Secure Indexes*

Eu-Jin Goh
eujin@cs.stanford.edu

March 16, 2004

Abstract

A secure index is a data structure that allows a querier with a "trapdoor" for a word x to test in $O(1)$ time only if the index contains x; The index reveals no information about its contents without valid trapdoors, and trapdoors can only be generated with a secret key. Secure indexes are a natural extension of the problem of constructing data structures with privacy guarantees such as those provided by oblivious and history independent data structures. In this paper, we formally define a secure index and formulate a security model for indexes known as semantic security against adaptive chosen keyword attack (IND-CKA). We also develop an efficient IND-CKA secure index construction called Z-IDX using pseudo-random functions and Bloom filters, and show how to use Z-IDX to implement searches on encrypted data. This search scheme is the most efficient encrypted data search scheme currently known; It provides $O(1)$ search time per document, and handles compressed data, variable length words, and boolean and certain regular expression queries. The techniques developed in this paper can also be used to build encrypted searchable audit logs, private database query schemes, accumulated hashing schemes, and secure set membership tests.

1 Introduction

Keyword indexes let us search in constant time for documents containing specified keywords. Unfortunately, standard index constructions such as those using hash tables are unsuitable for indexing encrypted (and presumably sensitive) documents because they leak information about the document contents (and hence break semantic security). Informally, a secure index allows users with a "trapdoor" for a word x to test the index only for x; The index reveals no information about its contents without valid trapdoors, and trapdoors can only be generated with a secret key. Data structures with such privacy guarantees can be used to safely index the contents of semantically secure ciphertexts. We note that secure indexes do not hide information such as document size that can be obtained by simply examining the encrypted documents.

Secure indexes are a natural extension of the problem of constructing data structures with privacy guarantees such as those provided by oblivious [16] and history independent [19, 8] data structures. In oblivious (history independent) data structures, the shape (memory representation) of the data structure reveals no information about the sequence of operations applied to the data structure other than the final result. History independence is a necessary, but not sufficient, condition for a secure index; A history independent data structure guarantees nothing about the privacy of its contents, which is exactly the property required by secure indexes.

*A early version of this paper first appeared on the Cryptology ePrint Archive on October 7th 2003.

Our Contribution. The first contribution of this paper is in defining a secure index and formulating a security model for indexes known as semantic security against adaptive chosen keyword attack (IND-CKA). The IND-CKA model captures the intuitive notion that the contents of a document are not revealed from its index and the indexes of other documents apart from what an adversary already knows from previous query results or other channels. For example, consider a document D containing n words, of which an adversary already knows m words and wants information about the set W of $n - m$ unknown words. Even when the adversary $\mathcal{A}$ has full access to other index-document pairs, and can adaptively obtain trapdoors for any word except those in W, $\mathcal{A}$ still cannot deduce any information about any word in W from D's index.

For an index to be IND-CKA secure, two encrypted documents of equal size must have indexes that appear to contain the same number of keywords. As Chang and Mitzenmacher [9] observe, our model allows for indexes that still leak some information, namely the approximate total number of tokens (some tokens may not be keywords) in the document.[1] They propose a stronger model, which we call IND2-CKA, where the indexes for two documents with different numbers of keywords cannot be distinguished. On the other hand, the approximate total number of tokens in the document can be deduced from the (public) size of the encrypted document.

The second contribution is an efficient IND-CKA secure index construction called Z-IDX using pseudo-random functions and Bloom filters. Previous attempts at building secure indexes using hash tables [22] are incomplete because their indexes cannot securely handle updates. We also show how to use Z-IDX for the application of searching on encrypted data, and as a result, obtain the most efficient scheme currently known for this application. Our search scheme requires only $O(1)$ search time per document, and can (without any modification) handle variable length words, and boolean and certain regular expression queries. Our search scheme, being index-based, is indifferent to the compression and encryption algorithm used on the documents.

Extending the techniques in this paper, Chang and Mitzenmacher [9] later developed two secure index constructions using pre-built dictionaries. Their main contribution is an index blinding technique, enabling them to prove security in the stronger IND2-CKAmodel. The price they pay for extra security using this index blinding technique is threefold —

1. Computationally less efficient: For example, Z-IDX's index building algorithm for a document with n words requires $O(n)$ time, whereas their scheme requires $3n + 2^{d+1}$ time per document where 2^d is the size of the dictionary. Note that $2^d \gg n$ for the majority of documents.
2. Inability to handle arbitrary updates: Unlike Z-IDX their schemes use pre-built dictionaries where the dictionary size and contents are fixed when the dictionary and indexes are built for a set of documents. Since it is difficult to anticipate all possible words that need to be indexed, it is fair to say that their schemes cannot efficiently handle arbitrary updates. For the same reasons as the hash table index [22], it appears difficult to securely update the dictionary on a remote server.
3. Relatively large fixed size indexes: Regardless of the document size, their indexes must be 2^d bits long. For small documents, their index can be many times larger than the document, whereas Z-IDX indexes may be encoded so that they are only slightly larger than the document.

We note that this extra security (IND2-CKA) is not required for securing index contents, but if

[1] In an earlier version of their paper, Chang and Mitzenmacher suggest suspected flaws in our proof, it appears that their disagreement is not with the proof but with the assumptions behind the model. Nevertheless, we provide simple extensions to both our model to capture a similar level of security (IND2-CKA) as Chang and Mitzenmacher, and also to our index construction (described in Section 3.2) to ensure that it is secure in this stronger model.

required, a small modification to Z-IDX described in Section 3.2 achieves IND2-CKA security — our resulting scheme is simpler, computationally more efficient, and can also handle arbitrary updates; On the other hand, this increased security comes at the cost of larger indexes.

Applications. Our interest in secure indexes is primarily motivated by the application of searching on encrypted data [22, 7, 9], but secure indexes can also be used for accumulated hashing [5, 20] and testing set membership securely. In fact, the techniques in this paper have already been used to build encrypted and searchable audit logs [23], and similar techniques were recently used to build databases that allow for private queries using a semi-trusted third party [4]. We now describe a few applications in greater detail.

1. **Searching on Encrypted Data.** Increasingly, organizations and users are storing their data on servers outside their direct control. Backups and remote data storage are commonly outsourced to data warehousing companies. Even when data is stored locally, organizations hire system administrators (sysadmins) to run backup and storage servers. In either case, sysadmins typically have full administrator access to storage systems, and can read confidential data. Therefore, sensitive documents should be encrypted before being storing remotely.

   With files encrypted on a remote server, it is difficult to retrieve files based on their content. For example, consider a user Alice who stores her work documents on a untrusted file system. Suppose Alice wishes to retrieve all documents containing the word "aardvark". Since the document files are encrypted and the server cannot be trusted with the document keys or their contents, Alice has to download all the document files, decrypt them, and then search the decrypted documents on her local machine; This naive solution is inefficient. The ideal solution is to let the server search the encrypted documents and return only relevant ones, while ensuring that it learns nothing about the keyword or document contents. Section 4 discusses both the use of secure indexes and related work for this search problem.

