in \mathsf{DB}(w_i) : \exists \left(\mathsf{id}, \text{curr}', \text{next}'\right) \in S \text{ where } \text{curr} = \text{next}'\} .$$
   (b) Set $S = S'$.
3. All of the document identifiers, in S, are sent to the client.

5.1 Analysis of the Trivial System

The trivial system allows for arbitrarily long phrase queries and parallel search. Thus most of our goals are satisfied. The security of our system is sufficient as well (can be shown to be CQA2 secure). Observe that our trivial system exhibits a minor addition in the what is leaked about the execution and results of a query. Specifically due to position obfuscation, only relative word position relationships are leaked in a given document.

The efficiency of the trivial construction can be improved. In the trivial construction, EDB is very large which directly impacts the time required to execute a query. In particular, the size of EDB is proportional to $\sum_{w \in W} |\mathsf{DB}(w)|$. Because of the size of EDB, the Search operation wastes time following false positives. Statistically, most of the entries we check will *not* be useful in processing the final query. This is because the the trivial system checks every occurrence of a word in the document collection. This leads to the processing of entries in EDB for documents that only contain at most one word of the query. We proceed to improve this situation.

6 An Efficient SSE Construction for Phrase Search

For greatest generality, we base our construction on Π_{Bas} given in [4]. We, however, take a very different approach in the construction of DB. Specifically, we must add a significant one-time preparation phase in order to construct a suitable DB. A key piece of our preparation phase is inspired by next-word indexing [23].

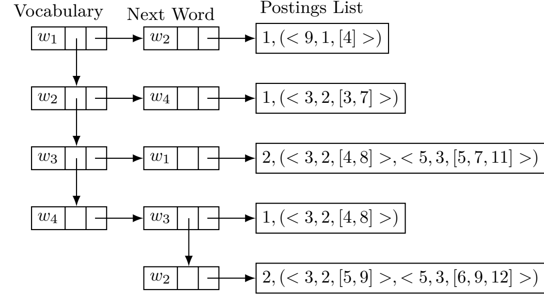


Fig. 1. A sample next-word index.

6.1 Next-word indexing

A Next-word index is an inverted index structure consisting of the following three components:

1. a vocabulary list of every word w_i in W;
2. a set of next-word lists consisting of each word $w_j \in W$ that directly follows w_i in some document in DB;
3. a set of postings list information consisting of, for each pair of words (w_i, w_j), the number of documents that contain the pair and a set of lists. Each list consists of the document identifier that contains the pair, the number of occurrences of the pair in that document, and the list of locations of the pairs in that document (see Fig. 1).

For example, a posting-list entry of "$1, (\langle 9, 1, [4] \rangle)$", in Figure 1, means that the word pair (bigram) $w_1 w_2$ occurs in one document in the entire database DB. In particular, it occurs once in document 9 at position 4.

To search for a phrase $p = w_1 w_2 \cdots w_k$ in a next-word index we proceed as follows:

1. Allocate an empty set S.
2. Processing in succession for every i in the range $1 \leq i < k$:
   (a) Find the next-word list associated with w_i.
   (b) Walk the next-word list until you find w_{i+1}, if not found the result set is empty and the search stops.
   (c) Collect into set S' all document identifiers and locations.
   (d) If $S \neq \emptyset$, for each document identifier d in S'

i. Let L'_d be the list of locations associated with document d in set S'. Let L_d be the list of locations associated with document d in set S.

ii. If L_d is does not exist, remove data associated with d from S'

iii. For every entry $l \in L'_d$ check if $(l-1) \in L_d$. If $(l-1) \notin L_d$ then, remove l from L'_d. If L_d becomes empty, remove all data associated with d from S'.

(e) Set $S = S'$.

3. All of the document identifiers in S are the results of the search.

The next-word index is a very efficient data structure that is often used in phrase searching in the information retrieval space. As mentioned in [23], the next-word index allows for faster phrase queries as the next-word index is smaller than an index that catalogs the positions of every word in the document collection. In addition, there is only a limited reduction in query expressiveness. Namely, single word queries are not possible in the standard next-word index. However, to overcome this problem, an inverted index of just the words and their associated document identifiers can be constructed to resolve single word queries.

It is possible to generalize the next-word indexing technique to longer subphrases. Generalizing to a length n subphrase will require $n-1$ additional inverted indexes to maintain phrase search. Observe that an index for a subphrase of length n can only identify phrases p with $|p| \geq n$. This means our $n-1$ subphrase indexes are used to track correct location information for phrases longer than one word and shorter than n. Thus, the space optimal subphrase length for the next-word index is $n = 2$.

6.2 Preparing the Collection

Our preparation of the collection for phrase search takes its inspiration from the observation that the next-word index structure can be linearized while still retaining its power. In a next-word index every pair of words has a posting list associated with it, where the posting list contains a series of document identifiers along with the locations in the identified document. The preparation phase seeks to collapse a next-word index into a key-value pair format, defined as follows:

- **Key**. The keys are pairs of consecutive words in the next-word index.
- **Value**. The value for each key is an ordered list of the identifier of a document that contains the key and a location in the document.

The following is an example of a key-value pair: $(w_1w_2, (\mathsf{id}, l))$, where w_1w_2 appears in document id at location l (i.e., $l \in L_i(w_1w_2)$). To handle all entries in the posting list, multiple key-value pairs must be created that share the same key.

If we wish to support queries with a length of one word, we must insert a special key-value pair $(w_1 \perp, (\mathsf{id}, l))$, where w_1 appears in id, $\perp$ is a special slug that does not appear as a word in the document collection, and l is a unique location beyond the last word in the document.

In what follows we will use WP_i to denote the set of all consecutive word pairs w_jw_{j+1} in the document with document identifier id_i. Let WP be the set of all unique bigrams in the entire collection.

For every document identifier id_i $(1 \leq i \leq d)$, the client constructs a list of key-value pairs

$$D_{\mathsf{id}_i} = \{(w_1w_2, (\mathsf{id}_i, l)) : w_1w_2 \in WP_i \text{ and } l \in L_i(w_1w_2)\}.$$

The key-value pairs need to undergo obfuscation to reduce potential query time ($\mathfrak{L}_2$) leakage. Since the key-value pairs are to be stored in the cloud, they must be encrypted. However, we also want the cloud to be able to link the current location of a word pair to the next word pair in order to carry out phrase search.

In particular, the client proceeds to generate a key $K_3 \in \{0,1\}^*$ sampled uniformly at random. Then for every list D_{id_i}, replaces each location l with (curr, next), where $\text{curr} = F(K_3, l)$ and $\text{next} = F(K_3, l+1)$.

