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
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 straightforward 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 that insertion of new data into the collection does not allow an attacker to infer if the new data would match.

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

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

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

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

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 $w_1w_2$ 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

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

Formally we have:

- **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 $\lambda$, a security parameter, as input and returns a secret key $K$ such that the length of $K$ is polynomially bounded in $\lambda$.
- **BuildIndex ($K, DB$).** 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 $\lambda$, and returns an encrypted index $EDB$ such that the length of $EDB$ is polynomially bounded in $\lambda$.
- **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 $T_w$.
- **Search ($EDB, T_w$).** 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 $DB(w)$, the set of identifiers of documents containing $w$.

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

We say that a SSE scheme is correct if executing $\text{Search}$ with a token $T_w$ for word $w \in W$, returns $DB(w)$ 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 \( D \) 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 \( \bot \) 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 \( \Pi_{\text{Bas}} \) in [4], the system of Cash et al. begins by building a list of key-value pairs for every \( w \in W \):

\[
(F(K_1, c), E_{K_2}(\text{id})),
\]

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

\[
K_1 = F(K_1, 1 \parallel w),
K_2 = F(K_2, 2 \parallel w),
\]

\( K \) is a secret key generated by Setup \((1^\lambda) \), 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: \((K_1 = F(K_1, 1 \parallel w), K_2 = F(K_2, 2 \parallel w))\). The cloud uses these tokens to execute Search queries on the dictionary. The resulting document identifiers \( \text{DB}(w) \) 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 \( \text{EDB} \).
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 \( \mathcal{L}_1 \) and \( \mathcal{L}_2 \), where \( \mathcal{L}_1 \) captures what is leaked about \( \text{DB} \), and \( \mathcal{L}_2 \) captures what is leaked as a result of issuing queries. By assumption, both \( \mathcal{L}_1 \) and \( \mathcal{L}_2 \) implicitly share state.

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

5 A Trivial Phrase System

Using \( \Pi_{\text{Bas}} \), with little modification, we can obtain a generic SSE system that supports phrase search. The idea is to tag every entry in \( \text{EDB} \) 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

\[
(F(K_1, c), E_{K_3}(\text{id}, \text{curr}, \text{next})),
\]

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

\[
L'_i(w) = \{(F(K_3, x), F(K_3, x + 1)) : x \in L_i(w)\},
\]

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 \( \Pi_{\text{Bas}} \). 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 $DB(w_1)$ 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,
   \[ S' = \{ (id, curr, next) \in DB(w_i) : \exists (id, curr', next') \in S \text{ where } curr = next' \} \]  
   (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 $\sum_{w \in W} |DB(w)|$. 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 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 $\Pi_{Bas}$ 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].

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 $(w_1, w_2)$, 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, (9,1, [4])”, 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: \((w_{1}w_{2},(\text{id},l))\), where \( w_{1} \) appears in document \( \text{id} \) at location \( l \) (i.e., \( l \in L_{i}(w_{1}w_{2}) \)). 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 \((w_{1}\perp,(\text{id},l))\), where \( w_{1} \) appears in \( \text{id} \), \( \perp \) 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 \( w_{j}w_{j+1} \) in the document with document identifier \( \text{id}_{i} \). Let \( WP \) be the set of all unique bigrams in the entire collection.

For every document identifier \( \text{id}_{i} (1 \leq i \leq d) \), the client constructs a list of key-value pairs

\[
D_{\text{id}_{i}} = \{(w_{1}w_{2},(\text{id}_{i},l)) : w_{1}w_{2} \in WP_{i} \text{ and } l \in L_{i}(w_{1}w_{2})\}.
\]

The key-value pairs need to undergo obfuscation to reduce potential query time (\( \mathcal{L}_{2} \)) 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 \( D_{\text{id}_{i}} \), replaces each location \( l \) with \((\text{curr},\text{next})\), where \( \text{curr} = F(K_{3},l) \text{ and next} = F(K_{3},l+1) \).

