Building Secure Indexes for Searching Efficiently on Encrypted Compressed Data

Eu-Jin Goh
eujin@cs.stanford.edu

October 29, 2003

Abstract

This paper focusses on building secure keyword indexes that allow efficient search on encrypted compressed documents. Our indexes are constructed using Bloom filters and pseudo-random functions. Our construction requires only $O(1)$ search time per document, and can, without any modification, handle both variable length keywords, and boolean and certain regular expression queries. Unlike previous index schemes, index updates in our scheme are provably secure. Key management is simple with only one key. Furthermore, our scheme, being index-based, is indifferent to the compression and encryption algorithm used on the documents. Our security model for indexes, called semantically security against against adaptive chosen keyword attack, is of independent interest because it provides very strong security guarantees in a realistic setting. We prove that our index satisfies this model. Apart from the application to keyword search, our Bloom filter construction can also be used for accumulated hashing, and also in any application that requires a security preserving test of set membership.

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. Previous attempts at building secure indexes [26] are incomplete because their indexes cannot securely handle updates. In this paper, we focus on the problem of building indexes that are semantically secure against adaptive chosen keyword attack. Our interest in secure indexes is primarily motivated by the application of searching on encrypted data, but as we will see later, our construction can also be used for accumulated hashing and secure set membership tests.

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, it is prudent to either use untrusted storage systems [16, 19] that encrypt data on before remote storage, or to manually encrypt sensitive documents before placing them on remote servers.

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. We discuss these design and security goals in Section 3.

Overview. Our construction requires only $O(1)$ search time per document, and can, without any modification, handle both variable length words, and boolean and certain regular expression queries. Furthermore, index updates in our scheme are provably secure. Key management is simple with only one key. Our scheme, being index-based, is indifferent to the compression and encryption algorithm used on the documents. We defer a discussion of related work to Section 9.

Of separate interest is the development of our security model of semantically security against adaptive chosen keyword attack for indexes. An index that satisfies our security model provides very strong security in the following sense — Consider a document P containing n words, of which an adversary already knows m words and wants information about the $n-m$ unknown words. We call the set of $n-m$ unknown words W. Even when the adversary is given plain text access to all other documents except on W, and is also allowed to make arbitrary queries of its choice on any word except those in W, the adversary still cannot obtain any information about any word in W from P's index. Our model captures the intuitive notion that an adversary learns nothing about a document from its index other than what it already knows from previous query results or other channels.

2 Background

In our construction, Bloom filters are used with pseudo-random functions for security. We spend the rest of this section describing both Bloom filters and pseudo-random functions. Readers familiar with the material can skip to the next section.

2.1 Bloom Filters

Historical Overview. Bloom filters [8] were introduced by Burton Bloom in 1970 as a data structure for efficiently testing set membership. Knuth mentions that Bloom filters are an application of "superimposed coding" [17] to search problems. Bloom filters have been used in the database community for efficiently implementing semi-join operations [21], and in the caching community for web cache sharing [14].

Description. A Bloom filter 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 $f_1, \ldots, f_r$, where $f_i : \{0,1\}^* \rightarrow [1, m]$ for $1 \leq i \leq r$. For each element $s \in S$, the array bits at positions $f_1(s), \ldots, f_r(s)$ 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 $f_1(a), \ldots, f_r(a)$. 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 of the checked bits is 0, then a is definitely not a member of S.

Analysis. We quantify the probability of a false positive occurring. Assume that the hash functions map each element uniformly over the range $[1, m]$. After using the r hash functions to insert n distinct elements into the array of size m, the probability that bit i in the array is 0 is

$$\left(1 - \frac{1}{m}\right)^{rn} \approx e^{-rn/m}.$$

Therefore, the probability of a false positive is

$$\left(1 - \left(1 - \frac{1}{m}\right)^{rn}\right)^r \approx (1 - e^{-rn/m})^r.$$

Since m and n typically are 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 $(1/2)^r$. In practice, r is rounded to the nearest integer.

2.2 Pseudo-Random Functions

Intuitively, a pseudo-random function is indistinguishable from a random function — given $x_1, \ldots, x_m$ and $F_k(x_1), \ldots, F_k(x_m)$, an adversary cannot predict $F_k(x_{m+1})$ for any x_{m+1}. We formalize this intuition with the following definition.

Definition 2.1 (Pseudo-Random Function). A function $f : \{0,1\}^n \times \{0,1\}^s \to \{0,1\}^m$ is a (t, ϵ, q)-*pseudo-random function* if

1. given input $x \in \{0,1\}^n$ and key $k \in \{0,1\}^s$, there is an efficient algorithm to compute

$$f_k(x) \stackrel{\text{def}}{=} f(x, k).$$

2. for any t time oracle algorithm $\mathcal{A}$,

$$\left| \Pr_{k \leftarrow \{0,1\}^s}[\mathcal{A}^{f_k} = \texttt{rand}] - \Pr_{g \leftarrow F}[\mathcal{A}^g = \texttt{rand}] \right| < \epsilon$$

where $F = \{g : \{0,1\}^n \to \{0,1\}^m\}$ and $\mathcal{A}$ makes at most q queries to the oracle adaptively.

In practice, we use HMAC-SHA1 [4, 5] as a pseudo-random function.

3 Design and Security Goals.

We first give a high level outline of our goals before describing our construction.

Minimize Communication and Computational Overhead. When searching for documents containing a keyword, Alice sends only a small codeword representing the word, and the server returns only those documents containing the keyword. In addition, we want to minimize the amount of computation the client and server performs on a search.