To build the database DB, the client inserts every key-value pair into the collection. To do this, construct, for each $w_1w_2 \in WP_i$, the following set:

$$L'_i(w_1w_2) = \{(\text{curr}, \text{next}) : (w_1w_2, (\mathsf{id}_i, l)) \in D_{\mathsf{id}_i}\},$$

where $l \in L_i(w_1w_2)$. Thus our collection becomes

$$\mathsf{DB} = \left(\mathsf{id}_i, WP_i, \left\{L'_i(w_1w_2) : w_1w_2 \in WP_i\right\}\right)_{i=1}^{d}.$$

```
1: function SETUP(1^λ)
2:     Sample K uniformly from {0,1}^λ
3:     return K
4: end function

1: function BUILDINDEX(K, DB)
2:     Allocate empyt list L
3:     for each two word phrase (w1 || w2) ∈ WP do
4:         K1 ← F(K, 1 || (w1 || w2))
5:         K2 ← F(K, 2 || (w1 || w2))
6:         c ← 0
7:         for each (id_i, cur, next) ∈ DB(w1 || w2) do
8:             ℓ ← F(K1, c)
9:             d ← E_K2(id_i || cur || next)
10:            c ← c + 1
11:            ▷ Insert in L lexicographically by ℓ
12:            INSERT(L, (ℓ, d))
13:        end for
14:    end for
15:    allocate dictionary EDB
16:    for each (ℓ, d) ∈ L do
17:        INSERT(EDB, ℓ, d)
18:    end for
19:    return EDB
20: end function

1: function TOKEN(K, p)
2:     Parse p as w1 || w2 || ··· || w|p|
3:     Tp ← ⟨⟩
4:     for i ← 1 to |p| − 1 do
5:         Append (F(K, 1 || (w_i || w_{i+1})),
            F(K, 2 || (w_i || w_{i+1}))) to sequence Tp
6:     end for
7:     return Tp
8: end function

1: function SEARCH(EDB, Tp)
2:     Allocate empty list list1
3:     first ← True
4:     n ← 0
5:     for each pair (K1, K2) ∈ Tp do
6:         n ← n + 1
7:         c ← 0
8:         Allocate an empty list list2
9:         repeat
10:            d ← LOOKUP(EDB, F(K1, c))
11:            if d ≠ ⊥ then
12:                (id, cur, next) ← D_K2(d)
13:                if first = True then
14:                    INSERT(list1, (id, cur, next))
15:                else            ▷ Check for match in list1.
16:                    if MATCH(list1, id, cur) = True then
17:                        INSERT(list2, (id, cur, next))
18:                    end if
19:                end if
20:            end if
21:            c ← c + 1
22:        until LOOKUP returns ⊥
23:        if first = False then
24:            list1 ← list2
25:        end if
26:        first = False
27:    end for
28:    Using list1, return the set of matching document
       identifiers, DB(p), to the client.
29: end function
```

Fig. 2. The Π_{Bas} (Cash et al.) protocol extended for encrypted phrase search

6.3 Instantiating the system

The client instantiates the operations as seen in Figure 2. Given a security parameter λ, the client is responsible for running Setup to generate a new key K and preparing the collection DB. From the collection, the client constructs an index EDB using the BuildIndex algorithm and forwards the index to the cloud. Letting $WP = \cup_{i=1}^{d} WP_i$, the index is constructed as follows: For each key $w_1 w_2 \in WP$, let

$$K_1 = F(K, 1 \parallel w_1 w_2), \; K_2 = F(K, 2 \parallel w_1 w_2).$$

For every key $w_1 w_2$ found in DB, suppose that $w_1 w_2$ appears in the c-th record, the client assigns to it a unique label

$$\ell = F(K_1, c).$$

This label will become the key for a corresponding entry in EDB. The value associated with this label is

$$E_{K_2}(\mathsf{id} \parallel \mathrm{cur} \parallel \mathrm{next}),$$

where $\mathsf{id} \parallel \mathrm{cur} \parallel \mathrm{next}$ comes from the c-th occurrence of w in DB. That is, the corresponding entry in EDB is the following key-value pair:

$$(F(K_1, c), E_{K_2}(\mathsf{id} \parallel \mathrm{cur} \parallel \mathrm{next})).$$

6.4 Query

To run a query on a phrase $w_1 w_2 \cdots w_k$, the client makes a call to Token to get a search token, for *each* consecutive pair of words:

$$w_1 w_2, w_2 w_3, \cdots, w_{k-1} w_k.$$

The resulting tokens are

$$\bigcup_{i=1}^{k-1} \left(F\left(K, 1 \parallel w_i w_{i+1}\right), F\left(K, 2 \parallel w_i w_{i+1}\right)\right).$$

These tokens are then given to the cloud.

6.5 Search

Upon receiving the query tokens, the cloud runs the Search algorithm to search for the document identifiers that contain the query phrase. These identifiers are then returned to the client. The Search algorithm treats the token as a sequence of pairs. For each pair, the cloud will generate a series of key labels by incrementing an initially zero value c. The cloud will stop once a Lookup operation on EDB fails.

Processing the first pair in the token, the cloud decrypts the results of each Lookup operation and stores it in a list list_1. On subsequent pairs from the tokens, the cloud creates a new list list_2, and potentially adds an entry to the list list_2 after each Lookup operation. In order for an entry to be added to list_2, it must be the case that the value returned from the Lookup operation matches an entry in list_1. An entry is *matched* if the entry has both the same document identifier as the result *and* the entry's next value matches the curr value in the result (represented by the function MATCH in Figure 2). After all Lookup operations are completed for a pair in the token, list_2 becomes the new list_1. After the entire token has been processed the document identifiers from list_1 are returned to the client.

It should be noted that thanks to the structure of the index EDB, the Search procedure can easily be parallelized. We inherit this feature from the base Π_{Bas} protocol. We note that parallel search is not a feature presented in previous encrypted phrase search systems. Moreover, EDB is space and time optimal.

7 Security

To establish security we must bring our construction into the random oracle model (ROM). In particular, we assume the existence of a random oracle

$$\mathcal{H} : \{0,1\}^{2\lambda} \to \{0,1\}^{\lambda}$$

and define our cryptographic operations, either all or in part, using this oracle. In particular, we define a pseudo-random function F by

$$F(K, x) = \mathcal{H}(K \parallel x).$$

We define encryption on plaintext messages m drawn from $\{0,1\}^{\lambda}$ as

$$E_K(m) = (r, \mathcal{H}(K \parallel r) \oplus m),$$

where r is drawn uniformly at random from $\{0,1\}^{\lambda}$. Notice that this is very similar to the standard CPA secure construction of encryption based on pseudo-random functions.

7.1 Leakage

We will establish two stateful leakage functions $\mathfrak{L}_1$ and $\mathfrak{L}_2$ that implicitly share state information. These functions describe what information is leaked by our system. The function $\mathfrak{L}_1$ describes what is leaked to an adversary about the collection DB. This information is solely what is available from looking at EDB. In the case of our protocol, the leakage is just the number of documents associated with each word pair. Formally,

$$\mathfrak{L}_1(\mathsf{DB}) = \sum_{w_1 w_2 \in WP} \left|\mathsf{DB}(w_1 w_2)\right|.$$

The collection DB is saved as state information.