To build the database \( DB \), the client inserts every key-value pair into the collection. To do this, construct, for each \( w_{1}w_{2} \in WP_{i} \), the following set:

\[
L'_{i}(w_{1}w_{2}) = \{(\text{curr},\text{next}) : (w_{1}w_{2},(\text{id}_{i},l)) \in D_{\text{id}_{i}}\},
\]

where \( l \in L_{i}(w_{1}w_{2}) \). Thus our collection becomes

\[
\text{DB} = \{(\text{id}_{i}, WP_{i}, L'_{i}(w_{1}w_{2}) : w_{1}w_{2} \in WP_{1})\}_{i=1}^{d}.
\]
1: function Setup(1\(^\lambda\))
2: Sample \(K\) uniformly from \(\{0,1\}\)^\(\lambda\)
3: return \(K\)
4: end function

1: function BuildIndex(K, DB)
2: Allocate empty list \(L\)
3: for each two word phrase \((w_1 \parallel w_2) \in WP\) do
4: \(K_1 \leftarrow F(K, 1 \parallel (w_1 \parallel w_2))\)
5: \(K_2 \leftarrow F(K, 2 \parallel (w_1 \parallel w_2))\)
6: \(c \leftarrow 0\)
7: for each \((id, cur, next) \in DB(w_1 \parallel w_2)\) do
8: \(\ell \leftarrow F(K_1, c)\)
9: \(d \leftarrow E_{K_2}(id \parallel cur \parallel next)\)
10: \(c \leftarrow c + 1\)
11: \(\triangleright Insert in L lexicographically by \ell\)
12: INSERT(L, (\(\ell, d\)))
13: end for
14: end for
15: allocate dictionary \(EDB\)
16: for each \((\ell, d) \in L\) do
17: INSERT(EDB, \(\ell, d\))
18: end for
19: return EDB
20: end function

1: function Token(K, p)
2: Parse \(p\) as \(w_1 \parallel w_2 \parallel \cdots \parallel w_p\)
3: \(T_p \leftarrow \emptyset\)
4: for \(i \leftarrow 1\) to \(|p| - 1\) do
5: Append \((F(K, 1 \parallel (w_i \parallel w_{i+1})), F(K, 2 \parallel (w_i \parallel w_{i+1})))\) to sequence \(T_p\)
6: end for
7: return \(T_p\)
8: end function

1: function Search(EDB, \(T_p\))
2: Allocate empty list \(list_1\)
3: \(first \leftarrow True\)
4: \(n \leftarrow 0\)
5: for each pair \((K_1, K_2) \in T_p\) do
6: \(n \leftarrow n + 1\)
7: \(c \leftarrow 0\)
8: Allocate an empty list \(list_2\)
9: repeat
10: \(d \leftarrow \) Lookup(EDB, \(F(K_1, c)\))
11: if \(d \neq \bot\) then
12: \((id, cur, next) \leftarrow D_{K_2}(d)\)
13: if \(first = True\) then
14: INSERT(list_1, (id, cur, next))
15: else \(\triangleright Check for match in list_1\)
16: if MATCH(list_1, id, cur) = True then
17: INSERT(list_2, (id, cur, next))
18: end if
19: end if
20: end if
21: \(c \leftarrow c + 1\)
22: until Lookup returns \(\bot\)
23: if \(first = False\) then
24: \(list_1 \leftarrow list_2\)
25: end if
26: \(first = False\)
27: end for
28: Using \(list_1\), return the set of matching document identifiers, \(DB(p)\), to the client.
29: end function

Fig. 2. The \(\Pi^\text{Bas}\) (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 \(\lambda\), 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 \(w_1w_2 \in WP\), let \(K_1 = F(K, 1 \parallel w_1w_2)\), \(K_2 = F(K, 2 \parallel w_1w_2)\).

For every key \(w_1w_2\) found in DB, suppose that \(w_1w_2\) 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 \(E_{K_2}(id \parallel cur \parallel next)\), where \(id \parallel cur \parallel next\) comes from the \(c\)-th occurrence of \(w\) in DB. That is, the corresponding entry in \(EDB\) is the following key-value pair:

\[(F(K_1, c), E_{K_2}(id \parallel cur \parallel next))\].
6.4 Query

To run a query on a phrase $w_1w_2 \cdots w_k$, the client makes a call to Token to get a search token, for each consecutive pair of words:

$$w_1w_2, w_2w_3, \cdots, w_{k-1}w_k.$$ 

The resulting tokens are

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

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 $\text{list}_1$. On subsequent pairs from the tokens, the cloud creates a new list $\text{list}_2$, and potentially adds an entry to the list $\text{list}_2$ after each Lookup operation. In order for an entry to be added to $\text{list}_2$, it must be the case that the value returned from the Lookup operation matches an entry in $\text{list}_1$. 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, $\text{list}_2$ becomes the new $\text{list}_1$. After the entire token has been processed the document identifiers from $\text{list}_1$ 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 $\Pi_{Bas}$ 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