Multi-user Setting. If we are interested only in a single user setting, an obvious solution is to build a local index. For example, the local index can be a hash table where each bucket contains all documents with a particular keyword. In this case, when Alice searches for documents containing a keyword, she first searches her local index and then retrieves the relevant documents.

Solutions using local indexes do not work well in multi-user scenarios where many users share a group of document files. When documents are added or deleted from the group, or if documents are altered, the set of words contained in the document set change. When these changes occur, all indexes must be updated. If the indexes are local, it is hard to propagate the updates to all users. In this paper, we only consider solutions that can handle both single and multi-user scenarios. Our constructions use indexes stored on a remote server, which ensures that users always access the latest version of the index.

Security versus Efficiency Tradeoff. This paper aims for practical constructions that can be immediately deployed. Song, Wagner, and Perrig (SWP) [26] were the first to consider practical search constructions. Like the SWP solutions, our scheme does not hide data access pattern hiding that Private Information Retrieval (PIR) schemes [12] provide. For example, after a search for keyword x using SWP or our construction, the server does not learn what the keyword is but knows which document files contain x. Our solution, however, is vastly more efficient than either SWP or PIR schemes.

Security Goals. Recall that the user sends a codeword representing a word to the server and the server returns matching documents. The coded keyword query must not reveal the actual search word, and the server cannot generate coded queries without the user. After a search, the server should learn nothing more about the documents than the search result. Indexes should reveal nothing about the document even on updates. We capture these properties in the security model of semantic security against adaptive chosen keyword attack.

4 Design

The basic design uses a Bloom filter per document as an index to track words in each document. In our scheme, words are represented by codewords, which are derived using pseudo-random functions. These codewords are called "word digests".

Definition 4.1 (Word Digest). Let $x \in \{0,1\}^*$. The *word digest* of x is defined to be $W(x) = f_1(x), \ldots, f_r(x)$, where $f_j(x)$ denotes computing the pseudo-random function f with key j on x.

We describe our index setup, search, and update algorithms by using the index for searching encrypted files on a remote server.

4.1 Setup Algorithm

Suppose a user $\mathcal{U}$ stores a set of n documents on an untrusted server $\mathcal{B}$ and wants to use our index to search. To achieve this, the following steps are carried out —

1. First derive suitable Bloom filter parameters. We delay the discussion of choosing suitable parameters until Section 7. For now, assume that the Bloom filter is instantiated with an

array of size m. Instead of using r independent hash functions, we use the same pseudo-random function f with r different keys. These keys are chosen uniformly at random from the key space.

2. Sequentially assign an integer from the range $[1, n]$ to each document.

3. Scan each document — for each word x in document j, first compute $W(x) = f_1(x), \ldots, f_r(x)$ and then insert $f_{f_1(x)}(j), \ldots, f_{f_r(x)}(j)$ into document j's Bloom filter. This procedure is carried out for all n documents, creating n document Bloom filters.

4. Compress and encrypt each document using standard algorithms. Mark each document and its associated Bloom filter with the assigned integer label before transferring the documents and their Bloom filters to $\mathcal{B}$.

Cost of Setup. It is easy to see that the cost of setup is linear with the size of documents.

Statistical Analysis Across Document Bloom Filters. In Step 3, the obvious idea is to insert the word digests $f_1(x), \ldots, f_r(x)$ directly into the document Bloom filter instead of the non-standard technique of applying pseudo-random functions that we have chosen.[1] The reason for this design decision is to prevent statistical analysis across a set of document Bloom filters.

As explained in Section 9, keyword hash tables are unsuitable for indexes because updates reveal too much information. Bloom filters suffer from the same problem if we insert the word digests directly into document Bloom filters. By comparing document Bloom filters, the overlap, or lack thereof, of 1's in the filter allows an adversary to deduce if two documents contain a similar set of unique words. A similar statistical attack on document updates can be mounted by tracking document Bloom filter changes between updates.

Nevertheless, observe that these attacks are effective only if the codeword representing a word remains the same across all documents. The simplest solution based on this observation is to use different keys for every document. This solution, however, leads to key proliferation and even worse, the user has to compute a codeword for every document and send it to the server on every query. On the other hand, the non-standard technique we have chosen allows us to vary the word digest while using only a small (constant) number of keys. In fact, Lemma 5.2 shows that our non-standard usage of the pseudo-random function transforms it into a pseudo-random generator. Therefore, it is impossible to correlate the words contained in the document Bloom filters across all the documents. This fact is crucial in the security proof of this construction.

4.2 Search Algorithm

Suppose a user $\mathcal{U}$ wants to obtain all documents on the untrusted server $\mathcal{B}$ that contain a word y. To achieve this, the following procedure is carried out —

1. $\mathcal{U}$ derives the word digest of y by computing $W(x) = h_1(y), \ldots, h_r(y)$. $\mathcal{U}$ sends the word digest of y to the server $\mathcal{B}$.

2. For every document i in the set of n documents, $\mathcal{B}$ computes $h_{h_1(y)}(i), \ldots, h_{h_r(y)}(i)$ and tests document i's Bloom filter for a match. All matching documents are returned to $\mathcal{U}$.

[1] It is easy to see that using pseudo-random functions in the standard way with Bloom filters results in an accumulated hashing scheme [7, 23]. This scheme's security is a direct result of the security proof given in Section 5.2.