The $\mathfrak{L}_2$ leakage function tracks what is leaked as a result of issuing queries. Specifically, this is the results of each subquery including obfuscated location information, if a query or portion of it has been repeated, and the collection of document identifiers that are ultimately returned to the client. This leakage is tracked as described below:

The first time the $\mathfrak{L}_2$ function is invoked on a query q, denoted by $\mathfrak{L}_2(q)$, an empty list called the query list, denoted by Q, is created as well as a special counter q_{cnt} that is set to one. For a query $q = w_1 w_2 \cdots w_{|q|}$, the $\mathfrak{L}_2$ leakage is formally defined by

$$\begin{aligned}
\mathfrak{L}_2(q) &= (\mathsf{DB}_q, \mathsf{SP}_q), \\
\mathsf{DB}_q &= \left\{\mathsf{DB}'(w_i w_{i+1}) : 1 \le i < |q|\right\}, \\
\mathsf{SP}_q &= \left\{\mathsf{SP}(w_i w_{i+1}) : 1 \le i < |q|\right\},
\end{aligned}$$

where $\mathsf{DB}'(w_i w_{i+1})$ is defined as

$$\mathsf{DB}'(w_j w_{j+1}) = \{(\mathsf{id}_i, (u,v)) : w_j w_{j+1} \in WP_i \text{ and } (u,v) \in L'_i(w_j w_{j+1})\}$$

and $\mathsf{SP}(w_i w_{i+1}) = \{j : (j, w_i w_{i+1}) \in Q\}$. Intuitively, the set SP_q is the *search pattern* consisting of the identifiers for repeated subqueries. The set DB_q is sometimes referred to as the *query results*.

Every call to $\mathfrak{L}_2$ results in the set of tuples

$$\{(q_{\mathrm{cnt}} + (i-1), w_j w_{j+1}) : 1 \leq j < |q|\}$$

being inserted into the list Q followed by q_{cnt} being incremented by $|q|$. It should be noted that due to the obfuscation of the locations of bigrams within a document, the $\mathfrak{L}_2$ leakage does not provide the *exact* location of a word pair in a given document.

7.1.1 Leakage Based Attacks

The $\mathfrak{L}_2$ leakage provides an adversary with the ability to glean several pieces of additional information that is not explicitly requested by the party issuing the query. Similar to traditional SSE systems, the adversary (storage provider) learns the most frequent queries. In this case it means the adversary learns the most common bigrams which allows for a classical frequency attack.

Beyond the classical SSE attacks [3, 10, 24], our phrase search scheme allows an adversary to:

1. construct a token for any subphrase, of length greater than two, using the token corresponding to a previous phrase; and
2. use tokens for previous subphrase queries to construct a token for a new phrase query.

Observe that from the $\mathfrak{L}_2$ leakage, the adversary can recover all the subphrases since the adversary is provided with all of the bigram location information. The obfuscation of the location information prevents absolute location information but, determining if a word is present in a document can still be easily tested. The simple way of mitigating this attack would be to treat every possible phrase as a keyword thus collapsing phrase query based SSE to a keyword SSE. We add that this would require an exponential amount of storage and thus is infeasible.

From the $\mathfrak{L}_2$ leakage, the adversary can generate new tokens for phrases that were not previously queried. The primary danger here is a compound attack where the adversary uses a frequency analysis to guess at bigrams and use the obfuscated location information to construct possible results to new phrase queries.

As an example of our attack, suppose the adversary has observed two distinct phrase queries: "Trade 300 shares of Apple" and "Trade 500 shares of IBM." Observing these encrypted tokens an adversary can construct a new token that corresponds to the phrase "Trade 300 shares of IBM." As mentioned previously, this would require the adversary to use frequency analysis to guess at the subtokens of a query.

Our construction leaks a lot more than the best known SSE schemes, and in particular, allows the adversary to construct tokens for new queries. We do not claim this leakage is reasonable, we do however hope that the ideas in our work lead to progress in the design of SSE schemes. We also note that both reducing the leakage of our encrypted phrase search schemes and designing inference attacks against their leakage profile are both important research directions. The latter, in particular, would lead to a better understanding of leakage and motivate stronger constructions.

7.2 Adaptive chosen query attack security

Our security goal is to ensure that any probabilistic polynomial-time (PPT) adversary cannot compromise the system even if they are allowed to make adaptive queries. By adaptive queries we mean that the queries may depend directly or indirectly on the results of previous queries. To formalize our security notion we will utilize computational indistinguishability and assert that the system is secure if there does not exist a probabilistic polynomial-time adversary that can distinguish interacting with a genuine cloud from interacting with a simulator.

Formally, we define Adaptive Chosen Query Attack (CQA2) security using the real/ideal simulation methodology. In the real/ideal simulation methodology, two games are defined for an adversary A interacting with either a real system, called the *real game*, or a simulator S that has access to leakage functions $\mathfrak{L}_1$ and $\mathfrak{L}_2$. The interaction with the simulator is called the *ideal game*. We denote the real game for a protocol Π and adversary A as $\mathbf{Real}_A^{\Pi}(\lambda)$, where λ is the security parameter. We similarly denote the ideal game as $\mathbf{Ideal}_{A,S}^{\Pi}(\lambda)$.

In both games, the adversary A first selects a database DB and passes it to the game. The real game runs algorithms in the real system and the ideal game runs algorithms based on a simulator of the system. Our goal is to show that the adversary cannot distinguish from which games the search results are obtained. The games are defined as follows:

$\mathbf{Real}_A^\Pi(\lambda)$

1. Adversary A selects a database DB and passes it to the game.
2. The game runs Setup to determine the key K and runs $\mathsf{BuildIndex}(K, \mathsf{DB})$ to construct the index EDB.
3. The game gives EDB to A.
4. Query Phase (repeated a polynomial, in the security parameter, number of times)
   (a) Adversary A requests the token for query q.
   (b) The game returns to A the result of $\mathsf{Token}(K, q)$.
   (c) Adversary A may now execute a search.
5. Adversary A returns, as output, a bit.

$\mathbf{Ideal}_{A,S}^\Pi(\lambda)$

1. Adversary A selects a database DB and passes it to the game.
2. The game runs simulator S of our system on $\mathfrak{L}_1(\mathsf{DB})$ to obtain the encrypted index EDB.
3. The game gives EDB to A.
4. Query Phase (repeated a polynomial number of times)
   (a) Adversary A requests a token for query q.
   (b) The game gives $\mathfrak{L}_2(q)$ to S.
   (c) S returns a token t to the game which is forwarded to A.
   (d) Adversary A may now execute a search.
5. Adversary A returns, as output, a bit.

We say that Π is $(\mathfrak{L}_1, \mathfrak{L}_2)$-*secure against adaptive attacks* (CQA2-secure, in short) if for all adversaries A, there exists a simulator algorithm S such that

$$\left|\mathsf{Pr}\left[\mathbf{Real}_A^\Pi(\lambda) = 1\right] - \mathsf{Pr}\left[\mathbf{Ideal}_{A,S}^\Pi(\lambda) = 1\right]\right| \leq \mathsf{negl}(\lambda)$$

In other words, a system is secure if there does not exist a PPT distinguisher that can distinguish the distribution generated by $\mathbf{Real}_A^\Pi(\lambda)$ from $\mathbf{Ideal}_{A,S}^\Pi(\lambda)$.