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

and define our cryptographic operations, either all or in part, using this oracle. In particular, we define a pseudorandom 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 $\mathcal{L}_1$ and $\mathcal{L}_2$ that implicitly share state information. These functions describe what information is leaked by our system. The function $\mathcal{L}_1$ 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,

$$\mathcal{L}_1(DB) = \sum_{w_1w_2 \in WP} |DB(w_1w_2)|.$$ 

The collection DB is saved as state information.

The $\mathcal{L}_2$ 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 $\mathcal{L}_2$ function is invoked on a query $q$, denoted by $\mathcal{L}_2(q)$, 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 $q = w_1w_2 \cdots w_{|q|}$, the $\mathcal{L}_2$ leakage is formally defined by

$$\mathcal{L}_2(q) = (DB_q, SP_q),$$

$$DB_q = \{DB'(w_iw_{i+1}) : 1 \leq i < |q|\},$$

$$SP_q = \{SP(w_iw_{i+1}) : 1 \leq i < |q|\},$$

where $DB'(w_iw_{i+1})$ is defined as

$$DB'(w_jw_{j+1}) = \{(id^1_i, u, v) : w_jw_{j+1} \in WP_i \text{ and } (u, v) \in L_i^{id^1} (w_jw_{j+1})\}.$$
and $SP = \{j : (j, w_i w_{i+1}) \in Q\}$. 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 $\Sigma_2$ results in the set of tuples

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

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 $\Sigma_2$ leakage does not provide the exact location of a word pair in a given document.

### 7.1.1 Leakage Based Attacks

The $\Sigma_2$ 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 $\Sigma_2$ 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 $\Sigma_2$ 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 $\Sigma_1$ and $\Sigma_2$. The interaction with the simulator is called the ideal game. We denote the real game for a protocol $\Pi$ and adversary $A$ as $\text{Real}^\Pi_A(\lambda)$, where $\lambda$ is the security parameter. We similarly denote the ideal game as $\text{Ideal}^\Pi_A(S, \lambda)$.

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:

[Note: The rest of the text is not transcribed due to the length and complexity of the content.]
Real\(_A^I\) (A)
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 BuildIndex\((K, DB)\) to construct the index EDB.
3. The game gives EDB to A.
4. Query Phase (repeated a polynomial number of times)
   (a) Adversary A requests the token for query \(q\).
   (b) The game gives \( \mathcal{L}_2(q) \) 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.

Ideal\(_{A,S}^I\) (A)
1. Adversary A selects a database DB and passes it to the game.
2. The game runs simulator S of our system on \( \mathcal{L}_1(DB) \) 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 \( \mathcal{L}_2(q) \) 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 \( \Pi \) is \((\mathcal{L}_1, \mathcal{L}_2)\)-secure against adaptive attacks (CQA2-secure, in short) if for all adversaries A, there exists a simulator algorithm S such that
\[
\Pr[\text{Real}_{A}^{I}(\lambda) = 1] - \Pr[\text{Ideal}_{A,S}^{I}(\lambda) = 1] \leq \text{negl}(\lambda)
\]

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

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 \( \text{Ideal}_{A,S}^{I}(\lambda) \). In step two of the \( \text{Ideal} \) game, we are given with \( \mathcal{L}_1(DB) = N \) 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, \( \mathcal{L}_2(q) \) is provided to the simulator S. The simulator S then uses the results from \( \mathcal{L}_2(q) \) to program a random oracle. There are two cases for each result in the \( \mathcal{L}_2(q) \): (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:
- \(H[K_1 || i] = \kappa\), where \(i\) is the \(i\)th response with the same document identifier.
- \(H[K_2 || r] = v \oplus (id || curr || next)\), where \((id || curr || next)\) 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 \( \text{Ideal}_{A,S}^{I} \) is computationally indistinguishable from \( \text{Real}_{A}^{I} \). 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 \(\approx 5000\) 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.
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.