Cost of Searching. The server tests a Bloom filter for a word by applying the pseudo-random function twice, and then checking r locations in the Bloom filter array. If we assume that computing a pseudo-random function and checking a Bloom filter location are both unit time operations, then searching a document for one word takes $3r = O(1)$ time. Using the same unit time assumption, the time taken in searching a set of documents for a word is linear in the number of documents. Notice that simultaneous word searches can be combined and will only require one pass through the document Bloom filter set. Note that the communication cost is dominated by the false positive rate of document Bloom filters. Section 7 discusses the choice of suitable Bloom filter parameters.

Boolean and Regular Expression Queries. Our construction can also handle "AND" and "OR" boolean queries involving multiple words. We give an example of how to perform an "AND" query. Suppose the user wants all documents containing words x and y. The user first derives the word digests for x and y and gives the server both word digests $W(x)$ and $W(y)$. The server performs the search by simultaneously checking each document Bloom filter using both word digests. "OR" queries can be carried out in a similar manner. The cost and search time for boolean queries grows linearly with the length of the query as compared with a single word query. Note that a boolean query can be completed with a single pass over all documents.

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

Less Revealing Queries. Although searching the entire document set for a word x does not disclose the value of x to the server, the server knows which documents match the codeword for x. This information leakage is the price our scheme and the SSKE schemes pay in exchange for efficiency. Queries restricted to a subset of the entire document set reveal less information to the server. In particular, queries of the form "Does document 5 contain word x" where the server is only given the document 5's codeword for x reveal no information about the other documents.

4.3 Update Algorithms

Adding and Deleting Documents. The setup algorithm in Section 4.1 is used to add new documents. Deletions simply require deleting the document and its Bloom filter from the server.

Altering Document Contents. Altering the contents of an existing document requires assigning the document a new (unique) number and regenerating the document Bloom filter based on the the new number. Using a new number prevents statistical analysis on document content updates.

Cost of Updates. Deleting a document is a constant time operation. Adding or updating a document requires regenerating a new document Bloom filter; Recall that the cost of generating a document Bloom filter is linear in the size of the document.

4.4 Properties

In this section, we mention several interesting properties of our construction.

1. Compressed and Encrypted Data

   Our scheme, being index-based, does not require access to the document contents after the index is built. Hence, we can compress and encrypt the documents with *any* compression and chosen cipher-text secure (IND-CCA2) [6, 13, 24] cipher.

2. Boolean and Limited Regular Expression Queries

   Our scheme can handle these queries efficiently. The algorithms for performing these queries are described in Section 4.2.

3. Variable Length Words

   Our scheme handles variable length words without any modification.

4. Simple Key Management

   As described, our construction requires r keys. Note that the r keys can be generated using a pseudo-random generator with a single secret seed.

5 Security Proofs

We first prove that our schemes use pseudo-random functions securely, and then develop a security model for secure indexes.

5.1 Pseudo-Random Functions

Our construction uses a fixed pseudo-random function with r different keys simultaneously on a single word in a document Bloom filter. The next lemma shows that using a pseudo-random function in this way does not seriously degrade the security of the pseudo-random function. Readers may wish to refer to Section 2.2 to recall the definition of a pseudo-random function. The proofs in this section are straightforward and are omitted.

Lemma 5.1 (Using Pseudo-Random Functions in Parallel). *Let* $f : \{0,1\}^n \times \{0,1\}^s \rightarrow \{0,1\}^m$ *be a* (t, ϵ, q)*-pseudo-random function. Consider the following pseudo-random function* f' *defined as*

$$f'_{k_1,\ldots,k_r}(x) = f_{k_1}(x) \| \ldots \| f_{k_r}(x).$$

Then, f' *is a* $(t, r\epsilon, q)$*-pseudo-random function.*

To prevent statistical analysis, codewords representing a single word are varied across a set of documents. The next lemma proves that the technique used in Section 4.1 transforms the pseudo-random function into a pseudo-random generator. Therefore, correlating codewords representing the same word across a set of documents is infeasible. We first define a pseudo-random generator.

Definition 5.1 (Pseudo-Random Generator). A (t, ϵ)-pseudo-random generator is a function $G : \{0,1\}^n \rightarrow \{0,1\}^m$ where $m \gg n$, and

1. G is efficiently computable by a deterministic algorithm

2. For all t time probabilistic algorithms $\mathcal{A}$,

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

Lemma 5.2. *Let $f : \{0,1\}^n \times \{0,1\}^k \to \{0,1\}^m$ be a (t, ϵ, q)-pseudo-random function. Consider*

$$G(s) = f_s(1) \| \ldots \| f_s(q).$$

Then G is a $(t - q, \epsilon)$-pseudo-random generator.

5.2 Security Model

Intuitively, our security model aims to capture the notion that an adversary learns nothing about the document from the index other than what it already knows from previous query results or other channels. That is, consider a set P (representing a document) containing n words, m of which are public or have been queried upon and are therefore known to an adversary. The challenge to the adversary is, given an index (a Bloom filter using pseudo-random functions in our case) encoding the words in P, deduce at least one of the $n - m$ words.

Suppose the challenger also gives the adversary two distinct sets of $n - m$ words together with an index and tells the adversary that one of these sets is encoded in the index. The new challenge to the adversary is to figure out which set is encoded. It is easy to see that an adversary that wins the previous challenge with non-negligible probability also wins this challenge with the same probability. Therefore, if this challenge is difficult, then the previous challenge — deducing one of the $n - m$ words — must also be difficult. As a result, if the adversary cannot tell which set is encoded in the index with probability greater than $1/2$, then clearly the index reveals nothing about the $n - m$ words. We use this idea of semantic security [6] to formalize the intuition behind this security game between an adversary and challenger.