Theorem 1. *Our SSE system is CQA2-secure if our pseudo-random functions and encryptions are generated with a random oracle.*

Proof. (sketch; a full proof is given in the appendix) In order to demonstrate the security of our system we must demonstrate how to construct a simulator S for the game $\mathbf{Ideal}_{A,S}^\Pi(\lambda)$. In step two of the **Ideal** game, we are given with $\mathfrak{L}_1(\mathsf{DB}) = N$ which is the number of entries in DB. We begin by constructing a list of N random values of the form $(\kappa, (r, v))$. The simulator then orders the key-value pairs in increasing order by the key. Finally, the simulator inserts all the key-value pairs into a dictionary EDB, which is returned to the adversary A.

When A issues a query in the game, $\mathfrak{L}_2(q)$ is provided to the simulator S. The simulator S then uses the results from $\mathfrak{L}_2(q)$ to program a random oracle. There are two cases for each result in the $\mathfrak{L}_2(q)$: (1) the subquery is new and the simulator has not committed to an entry in EDB; (2) a piece of the query was repeated and thus the simulator S has committed to a location in EDB. We have the following cases for each of the $|q| - 1$ subqueries:

Case 1: The subquery is new. Select two random keys K_1 and K_2 uniformly at random from $\{0,1\}^\lambda$. Thus emulating the portion of Token associated with the subquery. We then add the pair (K_1, K_2) to a dictionary indexed by a subquery identifier. For each result of the subquery we select a new location in $(\kappa, (r, v))$ in EDB. The random oracle is programmed as follows:

– $H[K_1 \parallel i] = \kappa$, where i is the ith response with the same document identifier.
– $H[K_2 \parallel r] = v \oplus (\mathsf{id} \parallel \text{curr} \parallel \text{next})$, where $(\mathsf{id} \parallel \text{curr} \parallel \text{next})$ is part of the current result.

Add the pair (K_1, K_2) to the token.

Case 2: The subquery is being repeated. In this case the subquery identifier is used to look up (K_1, K_2) in the subquery dictionary. This simulates Token on the subquery. The resulting key pair is then added to the token.

Using a hybrid argument, via a sequence of games, we can show that the game $\mathbf{Ideal}_{A,S}^\Pi$ is computationally indistinguishable from $\mathbf{Real}_A^\Pi$. We will give the full hybrid argument in appendix A. We conclude that our system is CQA2-secure. □

8 Implementation and Asymptotic Bounds

We implemented the system in ≈ 5000 lines of C using OpenSSL's libcrypto and GNU's implementation of the dbm key-value store. We tested our system on plaintext books, written in English, from the Project Gutenberg collection. For our purposes we were seeking exact phrase matches so no stemming or stop-word preprocessing was performed on the collection. A machine with 16 GB of RAM, 1 TB of disk space, and an AMD FX 8350 64-bit processor with 8 cores (supporting AES-NI) was used to collect the results.

Books	Key-Value Pairs	Average Time	Standard Deviation
10	492938	10.348	0.5561334772
20	1145852	24.875	0.6637310366
30	1795315	39.823	0.8878819992
40	2485966	55.132	1.1966323115
50	3187650	74.718	0.7698744486
60	3717873	86.406	5.3062315567
70	4298303	98.327	2.5036153858
80	4706366	109.545	2.3839381517
90	5111302	118.437	3.4830511847
100	5380289	127.037	3.2911330369
110	6007276	145.482	3.1928663542
120	6709367	160.881	4.7882018198
130	7175834	173.864	4.7569439303
140	7854670	189.891	5.1834854211
150	8393357	201.347	4.8515245027

Table 1. The BuildIndex average run time (in seconds) by number of key-value pairs (size of DB). These numbers only take into account the time it takes to produce EDB. Each test was repeated 100 times.

Books	Key-Value Pairs	Encrypted DB Size
10	492938	217
20	1145852	497
30	1795315	797
40	2485966	1087
50	3187650	1448
60	3717873	1643
70	4298303	1871
80	4706366	2040
90	5111302	2253
100	5380289	2424
110	6007276	2760
120	6709367	3011
130	7175834	3184
140	7854670	3449
150	8393357	3660

Table 2. The BuildIndex encrypted database size (in MB) by number of key-value pairs (size of DB).

The time it takes for BuildIndex to construct EDB for a given DB was measured. The results are in Table 1 and Figure 3. As can be seen, the behavior is linear in the size of DB as the theoretical bound states. The rising standard deviations for 60 books and up appear to be influenced greatly by one or two outliers occurring sporadically through the run. The size of the databases on disk is captured in Table 2.

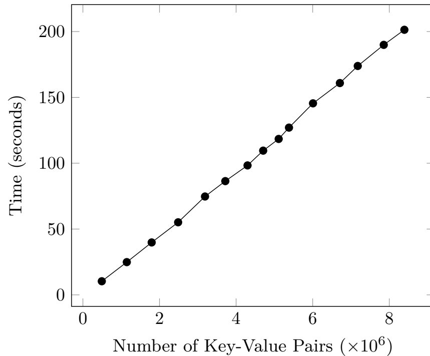


Fig. 3. The performance of BuildIndex on a varying number of key-value pairs. The graph plots only the average time it takes to produce EDB.

To test the query performance, a phrase common to all documents in the collection was chosen. In this way the work performed by the query operation is maximized. A single bigram, which is equivalent to a keyword search in our system, was also chosen to maximize the number of lookups that would have to be performed by the query operation. The results are summarized in Figure 4. The sharp jump in the time from the length four version of the phrase and the length five version of the phrase is due to the addition of a very common word pair, "for the."

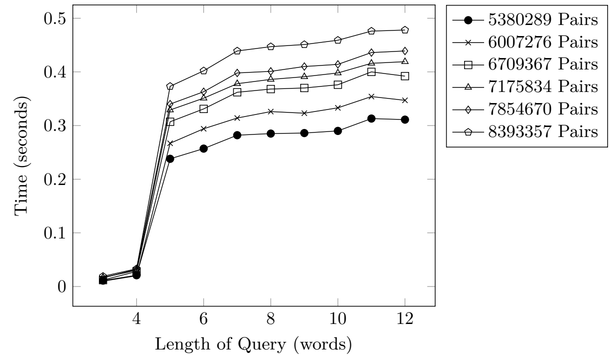


Fig. 4. The performance of Search on a varying number of key-value pairs and query sizes. Each query is found in every document in the collection.

The asymptotic bounds are summarized in table 3. We denote the length of a phrase query q as $|q|$ and the total number of bigrams inspected by evaluating query q as $m = \sum_{w_1 w_2 \in q} |\mathsf{DB}(w_1 w_2)|$.