   Note that secure indexes are useful for searching on encrypted data only in multi-user settings where the encrypted documents and their indexes on the remote server are frequently updated. If indexes are used by only one user or if indexes are never updated, the best solution is for each user to maintain a local index such as a hash table where each bucket contains pointers to documents with a particular search term. Since indexes are stored locally, no security is required. Such local indexes do not work well in multi-user settings because it is difficult to maintain the consistency of local indexes across all users when documents are updated.

2. **Encrypted and Searchable Audit Logs.** Waters et al. [23] build private audit logs that allow keyword searches only if the querier possesses trapdoors for those keywords. The symmetric key version of their scheme is based partly on techniques derived from this paper.

3. **Private Queries.** Using similar techniques, Bellovin and Cheswick [4] recently developed a scheme for performing private queries using a semi-trusted third party.

2 IND-CKA Secure Indexes

In this section, we define an IND-CKA secure index scheme.

Notation. Throughout the paper, we use $x, y, z \stackrel{R}{\leftarrow} S$ to denote random variables x, y, and z that are drawn uniformly at random from the set S. We write $x \stackrel{R}{\leftarrow} [1, N]$ to denote a random variable x drawn uniformly at random from the set of integers in $[1, N]$. For a randomized algorithm $\mathcal{A}$, we use $x \stackrel{R}{\leftarrow} \mathcal{A}()$ to denote the random variable x representing the output of the algorithm. The *symmetric set difference* of two sets A and B is defined as $A \bowtie B \stackrel{\text{def}}{=} (A - B) \cup (B - A)$. We write $|A|$ for a set A to denote the number of elements A. We use $\|$ to denote string concatenation.

Index Scheme. An index scheme consists of the following four algorithms —

`Keygen`(s): Given a security parameter s, outputs the master private key K_{priv}.
`Trapdoor`(K_{priv}, w): Given the master key K_{priv} and word w, outputs the trapdoor T_w for w.
`BuildIndex`(D, K_{priv}): Given a document D and the master key K_{priv}, outputs the index $\mathcal{I}_D$.
`SearchIndex`$(\mathrm{T}_w, \mathcal{I}_D)$: Given the trapdoor T_w for word w and the index $\mathcal{I}_D$ for document D, outputs 1 if $w \in D$ and 0 otherwise.

Semantic Security Against Adaptive Chosen Keyword Attack (IND-CKA). Intuitively, our security model aims to capture the notion that an adversary $\mathcal{A}$ cannot deduce a document's contents from its index, other than what it already knows from previous query results or from other channels. Roughly speaking, the game works as follows — Suppose the challenger $\mathcal{C}$ gives the adversary $\mathcal{A}$ two equal length documents V_0 and V_1, each containing some (possibly unequal) number of words, together with an index. Here, $\mathcal{A}$'s challenge is to determine which document is encoded in the index. If the problem of distinguishing between the index for V_0 and V_1 is hard, then deducing at least one of the words that V_0 and V_1 do not have in common from the index must also be hard. If $\mathcal{A}$ cannot determine which document is encoded in the index with probability non-negligibly different from 1/2, then the index reveals nothing about its contents. We use this formulation of index indistinguishability (IND) to prove the semantic security of indexes. We note that secure indexes do not hide information such as document size that can be obtained by examining the encrypted documents. More precisely, let (`Keygen`, `Trapdoor`, `BuildIndex`, `SearchIndex`) be an index scheme. We use the following game between a challenger $\mathcal{C}$ and an attacker $\mathcal{A}$ to define semantic security against an adaptive chosen keyword attack (IND-CKA) —

Setup. The challenger $\mathcal{C}$ creates a set S of q words and gives this to the adversary $\mathcal{A}$. $\mathcal{A}$ chooses a number of subsets from S.[2] This collection of subsets is called S^* and is returned to $\mathcal{C}$. Upon receiving S^*, $\mathcal{C}$ runs `Keygen` to generate the master key K_{priv}, and for each subset in S^*, $\mathcal{C}$ encodes its contents into an index using `BuildIndex`. Finally, $\mathcal{C}$ sends all indexes with their associated subsets to $\mathcal{A}$.

Queries. $\mathcal{A}$ is allowed to query $\mathcal{C}$ on a word x and receive the trapdoor T_x for x. With T_x, $\mathcal{A}$ can invoke `SearchIndex` on an index $\mathcal{I}$ to determine if $x \in \mathcal{I}$.

Challenge. After making some `Trapdoor` queries, $\mathcal{A}$ decides on a challenge by picking a non-empty subset $V_0 \in S^*$, and generating another non-empty subset V_1 from S such that $|V_0 - V_1| \neq 0, |V_1 - V_0| \neq 0$, and the total length of words in V_0 is equal to that in V_1.[3] Lastly, $\mathcal{A}$ must not have queried $\mathcal{C}$ for the trapdoor of any word in $V_0 \bowtie V_1$.

[2] S is the set of all the unique words in a collection of documents and each of the subsets represents a document.
[3] Note that it is possible for $|V_0| \neq |V_1|$ provided both V_0 and V_1 have equal total word length.

Next, $\mathcal{A}$ gives V_0 and V_1 to $\mathcal{C}$ who chooses $b \xleftarrow{R} \{0,1\}$, invokes $\texttt{BuildIndex}(V_b, K_{\text{priv}})$ to obtain the index $\mathcal{I}_{V_b}$ for V_b, and returns $\mathcal{I}_{V_b}$ to $\mathcal{A}$. The challenge for $\mathcal{A}$ is determine b. After the challenge is issued, $\mathcal{A}$ is not allowed to query $\mathcal{C}$ for the trapdoors of any word $x \in V_0 \bowtie V_1$.

Response. $\mathcal{A}$ eventually outputs a bit b', representing its guess for b. The advantage of $\mathcal{A}$ in winning this game is defined as $\texttt{Adv}_{\mathcal{A}} = |\Pr[b = b'] - 1/2|$, where the probability is over $\mathcal{A}$ and $\mathcal{C}$'s coin tosses.

We say that an adversary $\mathcal{A}$ (t, ϵ, q)-breaks an index if $\texttt{Adv}_{\mathcal{A}}$ is at least ϵ after $\mathcal{A}$ takes at most t time and makes q trapdoor queries to the challenger. We say that $\mathcal{I}$ is an (t, ϵ, q)-IND-CKA *secure index* if no adversary can (t, ϵ, q)-break it.

Stronger Security Model (IND2-CKA). We can strengthen the model to attain a similar level of security as Chang and Mitzenmacher [9] by slightly altering the criteria for the choice of the challenge subsets V_0, V_1 in the **Challenge** phase of the IND-CKA game. We call this stronger model IND2-CKA. In this game, the adversary $\mathcal{A}$ decides on a challenge by picking two non-empty subsets $V_0, V_1 \in S^*$ (possibly of unequal size and unequal total word length) such that $|V_0 \bowtie V_1| \neq 0$. The rest of the game is identical.

3 Constructing IND-CKA Secure Indexes

We first review Bloom filters, pseudo-random functions, and pseudo-random generators.