An index that satisfies our security model provides very strong security in the following sense — Consider a document P containing n words, of which an adversary already knows m words and wants information about the $n - m$ unknown words. We call the set of $n - m$ unknown words W. Even when the adversary is given plain text access to all other documents except on W, and is also allowed to make arbitrary queries of its choice on any word except on those in W, the adversary still cannot obtain any information about any word in W from P's index.

Definition 5.2 (Symmetric Set Difference). The *symmetric difference* of two sets A and B is

$$A \bowtie B = (A \cup B) - (A \cap B).$$

Semantic Security Against Adaptive Chosen Keyword Attack (SS-CKA) Game.

Setup. The challenger $\mathcal{C}$ creates a set S of q words and gives this to the adversary $\mathcal{A}$. From S, $\mathcal{A}$ chooses an arbitrary number of subsets.[2] This collection of subsets is called S^*. Next, $\mathcal{A}$ gives S^* to $\mathcal{C}$. For each subset in S^*, $\mathcal{C}$ encodes its contents into an index using some function f. Finally, $\mathcal{C}$ sends all indexes with their associated subsets to $\mathcal{A}$.

[2] S is the set of all the unique words in a collection of documents and each of the subsets represents a document.

Queries. At any point so far, $\mathcal{A}$ is allowed to query $\mathcal{C}$ on a word x and receive $f(x)$. In addition, $\mathcal{A}$ has access to a function Q that given $f(x)$ and an index I for a subset from S, outputs if I contains $f(x)$. That is,

$$Q(I, f(x)) = \begin{cases} 0 & \text{if } f(x) \notin I \\ 1 & \text{if } f(x) \in I. \end{cases}$$

Challenge. Upon receiving the indexes from $\mathcal{C}$, $\mathcal{A}$ makes some (or no) queries and then decides on a challenge. $\mathcal{A}$ picks a subset from S^*, call it P, and generates another subset Q from S such that that both P and Q are non-empty, of equal size, and $P \bowtie Q$ is non-empty. Furthermore, $\mathcal{A}$ must not have queried $\mathcal{C}$ on any word in $P \bowtie Q$.[3] Next, $\mathcal{A}$ gives P and Q to $\mathcal{C}$ who then randomly picks either P or Q. We refer to $\mathcal{C}$'s choice as V. For each word $x \in V$, $\mathcal{C}$ encodes x using the function f and stores $f(x)$ in an index D. Lastly, $\mathcal{C}$ sends D to $\mathcal{A}$. The challenge for $\mathcal{A}$ is to, given D, guess if $V = P$ or Q.[4] Furthermore, after the challenge has been given, $\mathcal{A}$ is not allowed to query $\mathcal{C}$ on all words $x \in P \bowtie Q$.

Response. After running for some time and making more queries, $\mathcal{A}$ makes its guess. Let g be $\mathcal{A}$'s guess. $\mathcal{A}$ wins if its guess is correct. $\mathcal{A}$'s advantage, $\mathtt{Adv}_{\mathcal{A}}$, is defined as

$$\mathtt{Adv}_{\mathcal{A}} = |\Pr[g = V] - 1/2|,$$

where the probabilities are over $\mathcal{A}$ and $\mathcal{C}$'s coin tosses.

Remark. The adaptive chosen keyword model gives the adversary more power than would normally be expected. In fact, we even allow the adversary to choose the contents of all the documents, and to choose the document that it wishes to be challenged on.

Definition 5.3 (SS-CKA Index). An adversary $\mathcal{A}$ (t, ϵ, q)-breaks a data structure in the SS-CKA model if $\mathtt{Adv}_{\mathcal{A}}$ is at least ϵ after at most t time and q queries to challenger. We say that D is a (t, ϵ, q)-*SS-CKA index* if no adversary can (t, ϵ, q)-break it.

The next theorem shows that Bloom filters using pseudo-random functions are SS-CKA indexes.

Theorem 5.3. *If the Bloom filter construction described in Section 4.1 uses a (t, ϵ, q)-pseudo-random function $f : \{0,1\}^n \times \{0,1\}^s \to \{0,1\}^m$, then the construction is a (t, ϵ, q)-SS-CKA index.*

Proof. We prove the theorem by contradiction. Suppose the Bloom filter using the pseudo-random function f is not a (t, ϵ, q)-SS-CKA data structure but f is a (t, ϵ, q)-pseudo-random function. Then there exists an algorithm $\mathcal{A}$ that (t, ϵ, q)-breaks the Bloom filter using f in the SS-CKA model. Using $\mathcal{A}$, we build another algorithm $\mathcal{C}$ that is a distinguisher for f. Algorithm $\mathcal{C}$ simulates the challenger and interacts with $\mathcal{A}$ as follows.

Setup. $\mathcal{C}$ first picks a set S of q strings from $\{0,1\}^n$ uniformly at random. Next, $\mathcal{C}$ sends S to $\mathcal{A}$ and receives a number of subsets from $\mathcal{A}$. Call this collection of subsets S^*. $\mathcal{C}$ assigns each subset an integer id, and makes the necessary queries to the oracle for f, $\mathcal{O}_f$, to create a Bloom filter for each subset according to Section 4.1. $\mathcal{C}$ stores all results of the oracle queries. Once the setup is complete, $\mathcal{C}$ returns all Bloom filters and associated subsets to $\mathcal{A}$.

[3] Suppose P and Q contain n words and share m words in common. The m shared words represent the information $\mathcal{A}$ already knows and $\mathcal{A}$'s job is to gain some information about the set difference of $n - m$ words.