9 Extensions and Modifications

Our main preprocessing step can be applied to other classic keyword based systems. We chose Π_{Bas} for sim-

Operation	Asymptotic Bound
Setup	$\Theta(1)$
BuildIndex	$\Theta(\lvert \mathsf{DB} \rvert)$
Token	$\Theta(\lvert q \rvert)$
Search	$O(m)$

Table 3. The asymptotic complexity of the SSE operations for phrase queries.

plicity. Recall from our preprocessing step that we are able to generate a new DB of the form

$$\mathsf{DB} = \left(\mathsf{id}_i,\, WP_i,\, \left\{L_i'(w_1w_2) : w_1w_2 \in WP_i\right\}\right)_{i=1}^{d}.$$

This increases the leakage of the underlying system by introducing the leakage of relative position of bigrams as well as slightly modifying any Search in much the same way as described in Section 6.5.

With some additional work we can also enrich cutting edge dynamic SSE constructions.

9.1 Non-Index Based Forward Secure Dynamic SSE

We can adapt the dynamic SSE construction devised by Stefanov et al. [19, 20] to work with phrases queries. Their construction is non-indexed based. In particular, they used a binary tree where each level of the tree stores encrypted edits to the document collection. These edits are either additions or deletions and the encryptions are computed under a per-level key.

In addition to the non-indexed approach, Stefanov et al's construction provides the first forward-secure SSE system. By forward-secure we mean that past query tokens can not be used to determine if an item inserted *after* the query belongs to the result set for the query in question.

For our purposes we are only concerned with setup, insertions, token generation, and searching. For ease of exposition, we will treat DB' as a collection without location information.

9.1.1 Setup

To setup the system the client will select a master encryption key mk and instruct the cloud to allocate an m level complete binary tree. For each level of the tree, the client will create an encryption key k_ℓ which will be maintained with mk as private state information.

9.1.2 Element Insertion

To insert an element $(\mathsf{id}_i, w) \in \mathsf{DB}'$ into EDB', the insertion process requires the construction of an ordered triple (hkey, c_1, c_2). To compute these values we begin by looking for the first empty level, ℓ, of the binary tree. Select the corresponding level key k_ℓ for constructing an hkey as

$$\text{hkey} = H_{F(k_\ell, h(w))}\left(0 \parallel op \parallel cnt\right),$$

where h is a cryptographic hash function, H a keyed cryptographic hash function, op either add or $delete$, and cnt represents the number of occurrences of w.

The value of c_1 is

$$c_1 = \mathsf{id}_i \oplus H_{F(k_\ell, h(w))}\left(1 \parallel op \parallel cnt\right).$$

The value of c_2 is

$$c_2 = E_{esk}\left(w, \mathsf{id}_i, op, cnt\right),$$

where esk is a special collection-wide key.

The client proceeds to download all levels above the empty level. Next, the client decrypts all of the c_2 values and rebuilds every entry using the new level key k_ℓ. During the rebuild, all cnt values must be appropriately adjusted. Finally, the items from the downloaded levels are deleted from their original locations and the newly constructed entries are stored in level ℓ. See Figure 5 for an graphic depiction of the process.

9.1.3 Token Generation

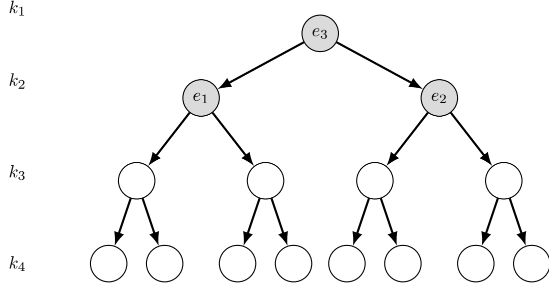
To generate a search token for keyword w, the client computes a search token for every non-empty level of the binary tree. In particular, if there are n non-empty levels the token returned will be:

$$T = \langle F(k_0, h(w)), F(k_1, h(w)), \ldots, F(k_{n-1}, h(w)) \rangle.$$

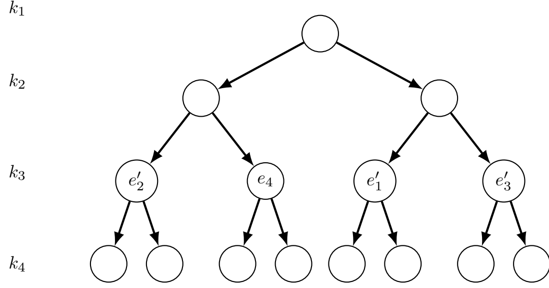
9.1.4 Search

Given a token T, the cloud searches for documents that match a keyword using breadth-first traversal starting with level 0. The processing of a level works by setting $ctr = 0$ and iteratively determining if $\text{hkey} = H_{F(k_\ell, h(w))}\left(0 \parallel add \parallel cnt\right)$ is contained in the level. If it is, the associated c_1 value is used to compute which identifier is to be added to the result set R_a. This is achieved by computing

$$\mathsf{id}_i = c_1 \oplus H_{F(k_\ell, h(w))}\left(1 \parallel add \parallel cnt\right).$$



(a) Locate the free level.



(b) Compress all levels into level three.

Fig. 5. Inserting edit e_4 requires locating an empty level in the tree (white nodes in Figure 5a). The gray nodes (Figure 5a) are filled with edits and must be downloaded in order to fill the level three in the last step of the insertion. The last level then becomes Figure 5b. We denote by e'_i the new encryption of node e_i. The encryptions are ordered by the hkey value left-to-right across the level

When an entry is not found the search proceeds to the next level of the tree.

Once all of the levels have been processed for *add* operations, the process is repeated for *delete* operations. Any identifiers recovered are added to the result set R_d.

The cloud will return to the client the set of identifiers $R_a \setminus R_d$.

9.2 Making a Dynamic SSE with Phrase Queries

We can modify the construction of Stefanov et al. to support phrase queries using our preprocessing step, which provide us with a collection containing obfuscated keyword pair locations. We demonstrate how to make the appropriate modifications to element insertion, token generation, and search.

9.2.1 Element Insertion

To insert an element $(id_i, w_1w_2 \parallel \text{curr} \parallel \text{next})$ from DB into EDB we again construct an ordered triple (hkey, c_1, c_2). The insertion process follows the same method as described in Section 9.1.2 except in the way we define hkey, c_1 and c_2. For phrase search we let

$$\text{hkey} = H_{F(k_\ell, h(w_1w_2))}\left(0 \parallel op \parallel cnt\right),$$

$$c_1 = (\mathsf{id}_i \parallel \text{curr} \parallel \text{next}) \oplus H_{F(k_\ell, h(w_1w_2))}\left(1 \parallel op \parallel cnt\right)$$

and let

$$c_2 = E_{esk}\left(w_1w_2, \mathsf{id}_i, \text{curr}, \text{next}, op, cnt\right),$$

where esk is a special collection-wide key. Any level rebuilding will follow the above definitions of hkey, c_1, and c_2.