3.1 Background

Bloom Filters. A Bloom filter [6] represents a set of $S = \{s_1, \ldots, s_n\}$ of n elements and is represented by an array of m bits. All array bits are initially set to 0. The filter uses r independent hash functions $h_1, \ldots, h_r$, where $h_i : \{0,1\}^* \to [1, m]$ for $i \in [1, r]$. For each element $s \in S$, the array bits at positions $h_1(s), \ldots, h_r(s)$ are set to 1. A location can be set to 1 multiple times, but only the first is noted. To determine if an element a belongs to the set S, we check the bits at positions $h_1(a), \ldots, h_r(a)$. If all the checked bits are 1's, then a is considered a member of the set. There is, however, some probability of a false positive, in which a appears to be in S but actually is not. False positives occur because each location may have also been set by some element other than a. On the other hand, if any checked bits are 0, then a is definitely not a member of S. Section 5 discusses the false positive rate. We note that Bloom filters are history independent [19, 8].

Pseudo-Random Functions. Intuitively, a pseudo-random function is computationally indistinguishable from a random function — given pairs $(x_1, f(x_1, k)), \ldots, (x_m, f(x_m, k))$, an adversary cannot predict $f(x_{m+1}, k)$ for any x_{m+1}. More precisely, we say that a function $f : \{0,1\}^n \times \{0,1\}^s \to \{0,1\}^m$ is a (t, ϵ, q)-*pseudo-random function* if

1. $f(x, k) \stackrel{\text{def}}{=} f_k(x)$ can be computed efficiently from input $x \in \{0,1\}^n$ and key $k \in \{0,1\}^s$.
2. for any t time oracle algorithm $\mathcal{A}$ that makes at most q adaptive queries,

$$\left|\Pr\left[\mathcal{A}^{f(\cdot,k)} = 0 \mid k \xleftarrow{R} \{0,1\}^s\right] - \Pr\left[\mathcal{A}^g = 0 \mid g \xleftarrow{R} \{F : \{0,1\}^n \to \{0,1\}^m\}\right]\right| < \epsilon.$$

Pseudo-Random Generators. Intuitively, a pseudo-random generator outputs strings that are computationally indistinguishable from random strings. More precisely, we say that a function $G : \{0,1\}^n \to \{0,1\}^m$ where $m > n$ is a (t, ϵ)-*pseudo-random generator* if

1. G is efficiently computable by a deterministic algorithm
2. For all t time probabilistic algorithms $\mathcal{A}$,

$$\left|\Pr\left[A(G(s)) = 0 \mid s \xleftarrow{\mathrm{R}} \{0,1\}^n\right] - \Pr\left[A(r) = 0 \mid r \xleftarrow{\mathrm{R}} \{0,1\}^m\right]\right| < \epsilon.$$

Our secure index construction uses pseudo-random functions as shown in the following lemma —

Lemma 3.1. *If $f : \{0,1\}^n \times \{0,1\}^s \to \{0,1\}^m$ is a (t, ϵ, q)-pseudo-random function, then*

$$G(k) = f(1,k) \parallel f(2,k) \parallel \ldots \parallel f(q,k)$$

where $k \in \{0,1\}^s$ is a $(t - q, \epsilon)$-pseudo-random generator.

The proof is straightforward and is omitted. Note that if f is a random function, the derived generator G is unconditionally secure.

3.2 Construction

We use a Bloom filter as a per document index to track words in each document. In our scheme, a word is represented in an index by a codeword derived by applying pseudo-random functions twice — once with the word as input and once with a unique document identifier as input. This non-standard use of pseudo-random functions ensures that the codewords representing a word x are different for each document in the set, and this technique together with blinding indexes with random tokens, ensures that our indexes are IND-CKA secure. Furthermore, the different codewords for a word x can be efficiently reproduced given just a short trapdoor for x. Both efficiently computable codewords and short trapdoors are important for the application of searching on encrypted data. We now describe our index called Z-IDX.

`Keygen`(s): Given a security parameter s, choose a pseudo-random function $f : \{0,1\}^n \times \{0,1\}^s \to \{0,1\}^s$ and the master key $K_{\text{priv}} = (k_1, \ldots, k_r) \xleftarrow{\mathrm{R}} \{0,1\}^{sr}$.

`Trapdoor`(K_{priv}, w): Given the master key $K_{\text{priv}} = (k_1, \ldots, k_r) \in \{0,1\}^{sr}$ and word w, output the trapdoor for word w as $\mathrm{T}_w = (f(w, k_1), \ldots, f(w, k_r)) \in \{0,1\}^{sr}$.

`BuildIndex`(D, K_{priv}): The input is the document D comprising of an unique identifier (name) $D_{\text{id}} \in \{0,1\}^n$ and a list of words $(w_0, \ldots, w_t) \in \{0,1\}^{nt}$, and $K_{\text{priv}} = (k_1, \ldots, k_r) \in \{0,1\}^{sr}$.

1. For each unique word w_i for $i \in [0, t]$, compute —
   (a) the trapdoor: $(x_1 = f(w_i, k_1), \ldots, x_r = f(w_i, k_r)) \in \{0,1\}^{sr}$,
   (b) the codeword for w_i in D_{id}: $(y_1 = f(D_{\text{id}}, x_1), \ldots, y_r = f(D_{\text{id}}, x_r)) \in \{0,1\}^{sr}$,
   (c) and insert the codeword $y_1, \ldots, y_r$ into document D_{id}'s Bloom filter BF.
2. Compute an upper bound u on the number of tokens for D based on the size of D after it has been encrypted (that is, total word length $|w_0| + \ldots + |w_t| +$ possible padding length). A reasonable value of u can be derived by assuming 1 token for every byte.

3. Let v be the number of unique words among the set of t words $(w_0, \ldots, w_t)$. Blind the index by inserting $(u - v) \cdot r$ number of 1's uniformly at random in the Bloom filter (possibly with repetitions). Note that this is equivalent to adding $u - v$ random words into the index, but without any pseudo-random function computations.
4. Output $\mathcal{I}_{D_{\mathrm{id}}} = (D_{\mathrm{id}}\,, \mathrm{BF})$ as the index for D_{id}.

`SearchIndex`$(\mathrm{T}_w, \mathcal{I}_D)$: The input is the trapdoor $\mathrm{T}_w = (x_1, \ldots, x_r) \in \{0,1\}^{sr}$ for word w and the index $\mathcal{I}_{D_{\mathrm{id}}} = (D_{\mathrm{id}}\,, \mathrm{BF})$ for document D_{id}.

1. Compute the codeword for w in D_{id}: $(y_1 = f(D_{\mathrm{id}}, x_1)\,, \ldots, y_r = f(D_{\mathrm{id}}, x_r)) \in \{0,1\}^{sr}$.
2. Test if BF contains 1's in all r locations denoted by $y_1, \ldots, y_r$.
3. If so, output 1; Otherwise, output 0.