[4] Note that $\mathcal{A}$ always wins the challenge if the encoding of P or Q in the index is deterministic and never changes on updates — $\mathcal{A}$ simply compares D to the index for P received before the challenge.

Queries. At any point, $\mathcal{A}$ can query $\mathcal{C}$ on words of its choosing. On $\mathcal{A}$'s query for a word x, $\mathcal{C}$ checks if it had previously queried $\mathcal{O}_f$ on x. If it did, $\mathcal{C}$ returns the stored response to $\mathcal{A}$. Otherwise, $\mathcal{C}$ queries $\mathcal{O}_f$ on x, stores the oracle's response, and returns the response to $\mathcal{A}$. $\mathcal{A}$ can use the responses with function Q to determine if a Bloom filter contains queried words.

Challenge. At some point after receiving the Bloom filters, $\mathcal{A}$ picks a challenge subset and returns two subsets of S, called P and Q. By definition, a legal challenge is when P and Q are non-empty, of equal size, $P \bowtie Q$ (the set difference) is non-empty, and $\mathcal{A}$ must not have queried $\mathcal{C}$ on any word in $P \bowtie Q$. After receiving P and Q, $\mathcal{C}$ picks either P or Q uniformly at random. Let V be $\mathcal{C}$'s choice. $\mathcal{C}$ then makes the necessary queries to $\mathcal{O}_f$ and creates a document Bloom filter for V with a new unique integer id. Finally, $\mathcal{C}$ gives the Bloom filter for V to $\mathcal{A}$. From this point on, $\mathcal{A}$ only queries $\mathcal{C}$ on all words x such that $x \notin P \bowtie Q$.

Response. $\mathcal{A}$ queries $\mathcal{C}$ on more words and takes up to time t before it produces a bit g, where $g = 0$ indicates $\mathcal{A}$ guesses that $V = P$ and vice versa. If $\mathcal{A}$ outputs the right answer, then $\mathcal{C}$ outputs that f is a pseudo-random function. Otherwise, $\mathcal{C}$ outputs that f is random.

Analysis. $\mathcal{C}$ takes at most t time because $\mathcal{A}$ takes at most t time. Furthermore, $\mathcal{C}$ makes at most q queries to $\mathcal{O}_f$ because there are only q strings in S. Since $\mathcal{A}$ breaks the Bloom filter using the pseudo-random function f with advantage at least ϵ, we see that $\mathcal{A}$ guesses V correctly with probability at least $1/2 + \epsilon$ if f is a pseudo-random function.

Claim 1. If f is a random function, then $\mathcal{A}$ guesses V correctly with at most probability $1/2$.

Before proving claim 1, we explore the consequences of its veracity. If claim 1 is true, then $\mathcal{C}$ is a distinguisher for f with advantage at least ϵ. This implication contradicts the initial hypothesis that f is a (t, ϵ, q)-pseudo-random function. Therefore, if claim 1 is true, then we have proved that the Bloom filter construction using f is a (t, ϵ, q)-SS-CKA index. We now prove claim 1.

Proof of Claim 1. Recall that Lemma 5.2 tells us that it is infeasible for an adversary to correlate codewords representing the same word across all document Bloom filters in our construction when f is a pseudo-random function. Extending the lemma to random functions, we see that it is not only infeasible but impossible for an adversary to correlate codewords across document Bloom filters. From the extended lemma, together with the restriction on the choice of sets P and Q and the restriction on $\mathcal{A}$'s queries after committing to the challenge, we see that the adversary gains *zero* information about $P \bowtie Q$ (and the challenge subset V) from the other subsets in S^* and their associated Bloom filters. Therefore, we only need to consider the challenge subsets.

Without loss of generality, assume that $P \bowtie Q$ contains only two words, x and y where $x \in P$ and $y \in Q$. Note that $\mathcal{A}$ can derive $f(z)$ where $z = x$ or y from D because there are only two words in $P \bowtie Q$. Suppose $\mathcal{A}$ guesses V correctly with probability $1/2 + \delta$ where $\delta > 0$. Then, $\mathcal{A}$, given $f(z)$, can determine if $z = x$ or y with advantage δ. We have shown that $\mathcal{A}$ is a distinguisher with advantage δ for a random function f, which is impossible. Therefore, $\mathcal{A}$ guesses V correctly with probability at most $1/2$. □

Insecurity of Other Index Constructions. Both the keyword hash table index and the Bloom filter index using pseudo-random functions in the standard way are not SS-CKA indexes. In particular, claim 1 in the proof above is false for these two indexes.

In Practice. Recall that a document Bloom filter stores codewords representing the words contained in the document. These codewords are generated using a pseudo-random function with a set of keys. We assume that an adversary has no access to pseudo-random function keys because they are secret and kept securely by the user. When the keys are secret, an adversary cannot deduce the actual search word from the query codeword, and cannot generate codewords without the aid of the user. Hence, it is hard for an adversary to perform adaptive searches on words of its choice. In most situations, an adversary can only passively observe word searches. Furthermore, documents are usually encrypted with a chosen cipher-text secure (IND-CCA2) [6, 13, 24] cipher, hence denying document content access to an adversary.

Corollaries. The results in this section show that the Bloom filter using pseudo-random functions can be used as to test set membership without revealing the set elements. This same property implies that our Bloom filter construction can also be using for accumulated hashing [7, 23]. A separate observation is that Bloom filters are history independent data structures [22].

6 Extensions

In this section, we describe three useful extensions.