9.2.2 Token Generation

To generate a search token for phrase $p = w_1w_2 \ldots w_m$, the client computes a search token for every non-empty level of the binary tree for every two subphrase in p. In particular, if there are n non-empty levels the token returned will be:

$$\begin{aligned} T = &\left\langle F\left(k_0, h\left(w_1w_2\right)\right), \ldots, F\left(k_{n-1}, h\left(w_1w_2\right)\right)\right\rangle \\ &\cup \left\langle F\left(k_0, h\left(w_2w_3\right)\right), \ldots, F\left(k_{n-1}, h\left(w_2w_3\right)\right)\right\rangle \\ &\vdots \\ &\cup \left\langle F\left(k_0, h\left(w_{m-1}w_m\right)\right), \ldots, F\left(k_{n-1}, h\left(w_{m-1}w_m\right)\right)\right\rangle. \end{aligned}$$

9.2.3 Search

Given a token $T = \langle T_1, T_2, \ldots T_m\rangle$ the cloud searches for documents that match a phrase sub-token using a breadth-first traversal starting with level 0. This means we look at all the matches for the first token in all the levels before proceeding to the next level.

The processing of a level works by setting a $ctr = 0$ and iteratively determining if

$$\text{hkey} = H_{T_i}\left(0 \parallel add \parallel cnt\right)$$

is contained in the level. If it is, the associated c_1 value is used to compute which identifier and location pair is to be added to the result set R_a. This is achieved by computing

$$(\mathsf{id}_i \parallel \text{curr} \parallel \text{next}) = c_1 \oplus H_{T_i}\left(1 \parallel add \parallel cnt\right).$$

When an entry is not found the search proceeds to the next level of the tree.

Once all of the levels have been processed for *add* operations, the process is repeated for *delete* operations. Any identifiers recovered are added to the result set R_d.

At the end of processing token T_i, we remove all document from R_a with document identifiers in set R_d result in set R'_a. In particular,

$$R'_a = \{(\mathsf{id}_i \parallel \mathrm{curr} \parallel \mathrm{next}) : (\mathsf{id}_i \parallel \mathrm{curr} \parallel \mathrm{next}) \in R_a \text{ and } \left(\mathsf{id}_i \parallel \mathrm{curr}' \parallel \mathrm{next}'\right) \notin R_d \text{ for all } \left(\mathrm{curr}', \mathrm{next}'\right)\right\}.$$

Next, given the current result set R we construct the set

$$R' = \left\{(\mathsf{id}_i \parallel \mathrm{curr} \parallel \mathrm{next}) : (\mathsf{id}_i \parallel \mathrm{curr} \parallel \mathrm{next}) \in R'_a, \left(\mathsf{id}_i \parallel \mathrm{curr}' \parallel \mathrm{next}'\right) \in R, \text{ and } \mathrm{curr} = \mathrm{next}'\right\}.$$

Lastly, we set $R = R'$ and begin processing the next token. Once all tokens have been processed, we return all the document identifiers found in set R to the client.

9.2.4 Security

We claim that our modifications do not change the security of the base system. Informally, we do not change the cryptography around insertions specifically; our only difference is the inclusion of location information. While this information changes what is leaked as a result of issuing a query, it does not change forward-security which is achieved through level download and re-encryption.

Our argument therefore hinges on CQA2 security for search. Observe that any simulator that simulates the original system can also simulate the augmented system as c_1 is a non-committing encryption. In particular, the exclusive-or used in the computation of c_1 will allow us to suitably program a simulator such that our desired location information can be available on search.

10 A Compiler Theorem

In this section we show that given *any* CQA2-secure SSE construction for keyword queries that leak $\mathsf{DB}(q)$, there exists a CQA2-secure SSE construction for phrase queries. Our proof is constructive and thus provides a method for compiling keyword constructions into phrase query constructions.

Before we formally state our theorem we set up a model for our SSE primitive based on our formal model defined in Section 3.2. From Section 3.1 we know we can think about DB being treated as solely binary data. Following this line of reasoning, the output of Search should be a subset of binary strings of polynomial length in λ. The key intuition is that we can encode information in the binary output of Search. More specifically, we notice that $\mathsf{id}_i \parallel \mathrm{curr} \parallel \mathrm{next}$ can be treated as a binary string. In addition any symmetric cryptography used to protected id_i can easily be extended to handle the longer string by replacing it with a suitably defined pseudo-random function.

Theorem 2. *Given any CQA2-secure SSE scheme* $S = (\mathsf{Setup}, \mathsf{BuildIndex}, \mathsf{Token}, \mathsf{Search})$ *that supports keyword queries and leaks the document identifiers as part of* $\mathcal{L}_2(q)$*, there exists a related CQA2-secure SSE scheme* $S' = (\mathsf{Setup}, \mathsf{BuildIndex}, \mathsf{Token}, \mathsf{Search})$ *that supports phrase queries for document collection* $\mathsf{DB} = (\mathsf{id}_i, WP_i, \{L'_i(w_1w_2) : w_1w_2 \in WP_i\})_{i=1}^{d}$.

Proof. (sketch; a full proof is given in the appendix) We begin by constructing S' in terms of S. Specifically,

- $S'.\mathsf{Setup}(\lambda)$ simply calls $S.\mathsf{Setup}(\lambda)$ directly and returns K to the caller.
- $S'.\mathsf{BuildIndex}(K, \mathsf{DB})$ first flattens collection DB into a form that $S.\mathsf{BuildIndex}$ can process. This is achieved by rewriting DB into DB′:

$$\left(\left((\mathsf{id}_i \parallel \mathrm{c} \parallel \mathrm{n},\ WP_i)_{(\mathrm{c},\mathrm{n}) \in L'_i(w_1w_2)}\right)_{w_1w_2 \in WP_i}\right)_{i=1}^{d}.$$

  This means that the location of every word pair in a document is going to result in a *new* document identifier. Each entry in WP_i is a binary string so there is no change their over the collection of binary strings W in a traditional SSE construction. We then call $S.\mathsf{buildIndex}\left(K, \mathsf{DB}'\right)$ and return EDB to the caller.
- $S'.\mathsf{Token}(K, w_1w_2 \cdots w_m)$ generates the necessary search tokens by calling $S.\mathsf{Token}$ for each of the two word subphrases in $w_1w_2 \cdots w_m$. The call returns with:

$$T = \bigcup_{i=1}^{m-1} \{S.\mathsf{Token}(K, w_iw_{i+1})\}.$$

- $S'.\mathsf{Search}(\mathsf{EDB}, T)$ proceeds as follows:
  1. Parse T as $T_1, T_2, \ldots, T_m$.
  2. Let $A = \emptyset$.
  3. For $i = 1$ to m:
     (a) Let $R = \emptyset$
     (b) Set $R = S.\mathsf{Search}(\mathsf{EDB}, T_i)$

Operation	Blowup
Setup	$\Theta(1)$
BuildIndex	$\Theta(\vert\text{DB}\vert)$
Token	$\Theta(\vert p\vert)$
Search	$\Theta(\vert p\vert)$

Table 4. The time blowup for base SSE keyword system for phrase p.