Note 1. The obvious idea of inserting the trapdoors $\mathrm{T}_x = f(x, k_1), \ldots, f(x, k_r)$ directly into the Bloom filter index makes the index vulnerable to correlation attacks where the similarity of two documents can be deduced by comparing Bloom filters indexes for overlaps, or lack thereof, of 1's in the Bloom filter. A similar attack on document updates can be mounted by tracking document Bloom filter changes between updates. Such correlation attacks are prevented by using pseudo-random functions such that codewords are the output of a pseudo-random generator (by Lemma 3.1) across all indexes.

Two equal length documents have the same number of tokens in their indexes, regardless of how many actual keywords the documents contain. The main disadvantage of adding random 1's into the Bloom filter is increasing the false positive rate; But this disadvantage is removed by increasing the size of the Bloom filter array when determining Bloom filter parameters using the procedure in Section 5. In fact, a simple calculation shows that it is sufficient to increase the Bloom filter array by a small constant factor to attain the same false positive rate as before.

Note 2. We delay the discussion of choosing suitable Bloom filter parameters until Section 5. In practice, we use the keyed hash function HMAC-SHA1 : $\{0,1\}^* \times \{0,1\}^{160} \rightarrow \{0,1\}^{160}$ [1] as the pseudo-random function f, which allows Z-IDX to handle arbitrary length words. Alternatively, we can use provably secure number theoretic pseudo-random functions [18], but these are much slower.

Cost of Algorithms. Since a 866 MHz Pentium 3 machine can compute roughly 150,000 HMAC-SHA1 evaluations per second on 16 byte inputs, it is reasonable to assume that computing a pseudo-random function is a constant time operation. We also assume that testing/inserting entries into a Bloom filter are constant time operations. As a result, the `Trapdoor` algorithm takes $O(1)$ time and produces trapdoors that are rs bits long; The `BuildIndex` algorithm on a document takes time linear in the number of words in the document; The `SearchIndex` algorithm takes $O(1)$ time.

Properties. The Z-IDX scheme possesses several useful properties —

1. Arbitrary Updates: Because Z-IDX does not pre-build dictionaries, it can handle *any* updates.
2. Compressible Indexes: Using techniques described in Section 5, the Bloom filter indexes can be very effectively compressed for small and medium sized documents.
3. Short Trapdoors and Efficiently Reproducible Codewords: Given the short trapdoor for word w, the different codewords of w for each index can be efficiently reproduced.

4. Boolean and Limited Regular Expression Queries: These queries can be performed efficiently on Z-IDX indexes; The algorithms for performing such queries are described in Section 4.1.
5. Occurrence Search: Z-IDX can be easily modified handle queries such as "does 'foo' occur at least twice in the index". This extension is described in Section 3.3.
6. Compressed and Encrypted Data: Index schemes do not require access to the document contents after the index is built. Hence, the documents can be compressed and encrypted with *any* compression algorithm and chosen ciphertext secure [14, 3, 21] cipher.
7. Variable Length Words: when HMAC-SHA1 is used as the pseudo-random function f.
8. Simple Key Management: As described, our construction requires r keys but the r keys can be generated by a pseudo-random generator with a single secret seed.

Theorem 3.2. *If f is a (t, ϵ, q)-pseudo-random function, then* Z-IDX *is a $(t, \epsilon, q/2)$-*IND-CKA *index.*

Proof. We prove the theorem using its contrapositive. Suppose Z-IDX is not a (t, ϵ, q)-IND-CKA index. Then there exists an algorithm $\mathcal{A}$ that (t, ϵ, q)-breaks Z-IDX. We build an algorithm $\mathcal{B}$ that uses $\mathcal{A}$ to determine if f is a pseudo-random function or a random function. $\mathcal{B}$ has access to an oracle $\mathcal{O}_f$ for the unknown function f that takes as input $x \in \{0,1\}^n$ and returns $f(x) \in \{0,1\}^s$. When running any of the four index algorithms, $\mathcal{B}$ substitutes evaluations of f with queries to the oracle $\mathcal{O}_f$. Algorithm $\mathcal{B}$ simulates $\mathcal{A}$ as follows —

Setup. Algorithm $\mathcal{B}$ picks a set S of $q/2$ strings from $\{0,1\}^n$ uniformly at random and sends S to $\mathcal{A}$. In response, $\mathcal{A}$ returns a polynomial number of subsets of S. We call this collection of subsets S^*. For each subset D in S^*, $\mathcal{B}$ assigns a unique id D_{id}, and runs `BuildIndex` on D with identifier D_{id} to obtain the index $\mathcal{I}_{D_{\text{id}}}$. After creating all the indexes, $\mathcal{B}$ gives all indexes and their associated subsets to $\mathcal{A}$.

Queries. On receiving $\mathcal{A}$'s query for the trapdoor of word x, $\mathcal{B}$ runs `Trapdoor` on x and replies with the trapdoor T_x.

Challenge. After making some `Trapdoor` queries, $\mathcal{A}$ decides on a challenge by picking a subset $V_0 \in S^*$, and generating another subset V_1 from S such that $|V_0 - V_1| \neq 0, |V_1 - V_0| \neq 0$, and the total length of words in V_0 is equal to that in V_1. Furthermore, $\mathcal{A}$ must not have queried $\mathcal{B}$ for the trapdoor of any word in $V_0 \bowtie V_1$. Next, $\mathcal{A}$ gives V_0 and V_1 to $\mathcal{B}$ who chooses $b \xleftarrow{\text{R}} \{0,1\}$ and also chooses a new (unique) identifier V_{id}. $\mathcal{B}$ runs `BuildIndex` on V_b with the identifier V_{id} to get the index $\mathcal{I}_{V_b}$, which is given to $\mathcal{A}$. After the challenge is issued, $\mathcal{A}$ is not allowed to query $\mathcal{C}$ for the trapdoors of any word $x \in V_0 \bowtie V_1$.

Response. $\mathcal{A}$ eventually outputs a bit b', representing its guess for b. If $b' = b$, then $\mathcal{B}$ outputs 0, indicating that it guesses that f is a pseudo-random function. Otherwise, $\mathcal{B}$ outputs 1.

We see that $\mathcal{B}$ takes at most t time because $\mathcal{A}$ takes at most t time. Furthermore, $\mathcal{B}$ makes at most q queries to $\mathcal{O}_f$ because there are only $q/2$ strings in S and $\mathcal{A}$ makes at most $q/2$ queries. Finally, we show that $\mathcal{B}$ can determine if f is a pseudo-random function or random function with advantage greater than ϵ by proving the following two claims —

Claim 1: When f is a pseudo-random function, then $\left|\Pr\left[\mathcal{B}^{f(\cdot,k)} = 0 \mid k \xleftarrow{\text{R}} \{0,1\}^s\right] - \frac{1}{2}\right| \geq \epsilon$.

Claim 2: When f is a random function, then $\Pr\left[\mathcal{B}^g = 0 \mid g \xleftarrow{\text{R}} \{F : \{0,1\}^n \to \{0,1\}^s\}\right] = \frac{1}{2}$.

It follows from these two claims that

$$\left|\Pr\left[\mathcal{B}^{f(\cdot,k)} = 0 \mid k \xleftarrow{\text{R}} \{0,1\}^s\right] - \Pr\left[\mathcal{B}^g = 0 \mid g \xleftarrow{\text{R}} \{F : \{0,1\}^n \to \{0,1\}^m\}\right]\right| \geq \epsilon,$$

thus proving the theorem.