6.1 Heuristically Increasing Security

We describe a technique that makes it harder for the server to figure out when a word is searched for more than once.

Technique. When searching for a word y, the user computes $f_1(y), \ldots, f_r(y)$ in the normal way but instead of sending all r outputs to the server, she sends only $r/2$ randomly chosen outputs. Note that the document Bloom filter is still built using all r outputs.

Benefits. The advantage of this technique is that it is harder for an eavesdropper to determine when the user is searching for a word multiple times. On the other hand, since the server has only half of pseudo-random function outputs, the document Bloom filters register more false positives. A higher rate of false positives increases the communication overhead but more false positives hides the documents that actually contain the keyword. We can choose appropriate Bloom filter parameters to give the desired information hiding rate versus communication overhead. Section 7 discusses the right choice of Bloom filter parameters.

Why Only Heuristic? In certain situations, the server can determine which truncated word digests 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 (nonexistent words). The number of documents returned by a query for a nonexistent 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 nonexistent word query returns all n documents is bounded by the Chebyshev inequality

n	$m = \sqrt{n}$	$(1 - \frac{1}{9\sqrt{n}})^{m-2}$
64	8.00	0.920
128	11.31	0.912
256	16.00	0.907
512	22.63	0.903
1024	32.00	0.901
16384	128.00	0.896
131072	362.04	0.895
1048576	1024.00	0.895
67108864	8192.00	0.895

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

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 $m = \sqrt{n}$. As we can see from the table, 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, the server can, with high probability, determine which are the two queries for x by observing which queries return the entire document set.

Still Useful in Practice. The example given above is a contrived "bad" case. In many common usage scenarios, it is hard (or impossible) for the server to perform similar kinds of analysis on the query patterns. Therefore, it is still useful to implement this modification for real applications.

6.2 Locating Words Within Documents

Instead of using a Bloom filter for every document, we can divide documents into chunks and use a Bloom filter for each chunk. With this modification, we can locate words with chunk size granularity within a document.

6.3 Occurrence Search

We show how to handle queries such as "find all documents where "foo" occurs at least twice". When creating codewords to insert into the Bloom filter, each word is prefixed with the order of its occurrence relative to previous occurrences of the same word in the document. Searching for documents where "foo" occurs twice involves creating a word digest for "(2‖foo)" and handing it to the server. A disadvantage is that this technique increases the number of unique words, which leads to worse Bloom filter parameters. We discuss choosing suitable Bloom filter parameters in Section 7.

6.4 Searching for Infrequent Words Efficiently

If preventing statistical analysis attacks (described in Section 4) is not required, then we can organize document Bloom filters into a binary tree. This tree is used to efficiently search for uncommon or nonexistent words. Appendix A contains a complete description of this extension.

7 Choosing Suitable Bloom Filter Parameters

In this section, we discuss the choice of suitable Bloom filter parameters.

False Positive Rate. Increasing the Bloom filter false positive rate causes the server to return more irrelevant documents in a word search. But the user can easily disregard irrelevant documents by mechanically scanning them for the word once they are retrieved and decrypted. Hence, increasing the false positive rate merely increases the communication overhead between the user and the server and does not affect the correctness of the result. On the other hand, as discussed in Section 6.1, high false positive rates deter statistical analysis. For most applications, a false positive rate of 1 in 100 or 1 in 1000 is sufficient.

Number of Pseudo-Random Function Keys. As shown in Section 2.1, the optimal number of hash functions, r, is given by the equation $r = (\ln 2)(m/n)$ where m is the size of the Bloom filter array and n is the number of words. Selecting r in this manner gives a false positive rate of $(1/2)^r$. In our application, the number of hash functions r is the number of pseudo-random function keys.

Array Size. We fix n by choosing an upper bound on the number of unique words in a document set. Note that we only need to take into account the set of unique words in the document set and not the set of all possible words in the universe. With r and n determined, we are left with choosing the array size m. Rearranging the equation, we see m is chosen such that $m = nr/\ln 2$.

How to Choose Bloom Filter Parameters. We now explicitly state the procedure to choose Bloom filter parameters. First choose the desired false positive rate, which in turn gives the number of hash functions r. Next, 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. With the expanded unique word set size n, compute the array size $m = nr/\ln 2$.

Word Sets are Usually Small. Observe that the set of possible words is potentially very large. For example, the second edition of the Oxford English Dictionary [25] contains 291500 entries, of which 47100 entries are obsolete words. Hence, the word set is approximately 2^{18}. In this case, for a false positive rate of 1 in 1024 (2^{10}), $m \approx 10 \cdot 2^{18}/\ln 2 \approx 462$ kB. If the documents are small, storing a 462 kB Bloom filter for each of them is inefficient. Nevertheless, most English document sets do not have a large unique word set. We support this statement with two examples.

1. Consider the Calgary Corpus [2, 3] which is the de facto standard for lossless compression evaluation. The standard corpus consists of 14 files containing English text, and is 3.1 megabytes in size. The corpus contains a total of 437501 English words and letters, of which only 22425 are unique. In this situation, a false positive rate of 1 in 1024 (2^{10}) can be obtained by using 10 hash functions and a Bloom filter array of about 58 kB in size.

2. Also consider the news archives for the `IPSEC` working group [1]. The archives from January to mid May 2003 consists of 1245 email messages with a total size of 4.5 megabytes. Excluding email headers and attachments, the email bodies of this archive contains 463869 English words and letters, of which only 10014 are unique. In this situation, a false positive rate of 1 in

1024 (2^{10}) can be obtained by using 10 hash functions and a document Bloom filter array of about 29 kB in size.