(c) If A is the empty set, then set $A = R$ otherwise compute

$$A' = \{(\mathsf{id}_i \parallel \text{curr} \parallel \text{next}) \in R : (\mathsf{id}_i \parallel \text{curr}' \parallel \text{next}') \in A \text{ and } \text{curr} = \text{next}'\};$$

set $A' = A$.

4. Return all of the document identifiers found in A to the caller.

We highlight the fact that if S is CQA2-secure than so is S'. To see this observe that if the underlying system is CQA2 secure then there must be a simulator that commits to document identifiers only after a query is made. Since, we have modified our document identifier to contain an encoding of location information, these new identifiers can also be simulated by the same simulator. We add that the query-time leakage will be adapted such that the new document identifiers are also maintained as part of the leakage. The token algorithm runs the underlying token algorithm multiple times which also can be handled by the simulator. Finally, the search algorithm allows the cloud to learn relative positions, but is not able to break the CQA2-security of S which is used to retrieve the information for S'. Combining the above, we can conclude that S' is also CQA2 secure. □

The performance of the construction that results for the compiler is competitive. We offer reasonable time blowups over the keyword SSE system fed into the compiler. These are summarized in Table 4

11 Conclusion

In this paper we demonstrated an asymptotically efficient, CQA2-secure, and parallelizable generic construction solving the encrypted phrase search problem. To our knowledge, this is the first such system. Additionally, we demonstrated how our preprocessing phase can be used to enhance an existing forward-secure dynamic SSE system so that it supports phrase queries. Of special interest is the fact that this system is forward secure.

Lastly, we provided a compiler theorem that allows us to convert any SSE construction for keyword queries into an SSE construction for phrase queries with only a small amount of additional query time leakage.

Acknowledgment

We would like to thank the anonymous reviewers for their valuable feedback. We would also like to thank Seny Kamara for providing editorial guidance as well as additional valuable feedback.

References

[1] Dan Boneh, Giovanni Di Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano. Public key encryption with keyword search. In Christian Cachin and Jan Camenisch, editors, *Advances in Cryptology - EUROCRYPT 2004*, volume 3027 of *Lecture Notes in Computer Science*, pages 506–522. Springer Berlin / Heidelberg, 2004.

[2] Christoph Bösch, Pieter Hartel, Willem Jonker, and Andreas Peter. A survey of provably secure searchable encryption. *ACM Comput. Surv.*, 47(2):18:1–18:51, August 2014.

[3] 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*, pages 668–679. ACM, 2015.

[4] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Dynamic searchable encryption in very large databases: Data structures and implementation. In *Proceedings of NDSS 2014*, 2014.

[5] David Cash, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Cătălin Roşu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Advances in Cryptology–CRYPTO 2013*, pages 353–373. Springer, 2013.

[6] Qi Chai and Guang Gong. Verifiable symmetric searchable encryption for semi-honest-but-curious cloud servers. In *IEEE International Conference on Communications, ICC'12*, June 2012.

[7] Yan-Cheng Chang and Michael Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *Proceedings of the Third international conference on Applied Cryptography and Network Security*, ACNS'05, pages 442–455, Berlin, Heidelberg, 2005. Springer-Verlag.

[8] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *Proceedings of the 13th ACM*

conference on Computer and communications security, CCS '06, pages 79–88, New York, NY, USA, 2006. ACM.
[9] Eu-Jin Goh. Secure indexes. *IACR Cryptology ePrint Archive*, 2003:216, 2003.
[10] MS Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Network and Distributed System Security Symposium (NDSS'12)*, 2012.
[11] Seny Kamara and Charalampos Papamanthou. Parallel and dynamic searchable symmetric encryption. In *Financial Cryptography and Data Security*, pages 258–274. Springer Berlin Heidelberg, 2013.
[12] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic searchable symmetric encryption. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 965–976, New York, NY, USA, 2012. ACM.
[13] Zachary A. Kissel and Jie Wang. Verifiable phrase search over encrypted data secure against a semi-honest -but-curious adversary. In *IEEE International Conference on Distributed Computing Systems, CCRM Workshop*, July 2013.
[14] Jin Li, Qian Wang, Cong Wang, Ning Cao, Kui Ren, and Wenjing Lou. Fuzzy keyword search over encrypted data in cloud computing. In *Proceedings of the 29th conference on Information communications*, INFOCOM'10, pages 441–445, Piscataway, NJ, USA, 2010. IEEE Press.
[15] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology — CRYPTO '87: Proceedings*, pages 369–378. Springer Berlin Heidelberg, Berlin, Heidelberg, 1988.
[16] Daniele Micciancio. Oblivious data structures: applications to cryptography. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 456–464. ACM, 1997.
[17] Moni Naor and Vanessa Teague. Anti-presistence: history independent data structures. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 492–501. ACM, 2001.
[18] Dawn Xiaoding Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, SP '00, pages 44–55, Washington, DC, USA, 2000. IEEE Computer Society.
[19] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. Practical dynamic searchable encryption with small leakage. *IACR Cryptology ePrint Archive*, 2013:832, 2013.
[20] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. Practical dynamic searchable encryption with small leakage. In *NDSS*, volume 14, pages 23–26, 2014.
[21] Yinqi Tang, Dawu Gu, Ning Ding, and Haining Lu. Phrase search over encrypted data with symmetric encryption scheme. *2012 32nd International Conference on Distributed Computing Systems Workshops*, 0:471–480, 2012.
[22] Jianfeng Wang, Xiaofeng Chen, Hua Ma, Qiang Tang, Jin Li, and Hui Zhu. A verifiable fuzzy keyword search scheme over encrypted data. *Journal of Internet Services and Information Security (JISIS)*, 2(1/2):49–58, 2 2012.
[23] Hugh E Williams, Justin Zobel, and Phil Anderson. What's next? index structures for efficient phrase querying. In *Proceedings of the Australasian Database Conference*, pages 141–152. Australian Computer Society, 1999.
[24] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. *IACR Cryptology ePrint Archive*, 2016:172, 2016.

A Full Proof of Theorem 1

We proceed to give the full proof of Theorem 1. To do this we employ a hybrid argument to show that, given any probabilistic polynomial-time adversary A and simulator S, we have

$$\left|\Pr\left[\mathbf{Real}_A^{\Pi}(\lambda)=1\right]-\Pr\left[\mathbf{Ideal}_{A,S}^{\Pi}(\lambda)=1\right]\right|\leq \mathsf{negl}(\lambda).$$

In a hybrid argument one constructs a sequence of $G_0,\ldots,G_n$ games, where n is bounded by some polynomial in λ. We start with G_0 being the game $\mathbf{Real}_A^{\Pi}(\lambda)$ and G_n being the game $\mathbf{Ideal}_{A,S}^{\Pi}(\lambda)$. For all games G_i $(i>0)$, we will define game G_i in terms of game G_{i-1}. In particular, G_i is formed by replacing a cryptographic operation in game G_{i-1} with its idealized form. We will argue that the distribution generated by game G_i is computationally indistinguishable from the distribution generated by game G_{i-1}. If we can find a sequence of such games, through transitivity of computational indistinguishability, we can conclude that the distributions of game G_0 and game G_n are computationally indistinguishable. In what follows, we will denote computationally indistinguishability of the distributions for game G_i and G_j using the notation $G_i\approx_c G_j$.