It remains to prove the two claims. The proof of Claim 1 is immediate. When f is a pseudo-random function, algorithm $\mathcal{B}$ simulates perfectly the challenger in an IND-CKA game. Therefore, the claim follows by the definition of $\mathcal{A}$.

We now prove Claim 2. We first show that we only need to consider the challenge subsets in our analysis because the other subsets of S^* and further trapdoor queries reveal no information about the challenge subsets. When f is a pseudo-random function, Lemma 3.1 implies that it is infeasible for $\mathcal{A}$ to correlate codewords representing the same word across all document Z-IDX indexes. Since f is a random function, the same lemma applied to random functions implies that it is information theoretically impossible for $\mathcal{A}$ to correlate codewords across document Bloom filters. From this extended lemma, together with both restrictions on the choice of the challenge subsets V_0 and V_1 and on $\mathcal{A}$'s queries after receiving the challenge index, it follows that from $\mathcal{A}$'s view, each codeword representing a word $z \in V_0 \bowtie V_1$ is independent of all other codewords for z across all subsets in S^* and their indexes. That is, $\mathcal{A}$ learns nothing about $V_0 \bowtie V_1$ from the other subsets in S^* and their indexes. As a result, we need only consider the challenge subsets.

Without loss of generality, assume that $V_0 \bowtie V_1$ contains only two words, x and y where $x \in V_0$ and $y \in V_1$. Suppose $\mathcal{A}$ guesses b correctly with advantage $\delta > 0$. It follows that $\mathcal{A}$, given $f(z)$, can determine if $z = x$ or y with advantage δ; That is, $\mathcal{A}$ can distinguish between outputs of a random function f with advantage δ, which is impossible. Therefore, $\mathcal{A}$ at best guesses b correctly with probability 1/2. It follows that $\Pr\left[\mathcal{B}^g = 0 \;\middle|\; g \xleftarrow{\mathrm{R}} \{F : \{0,1\}^n \rightarrow \{0,1\}^s\}\right] = \Pr\left[b' = b\right] = \frac{1}{2}$, thus proving Claim 2. □

Corollaries. The results in this section show that Z-IDX indexes can be used 1) to test set membership without revealing the set elements, and 2) for accumulated hashing [5, 20].

Attaining IND2-CKA Security. Z-IDX is easily extended to achieve IND2-CKA security, which is similar to that proposed by Chang and Mitzenmacher [9], where the indexes for two (possibly unequal size) documents with different numbers of keywords cannot be distinguished. Only the `BuildIndex` algorithm is modified — instead of estimating an upper bound u on the number of tokens for each document, simply set a global u for all documents that is an estimate for the maximum number of tokens in a single document. The maximum number of unique words u can be estimated while scanning and tokenizing the documents. The proof of security of this variant is similar to that given for Z-IDX and is omitted.

3.3 Occurrence Search

Z-IDX can be modified handle queries such as "does 'foo' occur at least twice in the index". Only the `Trapdoor` and `BuildIndex` algorithms need to change.

`Trapdoor`$(K_{\mathrm{priv}}, w, i)$: Given the master key $K_{\mathrm{priv}} = (k_1, \ldots, k_r) \in \{0,1\}^{sr}$, the word w, and the occurrence desired i, output $\mathrm{T}_w = (f(i \parallel w, k_1), \ldots, f(i \parallel w, k_r)) \in \{0,1\}^{sr}$ as the trapdoor.

`BuildIndex`(D, K_{priv}): The input is the document D comprising of an unique identifier (name) $D_{\mathrm{id}} \in \{0,1\}^n$ together with a list of words $(w_0, \ldots, w_t) \in \{0,1\}^{nt}$ and the master key $K_{\mathrm{priv}} = (k_1, \ldots, k_r) \in \{0,1\}^{sr}$. For each word w_i for $i \in [0, t]$, do the following —

1. Let z_i be the order of occurrence of w_i relative to previous occurrences of w_i in D. That is, if $w_i =$ 'foo' is the second time word 'foo' occurs in D, then $z_i = 2$.
2. Compute a modified trapdoor: $(x_1 = f(z_i \| w_i, k_1), \ldots, x_r = f(z_i \| w_i, k_r)) \in \{0,1\}^{sr}$,
3. Compute the codeword for w_i in D_{id}: $(y_1 = f(D_{\mathrm{id}}, x_1), \ldots, y_r = f(D_{\mathrm{id}}, x_r)) \in \{0,1\}^{sr}$,
4. Insert the codeword $y_1, \ldots, y_r$ into document D_{id}'s Bloom filter BF.

The index is blinded in the same way as before and $\mathcal{I}_{D_{\mathrm{id}}} = (D_{\mathrm{id}}, \mathrm{BF})$ is emitted as the index.

4 Secure Indexes Applied to Searching on Encrypted Data

As an application of secure indexes, we show how to use Z-IDX for searching on encrypted documents stored on a remote server (which was motivated in Section 1). The eight properties of Z-IDX discussed in Section 3.2 make Z-IDX indexes especially well suited for this application.

On a word search, recall that the querier sends the trapdoor for the word to the server and the server returns matching documents. In Z-IDX indexes, the trapdoor does not reveal the actual search word, and the server cannot generate trapdoors because it does not possess the keys. After a search, the server learns nothing about the documents than the search result. Furthermore, document updates are secure. These security properties and others are a direct consequence of the IND-CKA security model. Before describing the setup, search, and update algorithms for implementing searches on encrypted data, we briefly review some related work.

Related Work on Searching on Encrypted Data. Song et al. developed a Searchable Symmetric Key Encryption (SSKE) scheme [22] that allows a user, given a trapdoor for a word, to test if a ciphertext block contains the word. Boneh et al. developed Searchable Public Key Encryption (SPKE) [7], which is the public key equivalent of SSKE. When applied to the problem of searching encrypted documents, both schemes are inefficient because the server performs a linear scan through all the documents on every query. Song et al. [22] also consider encrypted hash table indexes but their solution is incomplete because their index updates are insecure. Chang and Mitzenmacher [9] developed two index schemes using dictionaries for searching on encrypted data. Like all of these schemes when applied to the problem of searching on encrypted data on a remote server, our search scheme using Z-IDX indexes sacrifices access pattern privacy for efficiency. For example, after a search for keyword x, the server does not learn x but learns which documents contain x. Our solution, however, is computationally more efficient than all schemes to date.

The seminal work of Goldreich and Ostrovsky on Oblivious RAMs [13] give techniques for searching on encrypted data stored on a server while hiding data access patterns. Their techniques require polylog rounds of communication. Although their best construction is asymptotically efficient, it is inefficient in practice because the big-O notation hides large constants. Note that the usage model here is different from that in the preceding paragraph; Here, the user performs a keyword search in a number of rounds by iteratively querying the server so as to narrow down the search, whereas for the schemes in the preceding paragraph, a user queries the server only once (in most cases) and the server performs the search, eventually returning the search result. Private Information Retrieval (PIR) [11, 15] schemes allow queries to a database where the database learns nothing about which records were read. Chor et al. show how to use multiple rounds of PIR to search on keywords [10]. On the other hand, PIR schemes are meant for public databases where the data is unencrypted on the server.