<table>
<thead>
<tr>
<th>Books</th>
<th>Key-Value Pairs</th>
<th>Average Time</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>492938</td>
<td>10.348</td>
<td>0.5561334772</td>
</tr>
<tr>
<td>20</td>
<td>1145852</td>
<td>24.875</td>
<td>2.5036153858</td>
</tr>
<tr>
<td>30</td>
<td>1795315</td>
<td>39.823</td>
<td>0.8878744486</td>
</tr>
<tr>
<td>40</td>
<td>2485966</td>
<td>55.132</td>
<td>5.3062315567</td>
</tr>
<tr>
<td>50</td>
<td>3187650</td>
<td>74.718</td>
<td>2.5036153858</td>
</tr>
<tr>
<td>60</td>
<td>3717873</td>
<td>86.406</td>
<td>1.1966323211</td>
</tr>
<tr>
<td>70</td>
<td>4298303</td>
<td>98.327</td>
<td>0.6637310366</td>
</tr>
<tr>
<td>80</td>
<td>4706366</td>
<td>109.545</td>
<td>1.1966323211</td>
</tr>
<tr>
<td>90</td>
<td>5111302</td>
<td>118.437</td>
<td>0.8878744486</td>
</tr>
<tr>
<td>100</td>
<td>5380289</td>
<td>127.037</td>
<td>3.2911330369</td>
</tr>
<tr>
<td>110</td>
<td>6007276</td>
<td>145.482</td>
<td>3.1928663542</td>
</tr>
<tr>
<td>120</td>
<td>6709367</td>
<td>160.881</td>
<td>4.7882018198</td>
</tr>
<tr>
<td>130</td>
<td>7175834</td>
<td>173.864</td>
<td>4.7569439303</td>
</tr>
<tr>
<td>140</td>
<td>7854670</td>
<td>189.891</td>
<td>5.1834854211</td>
</tr>
<tr>
<td>150</td>
<td>8393357</td>
<td>201.347</td>
<td>4.8515245027</td>
</tr>
</tbody>
</table>

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.

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

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

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 $m = \sum_{w_1, w_2 \in q} |DB(w_1 w_2)|$.

9 Extensions and Modifications

Our main preprocessing step can be applied to other classic keyword based systems. We chose $\Pi_{Bas}$ for sim-
<table>
<thead>
<tr>
<th>Operation</th>
<th>Asymptotic Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setup</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>BuildIndex</td>
<td>( \Theta(</td>
</tr>
<tr>
<td>Token</td>
<td>( \Theta(</td>
</tr>
<tr>
<td>Search</td>
<td>( O(m) )</td>
</tr>
</tbody>
</table>

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

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

\[
DB = (id_i, WP_i, \{ L'_i(w_1w_2) : w_1w_2 \in WP_i \})_{i=1}^d.
\]

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 \( k \ell \) which will be maintained with mk as private state information.

### 9.1.2 Element Insertion

To insert an element \((id_i, w) \in DB'\) 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, \( \ell \), of the binary tree. Select the corresponding level key \( k \ell \) for constructing an hkey as

\[
hkey = H_{F(k \ell, h(w))} (0 \parallel op \parallel cnt),
\]

where \( h \) is a cryptographic hash function, \( H \) a keyed cryptographic hash function, \( op \) either \text{add} or \text{delete}, and \( cnt \) represents the number of occurrences of \( w \).

The value of \( c_1 \) is

\[
c_1 = id_i \oplus H_{F(k \ell, h(w))} (1 \parallel op \parallel cnt).
\]

The value of \( c_2 \) is

\[
c_2 = E_{esk} (w, id_i, op, cnt),
\]

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 \( k \ell \). 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 \( \ell \). 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:

\[
T = (F(k_0, h(w)), F(k_1, h(w)), \ldots, F(k_{n-1}, h(w))).
\]

### 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 \( hkey = H_{F(k_\ell, h(w))} (0 \parallel add \parallel cnt) \) 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 \( R_\alpha \). This is achieved by computing

\[
id_i = c_1 \oplus H_{F(k_\ell, h(w))} (1 \parallel add \parallel cnt).
\]
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 provides 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 \((id_i, w_1w_2 || curr || next)\) 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

\[
hkey = H_{F(k_i, h(w_1w_2))} (0 || op || cnt),
\]

\[
c_1 = (id_i || curr || next) \oplus H_{F(k_i, h(w_1w_2))} (1 || op || cnt)
\]

and let

\[
c_2 = E_{esk} (w_1w_2, id_i, curr, next, op, cnt),
\]