In all games below, except G_0, we assume that the adversary A and simulator S will have access to a table H. All queries from the adversary A will run on the algorithms from the SSE model with the changes noted in the game.

Let game G_0 be $\mathbf{Real}_A^{\Pi}(\lambda)$ as defined in Section 7.2.

Let game G_1 be defined in terms of G_0 with a slight modification of the $\mathsf{BuildIndex}$ and Token algorithm. Specifically, G_1 will use a simulator that replaces the first generation of K_1 in K_2 in the $\mathsf{BuildIndex}$ algorithm with the selection of random binary strings from $\{0,1\}^{\lambda}$. This information is recorded in the table H with the key $(K \parallel 1 \parallel w_1w_2)$ for K_1 and $(K \parallel 2 \parallel w_1w_2)$ for K_2. Any time that K_1 or K_2 need to be recomputed, the simulator will access the appropriate location in the table. Any time that the $\mathsf{BuildIndex}$ algorithm in G_0

tries to compute K_1 or K_2, the simulator will respond as described above. By the definition of pseudo-random functions we know that the behavior of F is computationally indistinguishable from an oracle that generates random strings of the same length. Because this is the only change between games G_0 and G_1 we can conclude that $G_0 \approx_c G_1$.

Let game G_2 be defined in terms of G_1 except the simulator will replace the generation of the value $\ell = F(K_1, c)$ in $\mathsf{BuildIndex}$ with a random string from $\{0,1\}^\lambda$ and store that value with key $K_1 \parallel c$ in the table H. Any time the value of ℓ needs to be computed it can be looked up in table H. Any time the simulator in G_1 tries to compute ℓ, the simulator for G_2 will respond as described above. By the definition of pseudo-random functions we know that the behavior of F is computationally indistinguishable from an oracle that generates random strings of the same length. Because, this is the only change between games G_1 and G_2 we can conclude that $G_1 \approx_c G_2$.

Let game G_3 be defined in terms of G_2 except the simulator will replace $E_{K_2}(\mathsf{id}_i \parallel \text{cur} \parallel \text{next})$ with a binary string of appropriate length. Because we must preserve decryption in Search, we will add two entries to the table H. For encryption we will select a binary string, $r \parallel v$ at random from $\{0,1\}^{2\lambda}$ and assign it to location $\mathsf{id}_i \parallel \text{cur} \parallel \text{next}$. For decryption, we will add to location $K_2 \parallel r$ the value $v \oplus (\mathsf{id}_i \parallel \text{cur} \parallel \text{next})$. Any time the simulator in game G_2 uses encryption or decryption operations, the simulator for G_3 will respond as described above. By the definition of pseudo-random functions we know that the behavior of F is computationally indistinguishable from an oracle that generates random strings of the same length. Because, this is the only change between games G_2 and G_3 we can conclude that $G_2 \approx_c G_3$.

Notice that the EDB produced by $\mathsf{BuildIndex}$ in game G_3 is exactly like what is produced by the game $\mathbf{Ideal}^{\Pi}_{A,S}(\lambda)$, so is the programming of the random oracle (table) H. Thus, the distribution generated is exactly the same as that of $\mathbf{Ideal}^{\Pi}_{A,S}(\lambda)$. We can conclude that G_3 is $\mathbf{Ideal}^{\Pi}_{A,S}(\lambda)$. Since the number of games is polynomial in λ, we can conclude that $\mathbf{Real}^{\Pi}_{A}(\lambda) \approx_c \mathbf{Ideal}^{\Pi}_{A,S}(\lambda)$.

B Full Proof of Theorem 2

We proceed to give a full proof of Theorem 2. We denote by $\mathcal{S}^k$ the simulator for the keyword SSE system used by the compiler and the simulator we construct for the phrase SSE system by $\mathcal{S}^p$. The leakage for our keyword based SSE system is defined in terms of two stateful leakage functions, $\mathfrak{L}^k_1$ and $\mathfrak{L}^k_2$. Our leakage for the system produced by the compiler is denoted as $\mathfrak{L}_1$ and $\mathfrak{L}_2$ which is defined in Section 7.1. Formally, we will define the leakage of a keyword SSE system by:

$$\mathfrak{L}^k_1(\mathsf{DB}) = \sum_{w \in W} |\mathsf{DB}(w)|,$$

where W is the set of all words in the collection, and

$$\mathfrak{L}^k_2(w) = (\mathsf{DB}(w), \mathsf{SP}(w)),$$

where the search pattern behaves as defined in Section 7.1. In the above $\mathsf{DB}(w)$ is just the set of identifiers for documents that contain w.

The simulator $\mathcal{S}^p$ is constructed to respond to the ideal game. In step two $\mathcal{S}^p$ is given $\mathfrak{L}_1(\mathsf{DB})$ which is equivalent to $\mathfrak{L}^k_1(\mathsf{DB})$, provided the document identifiers are allowed to be long binary strings. Thus $\mathfrak{L}_1(\mathsf{DB})$ is provided, unmodified, to $\mathcal{S}^k$ by $\mathcal{S}^p$. The resulting EDB generated by $\mathcal{S}^k$ is passed back to $\mathcal{S}^p$, and finally by $\mathcal{S}^p$ to the game.

During the query phase $\mathcal{S}^p$ receives $\mathfrak{L}_2(q)$ where the query consists of $m = |q|$ subqueries. For a given query q, $\mathfrak{L}_2(q) = \left(\bigcup_{i=1}^m \mathsf{DB}'(q_i), \bigcup_{i=1}^m \mathsf{SP}(q_i)\right)$. From this leakage $\mathcal{S}^p$ can construct the $\mathfrak{L}^k_2$ that needs to be given to $\mathcal{S}^k$. For each subquery q_i:

1. Construct $\mathsf{DB}(q_i)$ by collapsing all of the location information and document identifier information into one long binary string. Formally,

$$\mathsf{DB}(q_i) = \left\{(\mathsf{id} \parallel c \parallel n) \mid (\mathsf{id}, (c, n)) \in \mathsf{DB}'(q_i)\right\}.$$

2. Pass $\mathfrak{L}^k_2(q_i) = (\mathsf{DB}(q_i), \mathsf{SP}(q_i))$ to $\mathcal{S}^k$. The resulting token is appended to $\mathcal{S}^p$'s token.

After all subqueries have been processed, $\mathcal{S}^p$'s token is given back to the game.

Observe that $\mathcal{S}^p$'s operation depends entirely on $\mathcal{S}^k$ which is a simulator for a CQA2 secure keyword SSE system. In fact, the only work that $\mathcal{S}^p$ does is apply a small transformation to the leakage. Therefore if the underlying keyword SSE system is CQA2 secure then, we must conclude that our phrase SSE system is CQA2 secure.