4.1 Setup, Search, and Update Algorithms

Suppose a user $\mathcal{U}$ stores a set of n documents $D_1, \ldots, D_n$ on an untrusted server $\mathcal{S}$. The search system is set up by running the `ES-Setup` algorithm to build indexes for $D_1, \ldots, D_n$. The `ES-Search` algorithm is used for performing searches. If documents are added, deleted, or altered, the `ES-Update` algorithm updates the indexes. HMAC-SHA1 [2] is used as the pseudo-random function f.

`ES-Setup`: Before the n documents are placed on the server, the indexes are built as follows —

1. First derive suitable Bloom filter parameters for the indexes. We delay the discussion of choosing suitable parameters until Section 5. For now, assume that the Bloom filters are instantiated with an array of size m. Next invoke `Keygen`(s) to obtain the pseudo-random function $f : \{0,1\}^* \times \{0,1\}^s \rightarrow \{0,1\}^s$ and the master key $K_{\text{priv}} = (k_1, \ldots, k_r) \in \{0,1\}^{sr}$.
2. Assign (and insert) an integer $i \in [1, n]$ to each document D_i as its unique identifier.
3. For each document D_i, build its index $\mathcal{I}_{D_i} \leftarrow$ `BuildIndex`(D_i, K_{priv}).
4. Compress and encrypt each document using standard algorithms before transferring the documents and their indexes to the server $\mathcal{S}$.

Cost of `ES-Setup`: It is easy to see that the cost is linear with the total size of documents.

`ES-Search`: When a user $\mathcal{U}$ wants all documents on the server $\mathcal{S}$ containing the word y, the following actions are performed —

1. $\mathcal{U}$ computes the trapdoor $\mathrm{T}_y \leftarrow$ `Trapdoor`(K_{priv}, y) for y and sends T_y to the server.
2. For every index $\mathcal{I}_{D_i}$, $\mathcal{S}$ invokes `SearchIndex`$(\mathrm{T}_y, \mathcal{I}_{D_i})$ to test for a match. All matching documents are returned to $\mathcal{U}$.

Cost of `ES-Search`: $\mathcal{U}$ takes $O(1)$ time to compute the trapdoor. Each `SearchIndex` evaluation takes $O(1)$ time and $\mathcal{S}$ runs it on all documents; Therefore, the cost of searching all documents is linear in the number of documents (and not the size of the documents). We note that simultaneous word searches can be combined and will only require one pass through all the indexes.

Boolean and Regular Expression Queries. The `ES-Search` algorithm can also efficiently perform "AND" and "OR" boolean queries involving multiple words.[4] The procedure for handling these queries is best illustrated with an example. Suppose the user $\mathcal{U}$ wants all documents containing words x AND y. $\mathcal{U}$ first computes the trapdoors for both x and y and gives the server $\mathcal{S}$ both T_x and T_y. With a single pass over all indexes, $\mathcal{S}$ invokes `SearchIndex` once on each trapdoor. Documents whose indexes match both trapdoors are returned to $\mathcal{U}$. "OR" queries can be carried out in a similar manner. The cost of such boolean queries is linear with the number of terms in the boolean expression, but can be completed with a single pass over all documents, whereas the naive method of performing such boolean queries involves multiple passes over the documents.

Certain regular expression queries such as "$ab[a - z]$" can be expressed as boolean queries $(aba \wedge \ldots \wedge abz)$. Hence, these queries can also be done with a single pass over all documents. Wild-card regular expression queries such as "$ab*$" are much harder because of the exponential blowup in the number of possible strings.

[4] This technique is aimed at increasing the efficiency of such queries and is only as secure as performing individual queries for each term.

n	$m = \sqrt{n}$	$(1 - \frac{1}{9\sqrt{n}})^{m-2}$
64	8.00	0.920
1024	32.00	0.901
16384	128.00	0.896
131072	362.04	0.895
67108864	8192.00	0.895

Figure 1: Probability that none of the $m-2$ queries on n documents return all n documents

`ES-Update:` There are three possible types of updates —

1. Adding Documents: To add a new document D, the `BuildIndex` algorithm is used to build D's index after D is assigned a unique identifier.
2. Deleting Documents: Deletions simply involve deleting the document and its index from $\mathcal{S}$.
3. Altering Document Contents: Altering the contents of an existing document requires assigning the document a new (unique) number and regenerating its index by invoking `BuildIndex` on the document with the new identifier.

Cost of `ES-Update`: Deleting a document is a constant time operation. Adding or updating a document requires invoking `BuildIndex`, which has cost linear in the size of the document.

4.2 Extensions

We describe three useful extensions to the basic encrypted search algorithms.

1. Heuristically Increasing Security. We describe a technique that makes it harder for the server to identify duplicate queries for the same word.

Technique. To search for word y, the user computes the trapdoor $\mathrm{T}_y = f(y, k_1), \ldots, f(y, k_r)$ but instead of sending all r pseudo-random outputs to the server, she sends only $r/2$ randomly chosen outputs. Note that the index is still built using all r outputs.

Benefits. In most normal situations, this technique makes it harder for an eavesdropper to identify multiple queries for the same word. On the other hand, since the server has only half of the trapdoor, the document indexes constructed using Bloom filters register more false positives. A higher rate of false positives increases the communication overhead but more false positives also hide the documents that actually contain the keyword. Section 5 discusses the right choice of Bloom filter parameters to give the desired information hiding rate versus communication overhead.

Why Only Heuristic? In certain situations, the server can determine which truncated trapdoors refer to the same word with no extra work. We show this using an example. Assume we have a set of n documents, all of which contain a word x. Also assume that using the technique described above gives a false positive rate of $1/10$.

Suppose a user performs m queries, of which two are for the same word x and the other $m-2$ are queries for words not contained by any document in the set (non-existent words). The number of documents returned by a query for a non-existent word is given by the binomial distribution with $p = 1/10$. Observe that the two queries for x return all n documents. In contrast, the probability that a non-existent word query returns all n documents is bounded by the Chebyshev inequality to be at most $\frac{1}{9\sqrt{n}}$. Hence, the probability that all $m-2$ queries return less than n documents is at least $(1-\frac{1}{9\sqrt{n}})^{m-2}$. Figure 1 lists some sample values of n and the corresponding values of m where $m=\sqrt{n}$. From the table, we see that the probability that all $m-2$ queries return less than n documents is high even with relatively low values of m. Therefore, after all m queries are complete, an eavesdropper can pinpoint with high probability the two queries for x by observing which queries return the entire document set. The example given above is a contrived "bad" case. In many common scenarios, it is hard (or impossible) for the server to perform similar analysis on the query patterns.

2. Locating Words Within Documents. Instead of using an index for every document, we can divide documents into chunks and create indexes for each chunk. With this modification, words can be located with chunk size granularity within a document.