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:

\[
T = \langle F(k_0, h(w_1w_2)), \ldots, F(k_{n-1}, h(w_1w_2)) \rangle
\]

\[
\cup \langle F(k_0, h(w_2w_3)), \ldots, F(k_{n-1}, h(w_2w_3)) \rangle
\]

\[
\cup \ldots
\]

\[
\cup \langle F(k_0, h(w_{m-1}w_m)), \ldots, F(k_{n-1}, h(w_{m-1}w_m)) \rangle.
\]

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

\[
hkey = H_{T_i} (0 || add || cnt)
\]

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

\[
(id_i || curr || next) = c_1 \oplus H_{T_i} (1 || add || cnt).
\]
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,

\[
R'_a = (\langle \text{id}_i \parallel \text{curr} \parallel \text{next} \rangle : (\langle \text{id}_i \parallel \text{curr} \parallel \text{next} \rangle \in R_a \text{ and } (\langle \text{id}_i \parallel \text{curr}' \parallel \text{next}' \rangle \notin R_d \text{ for all } (\text{curr}', \text{next}'))).
\]

Next, given the current result set \( R \) we construct the set

\[
R' = (\langle \text{id}_i \parallel \text{curr} \parallel \text{next} \rangle : (\langle \text{id}_i \parallel \text{curr} \parallel \text{next} \rangle \in R'_a, (\langle \text{id}_i \parallel \text{curr}' \parallel \text{next}' \rangle \in R_d \text{ and } \text{curr} = \text{next}'\})).
\]

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 insertsions 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 \( \text{DB}(q) \), 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 \( \text{DB} \) being treated as solely binary data. Following this line of reasoning, the output of \( \text{Search} \) should be a subset of binary strings of polynomial length in \( \lambda \). The key intuition is that we can encode information in the binary output of \( \text{Search} \). More specifically, we notice that \( \text{id}_i \parallel \text{curr} \parallel \text{next} \) can be treated as a binary string. In addition any symmetric cryptography used to protected \( \text{id}_i \) 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 \( S = (\text{Setup}, \text{BuildIndex}, \text{Token}, \text{Search}) \) that supports keyword queries and leaks the document identifiers as part of \( L_2(q) \), there exists a related CQA2-secure SSE scheme \( S' = (\text{Setup}, \text{BuildIndex}, \text{Token}, \text{Search}) \) that supports phrase queries for document collection \( \text{DB} = (\langle \text{id}_i, \text{WP}_i, \{L'_i(w_1 w_2) : w_1 w_2 \in \text{WP}_i\} \rangle)_{i=1}^\lambda \).

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

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

\[
\left( (\langle \text{id}_i \parallel \text{curr} \parallel \text{n} \parallel \text{WP}_i, \{L'_i(w_1 w_2) \in \text{WP}_i : w_1 w_2 \in \text{WP}_i\}) \right)_{i=1}^\lambda.
\]

This means that the location of every word pair in a document is going to result in a new document identifier. Each entry in \( \text{WP}_i \) is a binary string so there is no change their order the collection of binary strings \( W \) in a traditional SSE construction.

We then call \( S.\text{BuildIndex}(K, \text{DB}') \) and return \( \text{EDB} \) to the caller.

- \( S'.\text{Token}(K, w_1 w_2 \ldots w_m) \) generates the necessary search tokens by calling \( S.\text{Token} \) for each of the two word subphrases in \( w_1 w_2 \ldots w_m \). The call returns with:

\[
T = \bigcup_{i=1}^{m-1} \{S.\text{Token}(K, w_i w_{i+1})\}.
\]

- \( S'.\text{Search}(\text{EDB}, T) \) 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 \( R = S.\text{Search}(\text{EDB}, T_i) \)
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