Even if the word sets are large (resulting in large Bloom filters), we have effective methods for reducing the Bloom filter size. We describe these techniques in the next section.

8 Reducing Bloom Filter Array Size

In this section, we describe three techniques for reducing Bloom filter sizes.

8.1 Alternative Array Data Structures

In many applications, although the number of possible words in a set of many documents is large, each document is small and does not contain many unique words. Referring to the example given in the previous section on the `IPSEC` news archives, the set of possible words in an email is the total set of unique words (463869) but each email contains only a small number of unique words.

Since the number of unique words in each email is small, using an array is wasteful because most of the array entries are 0. Instead, we can simply keep track of the positions which contain 1's. For example, when the Bloom filter array is 2^{25} bits or 4 megabytes, every position in the array can be represented using $25 \approx 2^5$ bits. Hence, this technique is effective for document Bloom filters whose arrays have less than 2^{20} entries marked with 1's. For example, consider a large document containing 5000 unique words.[5] Using this technique, the Bloom filter size for this document is only $5000 * 25 = 125000$ bits or approximately 15 kilobytes, representing a 99.63% reduction in size from a 4 megabyte array.

This technique is most effective on sparse (or dense) arrays where entries are mostly 0's (or 1's). Also note that an array consisting mostly of 0's (or 1's) can be efficiently compressed. Compressing the array, however, causes the server to do more work during a word search.

Finally, observe that the large unique word set with small documents scenario where this technique is most effective frequently occurs in real applications. Therefore, this technique is applicable to a large number of real world situations.

8.2 Not Indexing All Words

In some situations, users might not need to search on all the content in a set of documents. For example, consider a set of documents, each labelled with a unique serial number and a set of attributes such as the date created, authors, and the subject of the document. In this case, it might suffice to index only the serial number and attributes. The number of unique words in this reduced set is smaller than if the entire document was indexed. Alternatively, a user could choose a list of words that he wishes to keep track of in his document set.

8.3 Compressed Bloom Filters

Compressed Bloom filters [20] were developed by Mitzenmacher to minimize the size of the Bloom filter. Experiments in the paper show that using compressed Bloom filters over standard Bloom filters gives a reduction of 5-15% in systems of reasonable size.

[5] In comparison, this paper (including undisplayed comments) only contains 1371 unique words out of 8700 words.

9 Related Work

Searchable Symmetric Key Encryption (SSKE). The most efficient practical solution to date is by Song, Wagner, and Perrig (SWP) [26]. Their paper concentrates on developing a SSKE scheme that allows a user, given a trapdoor, to test if the cipher text contains a word. To encrypt a document, the document is divided into blocks and each block is encrypted using the SSKE scheme. We call this method *SSKE Linear Search*. Given a trapdoor for a particular word, SSKE lets a user search for the word by linearly scanning and testing every SSKE encrypted block in the document. The disadvantages of this search method are twofold: First, SSKE uses fixed block sizes and needs some inelegant modifications to handle variable length words. Second, both the search time and computation by the server on a query is linear in the number of blocks in the document.

The SWP paper also describes an index solution which we call *SSKE Word Index*. SWP did not explicitly describe their index construction but it is clear from their description that they are thinking of a hash table where words hash into buckets, and each bucket contains pointers to documents with the word. The main disadvantage of their index construction is that index updates are insecure. For example, consider the case when a document is added to the document set. For every word in the new document, a pointer to that document is added to the appropriate word list in the index. By observing bucket length changes in the index, the server learns the set of coded words in the document and the size of the set. Note that this attack works even when the bucket entries are encrypted. The same situation occurs when documents are updated; the bucket length changes reveal the extent of the document update. To deter such statistical analysis, SWP suggest performing queries and updates in batches. Query batching, however, does not work for queries that depend on the results of previous queries. Furthermore, this solution only provides heuristic security and can be defeated by sampling more queries and updates. We describe a similar heuristic improvement for our scheme and an attack in Section 6.1. As a result, Song et. al. state that their index solution is suitable for "mostly read-only data".

Searchable Public Key Encryption (SPKE). Boneh et. al. developed the public key equivalent of SSKE for searching on encrypted data [9]. In their scheme, any user can create word digests by using a public key; a trapdoor generated by the private key is required to search documents for a particular word. Note that one trapdoor is required for one word. Their scheme provides similar security as our construction. Due to the expense of public key computations, their scheme is only suitable for encrypting and searching over a small number of words per document.

Private Information Retrieval (PIR). A closely related concept is Private Information Retrieval (PIR) [12]. PIR schemes allow users to retrieve information from a database such that the database server learns nothing about which records were read. PIR schemes aim to minimize communication complexity and provide either information theoretic [12] or computational security [10]. Unfortunately, the schemes proposed so far are impractical and either require large communication complexity, multiple servers, large computation cost by the server, no data confidentiality, or no controlled searching. Some work [10, 11, 15, 18] has been done to mitigate some of the limitations, but they are still impractical. Furthermore, PIR schemes only handle reads.

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 for suggesting the Calgary Corpus and IETF mail archives as sources of data, and also for comments on various aspects of the paper. Also thanks to Burt Kaliski for pointing out the connection to accumulated hashing. Thanks to Rajeev Motwani for pointing out a oddity in the original reduction whose investigation led to the improved model.

References

[1] IPSEC working group. `http://www.ietf.org/html.charters/ipsec-charter.html`.