3. Searching for Infrequent Words. If preventing statistical analysis attacks (described in Section 3) is not required, then document indexes (Bloom filters) can be organized into a binary tree, which facilitates efficient searches for uncommon or non-existent words. Appendix A contains a complete description of this extension.

5 Choosing Suitable Bloom Filter Parameters

We first quantify the probability of a false positive occurring in a Bloom filter. Assume that the hash functions $h_1, \ldots, h_r$ map arbitrary strings uniformly over the range $[1, m]$. After using the r hash functions to insert n distinct elements into an array of size m, the probability that bit i in the array is 0 is $(1-(1/m))^{rn} \approx e^{-rn/m}$. Therefore, the probability of a false positive is $(1-(1-(1/m))^{rn})^r \approx (1-e^{-rn/m})^r$. Since m and n are typically fixed parameters, we compute the value of r that minimizes the false positive rate. The derivative of the right hand side of the equation with respect to r gives a global minimum of $r = (\ln 2)(m/n)$, with a false positive rate of $(1/2)^r$. We round r to the nearest integer in practice. Given a set of documents that we want to build indexes for, the following procedure shows how to choose optimal Bloom filter parameters.

1. First choose the desired false positive rate fp. Note that increasing the Bloom filter false positive rate causes the server to return more irrelevant documents in a word search. But irrelevant documents can be weeded out by mechanical scanning by the user after being retrieved and decrypted. Hence, increasing the false positive rate increases the communication overhead between the user and the server without affecting the correctness of the result.

2. From the false positive rate, compute the number of pseudo-random function keys required. Recall that in Section 3.1, a false positive rate of $(1/2)^r$ is achieved by choosing the number of pseudo-random function keys r using the equation $r = (\ln 2)(m/n)$ where m is the size of the Bloom filter array and n is the number of unique words in the document set. Therefore, the number of keys required can be determined by computing $r = -\log_2(fp)$.

3. Scan every document in the set and count the number of unique words. Multiply the number of unique words by a constant factor to allow for updates, giving us the required value of n.

We only need to consider the unique words in the document set and not all possible words in the universe. With r and n determined, the array size m is given by $m = nr/\ln 2$.

Reducing Bloom Filter Array Size. In many applications, individual documents are small and do not contain many words. For example, consider a collection of a user's emails. Since the number of words in each email is small, using an array is inefficient because most of the array entries are 0. Instead, we keep track of the entries containing 1's. For example, when the Bloom filter array is 2^{25} bits or 4 megabytes, each entry in the array can be represented using only $25 \approx 2^5$ bits. Here, this technique is effective for documents containing roughly less than 2^{17} tokens (assuming 8 hash functions are used). Using this technique even on a medium size document containing 5000 words, the Bloom filter index is only $5000 * 25 = 125000$ bits or approximately 15 kilobytes, representing a 99.63% reduction in size from a 4 megabyte array. Finally, observe that the large unique word set with small documents scenario where this technique is most effective frequently occurs in real applications. Other techniques for reducing the Bloom filter size include using compressed Bloom filters [17].

Acknowledgements

This work is supported by NSF Grant CCR-0205733, CCR-0331640, and the Packard Foundation. The author is grateful to Hovav Shacham for his help in refining the security model, and also for suggesting the right `sh` commands for calculating unique word counts. The author is also grateful to his advisor, Dan Boneh, for help in developing the security model, and providing editorial comments. Thanks to Eric Rescorla, Benny Pinkas, Vanessa Teague, and Rafi Ostrovsky for comments, to Burt Kaliski for pointing out the connection to accumulated hashing, and to Rajeev Motwani for pointing out an oddity in the original reduction whose investigation led to the improved model. Finally, thanks to Hovav Shacham and Eric Rescorla for suggesting the more secure extension.

References

[1] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In N. Koblitz, editor, *Proceedings of Crypto 1996*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, Aug 1996.

[2] M. Bellare, R. Canetti, and H. Krawczyk. HMAC: Keyed-hashing for message authentication. RFC 2104, Internet Engineering Task Force (IETF), Feb 1997.

[3] M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway. Relations among notions of security for public-key encryption schemes. In H. Krawczyk, editor, *Proceedings of Crypto 1998*, volume 1462 of *Lecture Notes in Computer Science*, pages 26–45. Springer-Verlag, Aug 1998.

[4] S. Bellovin and W. Cheswick. Privacy-enhanced searches using encrypted bloom filters. Cryptology ePrint Archive, Report 2004/022, Feb 2004. `http://eprint.iacr.org/2004/022/`.

[5] J. C. Benaloh and M. de Mare. One-way accumulators: A decentralized alternative to digital signatures. In T. Helleseth, editor, *Proceedings of Eurocrypt 1993*, volume 765 of *Lecture Notes in Computer Science*, pages 274–285. Springer-Verlag, May 1993.

[6] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, Jul 1970.

[7] D. Boneh, G. D. Crescenzo, R. Ostrovsky, and G. Persiano. Public-key encryption with keyword search. In C. Cachin, editor, *Proceedings of Eurocrypt 2004*, LNCS. Springer-Verlag, May 2004.

[8] N. Buchbinder and E. Petrank. Lower and upper bounds on obtaining history independence. In D. Boneh, editor, *Proceedings of Crypto 2003*, volume 2729 of *LNCS*, pages 445–462. Springer-Verlag, 2003.

[9] Y.-C. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. Cryptology ePrint Archive, Report 2004/051, Feb 2004. `http://eprint.iacr.org/2004/051/`.

[10] B. Chor, N. Gilboa, and M. Naor. Private information retrieval by keywords. Technical Report TR CS0917, Department of Computer Science, Technion, Feb 1998.

[11] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval. *Journal of the ACM*, 45(6):965–981, Nov 1998.

[12] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary cache: a scalable wide-area Web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, Jun 2000.

[13] O. Goldreich and R. Ostrovsky. Software protection and simulation on Oblivious RAMs. *Journal of the ACM*, 43(3):431–473, May 1996.

[14] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and Systems Sciences*, 28(2):270–299, Apr 1984.

[15] E. Kushilevitz and R. Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *Proceedings of the 38th Annual IEEE Symposium on the Foundations of Computer Science*, pages 364–373, Oct 1997.

[16] D. Micciancio. Oblivious data structures: Applications to cryptography. In *Proceedings of the 29th ACM Symposium on Theory of Computing*, pages 456–464. ACM Press, Jul 1997.

[17] M. Mitzenmacher. Compressed bloom filters. *IEEE/ACM Transactions on Networking*, 10(5):604–612, Oct 2002.

[18] M. Naor and O. Reingold. Number-theoretic constructions of efficient pseudo-random functions. In *Proceedings of the 38th IEEE Symposium on Foundations of Computer Science*, pages 458–467, 1997.

[19] M. Naor and V. Teague. Anti-persistence: History independent data structures. In *Proceedings of the 33rd ACM Symposium on Theory of Computing*, pages 492–501. ACM Press, Jul 2001.