---

<table>
<thead>
<tr>
<th>Operation</th>
<th>Blowup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setup</td>
<td>$\Theta (1)$</td>
</tr>
<tr>
<td>BuildIndex</td>
<td>$\Theta (\lvert DB \rvert)$</td>
</tr>
<tr>
<td>Token</td>
<td>$\Theta (\lvert p \rvert)$</td>
</tr>
<tr>
<td>Search</td>
<td>$\Theta (\lvert p \rvert)$</td>
</tr>
</tbody>
</table>

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

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

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 then 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
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

$$|Pr[Real_{A}^{\Pi}(\lambda) = 1] - Pr[Ideal_{A,S}^{\Pi}(\lambda) = 1]| \leq \text{negl}(\lambda).$$

In a hybrid argument one constructs a sequence of games $G_{0}, \ldots, G_{n}$, where $n$ is bounded by some polynomial in $\lambda$. We start with $G_{0}$ being the game $Real_{A}^{\Pi}(\lambda)$ and $G_{n}$ being the game $Ideal_{A,S}^{\Pi}(\lambda)$. 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 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 $Real_{A}^{\Pi}(\lambda)$ as defined in Section 7.2.

Let game $G_{1}$ be defined in terms of $G_{0}$ with a slight modification of the BuildIndex and Token algorithm. Specifically, $G_{1}$ will use a simulator that replaces the first generation of $K_{1}$ in $K_{2}$ in the BuildIndex algorithm with the selection of random binary strings from $\{0, 1\}^{\lambda}$. This information is recorded in the table $H$ with the key $(K \parallel 1 \parallel w_{1}w_{2})$ for $K_{1}$ and $(K \parallel 2 \parallel w_{1}w_{2})$ 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 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 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 $\ell$ needs to be computed it can be looked up in table $H$. Any time the simulator in $G_1$ tries to compute $\ell$, 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 $E_{K_2}(id_i \parallel cur \parallel next)$ 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 $id_i \parallel cur \parallel next$. For decryption, we will add to location $K_2 \parallel r$ the value $v \oplus (id_i \parallel cur \parallel next)$. 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 BuildIndex in game $G_3$ is exactly like what is produced by the game $\text{Ideal}_{A,S}^1(\lambda)$, so is the programming of the random oracle (table) $H$. Thus, the distribution generated is exactly the same as that of $\text{Ideal}_{A,S}^1(\lambda)$. We can conclude that $G_3$ is $\text{Ideal}_{A,S}^1(\lambda)$. Since the number of games is polynomial in $\lambda$, we can conclude that $\text{Real}_{A,S}^1(\lambda) \approx_c \text{Ideal}_{A,S}^1(\lambda)$.

## B Full Proof of Theorem 2

We proceed to give a full proof of Theorem 2. We denote by $S^k$ the simulator for the keyword SSE system used by the compiler and the simulator we construct for the phrase SSE system by $S^p$. The leakage for our keyword based SSE system is defined in terms of two stateful leakage functions, $L_1^k$ and $L_2^k$. Our leakage for the system produced by the compiler is denoted as $L_1$ and $L_2$ which is defined in Section 7.1. Formally, we will define the leakage of a keyword SSE system by:

$$L_1^k(D) = \sum_{w \in W} |DB(w)|,$$

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

$$L_2^k(w) = (DB(w), SP(w)),$$

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

The simulator $S^p$ is constructed to respond to the ideal game. In step two $S^p$ is given $L_1(D)$ which is equivalent to $L_1^k(DB)$, provided the document identifiers are allowed to be long binary strings. Thus $L_1(D)$ is provided, unmodified, to $S^k$ by $S^p$. The resulting EDB generated by $S^k$ is passed back to $S^p$, and finally by $S^p$ to the game.

During the query phase $S^p$ receives $L_2(q)$ where the query consists of $m = |q|$ subqueries. For a given query $q$, $L_2(q) = (\bigcup_{i=1}^{m} DB'(q_i), \bigcup_{i=1}^{m} SP(q_i))$. From this leakage $S^p$ can construct the $L_2^k$ that needs to be given to $S^k$. For each subquery $q_i$.

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

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

2. Pass $L_2^k(q_i) = (DB(q_i), SP(q_i))$ to $S^k$. The resulting token is appended to $S^p$'s token.

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

Observe that $S^p$'s operation depends entirely on $S^k$ which is a simulator for a CQA2 secure keyword SSE system. In fact, the only work that $S^p$ 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.