[2] T. Bell, I. Witten, and J. Cleary. *Calgary Corpus*. University of Calgary, `http://www.data-compression.info/Corpora/CalgaryCorpus/calgarycorpus.htm`, 1989.

[3] T. Bell, I. Witten, and J. Cleary. Modeling for text compression. *ACM Computing Surveys*, 21(4):557–591, Dec 1989.

[4] 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.

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

[6] 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.

[7] 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.

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

[9] D. Boneh, G. D. Crescenzo, R. Ostrovsky, and G. Persiano. Searchable public key encryption. Cryptology ePrint Archive, Report 2003/195, Sep 2003. `http://eprint.iacr.org/2003/195/`.

[10] C. Cachin, S. Micali, and M. Stadler. Computationally private information retrieval with polylogarithmic communication. In J. Stern, editor, *Proceedings of Eurocrypt 1999*, volume 1592 of *Lecture Notes in Computer Science*, pages 402–414. Springer-Verlag, May 1999.

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

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

[13] D. Dolev, C. Dwork, and M. Naor. Nonmalleable cryptography. *SIAM Journal on Computing*, 30(2):391–437, 2000.

[14] 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.

[15] Y. Gertner, Y. Ishai, E. Kushilevitz, and T. Malkin. Protecting data privacy in private information retrieval schemes. *Journal of Computer and System Sciences*, 60(3):592–629, Jun 2000.

[16] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing Remote Untrusted Storage. In *Proceedings of the 10th Network and Distributed System Security (NDSS) Symposium*, pages 131–145. Internet Society (ISOC), Feb 2003.

[17] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*, pages 570–573. Addison-Wesley, 2nd edition, 1998.

[18] 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.

[19] D. Mazières and D. Shasha. Building secure file systems out of byzantine storage. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, pages 108–117. ACM, Jul 2002.

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

[21] J. Mullin. Optimal semijoins for distributed database systems. *IEEE Transactions on Software Engineering*, 16(5):558–560, May 1990.

[22] 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.

[23] 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.

[24] 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.

[25] J. Simpson and E. Weiner, editors. *Oxford English Dictionary.* Oxford University Press, 2nd edition, 1989.

[26] 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.

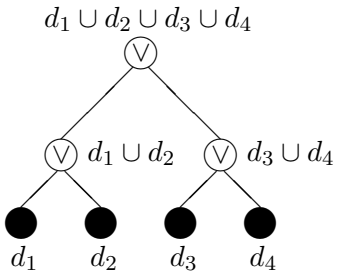

Figure 2: A Bloom filter tree for 4 documents.

A Searching for Infrequent Words Efficiently

If preventing statistical analysis attacks (described in Section 4) is not required, then we can organize document Bloom filters into a binary tree; This tree is used to efficiently search for uncommon or nonexistent words.

Building a Bloom Filter Tree. In this modified scheme, document Bloom filters contain the word digest $f_1(x), \ldots, f_r(x)$ instead of $f_{f_1(x)}(j), \ldots, f_{f_r(x)}(j)$ as described in the original scheme. Note that document numbers are no longer needed.

Observe that a bitwise OR operation between the arrays of two Bloom filters 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 Bloom filter 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 Bloom filters for each pair are the leaves of the Bloom filter tree.

2. Compute the bitwise OR of each pair of document Bloom filters and assign the resulting 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 Bloom filter at the root contains the set of unique words in the entire document set.

Figure 2 shows an example of a Bloom filter tree for a set of 4 documents. The Bloom filter tree can be built by the server or by the user and later transferred to the server. Note that building the tree is cheap because only about $2n$ 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 Bloom filter at node i contains the word digest for x.

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 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 Bloom filter tree allows us to rule out nonexistent 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 checks where v is the number of documents containing x. This method of searching is better than the linear search 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 Bloom filters 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.

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

Updates on a Bloom Filter Tree. Adding (or deleting) a document 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

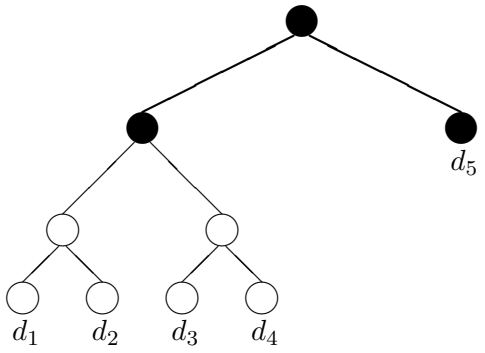

Figure 3: Searching a 5 document Bloom filter tree.

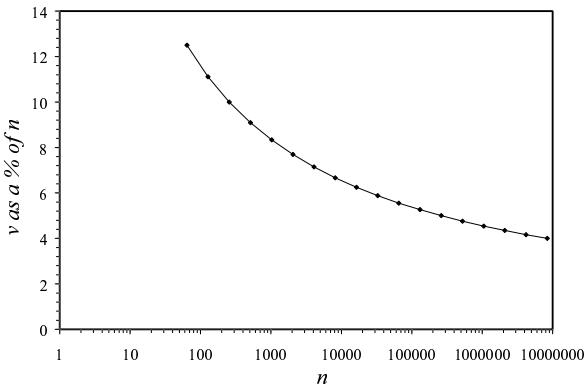

Figure 4: $v = n/2 \log n$ as a percentage of n versus n.

Bloom filters" [14] at all non-leaf nodes.

Cost of Updates. Adding a new document or deleted a document or changing document contents now incurs an additional cost of updating the Bloom filter tree. Note that changes to the Bloom filter tree on an update only affects 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.