[20] K. Nyberg. Fast accumulated hashing. In D. Gollman, editor, *Proceedings of the 3rd Workshop in Fast Software Encryption*, volume 1039 of *Lecture Notes in Computer Science*, pages 83–87. Springer-Verlag, Feb 1996.

[21] C. Rackoff and D. Simon. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In J. Feigenbaum, editor, *Proceedings of Crypto 1991*, volume 576 of *Lecture Notes in Computer Science*, pages 433–444. Springer-Verlag, Aug 1991.

[22] D. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 44–55. IEEE, May 2000.

[23] B. Waters, D. Balfanz, G. Durfee, and D. Smetters. Building an encrypted and searchable audit log. In *Proceedings of the 11th Network and Distributed System Security (NDSS) Symposium*, pages 205–214. Internet Society (ISOC), Feb 2004.

A Searching for Infrequent Words Efficiently

If preventing statistical analysis attacks (described in Section 3) is not required, then we can organize document indexes (Bloom filters) into a binary tree, which can be used to efficiently search for uncommon or non-existent words.

Bloom Filter Tree. In this modified scheme, document indexes (Bloom filters) contain the trapdoors $\mathrm{T}_x = f(x, k_1), \ldots, f(x, k_r)$ instead of codewords $f(D_{\mathrm{id}}, f(x, k_1))$, ... , $f(D_{\mathrm{id}}, f(x, k_r))$ as described in the original scheme. Note that document identifiers are no longer needed.

Observe that a bitwise OR operation between the arrays of two Bloom filters (indexes) representing sets S_1 and S_2 results in another bit array that represents the union of S_1 and S_2. The two Bloom filter arrays must be of the same length and must have been initialized with the same set of hash functions. We can now describe how to exploit this property to build a document index tree for a set of n documents. For ease of exposition, assume that n is a power of 2.

1. Divide the set of n documents into pairs of documents. The corresponding document indexes (Bloom filters) for each pair are the leaves of the index tree.

2. Compute the bitwise OR of each pair of document index (Bloom filters) and assign the resulting index (Bloom filter) as the parent node of the pair in the document tree.

   Note that this parent node represents the set of unique words in both documents. Also note that these parent nodes are the tree nodes at $\log n - 1$ depth.

3. Repeat the previous step for higher levels of the tree until we reach the root. Note that the index at the root contains the set of unique words in the entire document set.

Figure 2 shows an example of an index (Bloom filter) tree for a set of 4 documents. The tree can be built by the server or by the user and later transferred to the server. Note that the cost of building the tree is cheap because only approximately $2n$ Bloom filter bitwise ORs are computed for a set of n documents. Also note that documents are arbitrarily sorted to form pairs.

Searching a Bloom Filter Tree. We use the following procedure to search for all documents containing a word x in a set of n documents.

1. Perform a breadth-first traversal on the tree starting at the root. At every node i traversed, check if the index (Bloom filter) at node i contains the trapdoor for x.

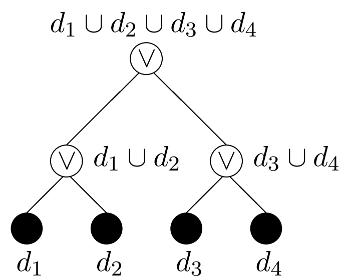

Figure 2: A Bloom filter tree for 4 documents.

2. If not, all nodes in the subtree rooted from node i are ignored for the rest of the search because no documents in the subtree rooted from node i contains x.

3. Otherwise, continue searching from node i. If i is a leaf node, the document represented by the index (Bloom filter) at i contains x.

Figure 3 illustrates the search procedure. In this example, only document d_5 contains word x. The dark lines and shaded circles mark the nodes traversed by the search algorithm.

Cost of Searching. If no document in the set contains the word, the word digest will not be found in the root node. Hence, the index (Bloom filter) tree allows us to rule out non-existent words with a single operation. Otherwise, the cost of searching for a word x in a set of n documents has an upper bound of $2v \log n$ Bloom filter tests where v is the number of documents containing x. This method of searching is better than linearly searching all indexes only if $v < n/2 \log n$.

Figure 4 plots the value of $v = n/2 \log n$ as a percentage of n versus n for n up to 2^{23}. The x-axis is plotted on a log scale. From the graph, we see that the efficiency of this method of searching drops as the document set grows in size. In a document set of size $2^{14} = 16384$, the tree construction is more efficient than linearly searching all document indexes if r is less than 6.25% of n (1024 out of 16384 documents). In a document set of size $2^{23} = 8388608$, the tree method is more efficient than linear search if v is less than 4% of n (335544 out of 8388608 documents). As the document set size increases, we observe a decrease in the rate at which v as a percentage of n drops. Therefore, this technique is still useful for very large document sets.

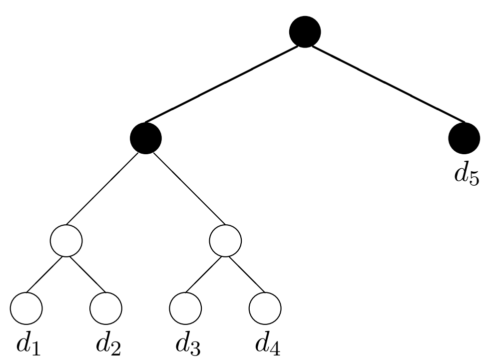

Figure 3: Searching a 5 document Bloom filter tree.

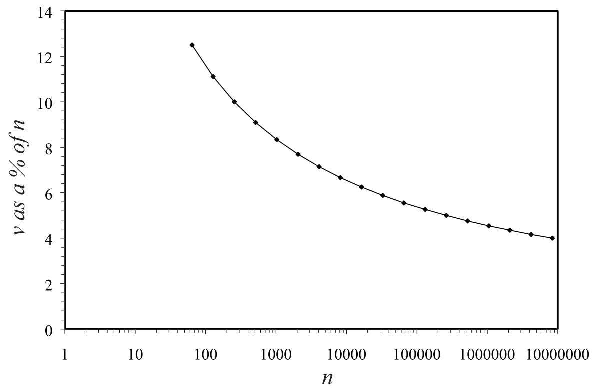

Figure 4: $v = n/2\log n$ as a percentage of n versus n.

In practice, the search cost can be heuristically reduced using the following technique — when building the index (Bloom filter) tree, choose pairs of documents such that the Hamming distance of each pair's indexes (Bloom filters) is minimized.

Updates on an Index Tree. Adding (or deleting) a document now requires an extra step for adding (or deleting) a new leaf node to the tree and then propagating the changes up to the root. Altering the contents of a document also requires an additional step of regenerating the Bloom filter nodes from the document leaf node back to the tree root.

Deleting a document or altering a document might remove unique words from the document set. Recall that standard Bloom filters cannot handle word deletions. Therefore, we use "counting Bloom filters" [12] at all non-leaf nodes.

Cost of Updates. Adding a new document, deleting a document, or changing document contents now incur an additional cost of updating the index (Bloom filter) tree. Note that changes to the tree on an update only affect the subtree where the changes occur. Therefore, the update cost has an upper bound of $2\log n$ where n is the new number of documents in the tree. The communication overhead remains the same as in the basic scheme.