|  |
| --- |
|  |
| Encrypted Search |
| Enabling standard information retrieval techniques while preserving confidentiality against an adversary with access to hidden query histograms and raw contents of secure indexes |

|  |
| --- |
| Alexander R. Towell  3-7-2014 |

1. Contents

[2 Motivation for Encrypted Search 5](#_Toc392880672)

[2.1 Advantages of cloud storage 5](#_Toc392880673)

[2.2 Disadvantage of cloud storage: loss of confidentiality 5](#_Toc392880674)

[2.3 Inefficient solution to confidentiality 5](#_Toc392880675)

[2.4 Efficient solution to confidentiality: encrypted search 6](#_Toc392880676)

[3 Confidentiality 6](#_Toc392880677)

[3.1 Compression 7](#_Toc392880678)

[3.2 Obfuscation 7](#_Toc392880679)

[3.3 Encryption 7](#_Toc392880680)

[3.3.1 Symmetric encryption 8](#_Toc392880681)

[3.3.2 Public-key encryption (asymmetric encryption) 8](#_Toc392880682)

[3.3.3 Trapdoors -- one-way (cryptographic) hash functions 8](#_Toc392880683)

[4 Information leaks 9](#_Toc392880684)

[4.1 Document confidentiality 9](#_Toc392880685)

[4.2 Query privacy 10](#_Toc392880686)

[4.3 Access patterns 10](#_Toc392880687)

[4.4 Primary goal of *Encrypted Search* 10](#_Toc392880688)

[5 Online and offline searching 11](#_Toc392880689)

[5.1 Online searching 11](#_Toc392880690)

[5.2 Offline (index) searching 12](#_Toc392880691)

[5.2.1 Inverted index 12](#_Toc392880692)

[5.2.2 Bloom Filter (Signature File) 12](#_Toc392880693)

[6 Mapping queries to documents 13](#_Toc392880694)

[6.1 Definition of a query 13](#_Toc392880695)

[6.2 Boolean search 14](#_Toc392880696)

[6.2.1 Extensions 14](#_Toc392880697)

[6.2.2 Exact phrase matching (word n-grams) 16](#_Toc392880698)

[6.3 Rank-ordered search -- degrees of relevancy 17](#_Toc392880699)

[6.3.1 Term importance weighting 17](#_Toc392880700)

[6.3.2 Term proximity weighting 18](#_Toc392880701)

[6.3.3 Semantic search 19](#_Toc392880702)

[7 Research objectives 19](#_Toc392880703)

[7.1 Secure indexes 20](#_Toc392880704)

[7.1.1 Overview 20](#_Toc392880705)

[7.1.2 Interfaces 24](#_Toc392880706)

[7.1.3 Perfect filter 24](#_Toc392880707)

[7.1.4 Bloom filter secure index 27](#_Toc392880708)

[7.1.5 Bloom filter secure index-block (BSIB) 27](#_Toc392880709)

[7.1.6 Perfect filter secure index (PSI) 27](#_Toc392880710)

[7.1.7 PsiFreq (PSIF) 29](#_Toc392880711)

[7.1.8 PsiPost (PSIP) 29](#_Toc392880712)

[7.1.9 PsiBlock (PSIB) 30](#_Toc392880713)

[7.1.10 PsiMin (PSIM) 31](#_Toc392880714)

[7.2 Definition of a hidden query 33](#_Toc392880715)

[7.3 Relevancy metrics 34](#_Toc392880716)

[7.3.1 Precision and recall 34](#_Toc392880717)

[7.3.2 Estimating relevancy with term weighting 34](#_Toc392880718)

[7.3.3 Relevancy of proximity 36](#_Toc392880719)

[7.3.4 Measuring performance of relevancy scores 38](#_Toc392880720)

[7.4 Information leaks 39](#_Toc392880721)

[7.4.1 Simulating an attack using query history 39](#_Toc392880722)

[7.4.2 Simulating an attacker reconstructing documents from secure index information 42](#_Toc392880723)

[7.4.3 Access pattern leaks 46](#_Toc392880724)

[8 Experiments 46](#_Toc392880725)

[8.1 Inputs 46](#_Toc392880726)

[8.2 Outputs 47](#_Toc392880727)

[8.3 Platforms 47](#_Toc392880728)

[8.4 Global Parameters 48](#_Toc392880729)

[8.5 Results 49](#_Toc392880730)

[8.5.1 BM25 MAP “Page One” Results 49](#_Toc392880731)

[8.5.2 MinDist\* “Page One” Results 51](#_Toc392880732)

[8.5.3 Attack Simulation: Obfuscations vs Accuracy 53](#_Toc392880733)

[8.5.4 Attack Simulation: Secrets vs Accuracy 55](#_Toc392880734)

[8.5.5 Attack Simulation: History Samples vs Accuracy 56](#_Toc392880735)

[8.5.6 Secrets vs Compression Ratio, Build Time, and Load Time 58](#_Toc392880736)

[8.5.7 False Positives vs BM25 MAP and Precision 60](#_Toc392880737)

[8.5.8 Obfuscations vs BM25 62](#_Toc392880738)

[8.5.9 Obfuscations vs MinDist\* 65](#_Toc392880739)

[8.5.10 Pages vs Secure Index Size 67](#_Toc392880740)

[8.5.11 Blocks vs Secure Index Size 69](#_Toc392880741)

[8.5.12 Documents (per corpus) vs Corpus Secure Index Size 71](#_Toc392880742)

[8.5.13 Pages vs Build Time 72](#_Toc392880743)

[8.5.14 Documents (per corpus) vs Build Time 73](#_Toc392880744)

[8.5.15 Pages vs Load Time 74](#_Toc392880745)

[8.5.16 Documents vs Load Time 76](#_Toc392880746)

[8.5.17 Pages vs MinDist\* Lag Time 77](#_Toc392880747)

[8.5.18 Simulation: Location Uncertainty vs Location Error 79](#_Toc392880748)

[8.5.19 Simulation: Location Uncertainty vs Minimum Pairwise Distance Error 80](#_Toc392880749)

[8.5.20 Location Uncertainty vs MinDist\* MAP 81](#_Toc392880750)

[8.5.21 Pages vs BM25 Lag Time 83](#_Toc392880751)

[8.5.22 Location Uncertainty vs BM25 MAP 85](#_Toc392880752)

[8.5.23 Pages vs Boolean Lag Time 87](#_Toc392880753)

[8.5.24 Compression Ratio (Secure Index Size to Document Size) vs MinDist\* MAP 89](#_Toc392880754)

[9 Future work 90](#_Toc392880755)

[9.1 experimental simulation of an adversary 90](#_Toc392880756)

[9.2 Compound set-theoretic queries 90](#_Toc392880757)

[9.2.1 Compound query grammar 90](#_Toc392880758)

[9.3 Fuzzy set-theoretic search 91](#_Toc392880759)

[9.4 Letter n-grams and word n-grams 92](#_Toc392880760)

[9.5 Wild-card searching 92](#_Toc392880761)

[9.6 Approximate searching and error tolerance 92](#_Toc392880762)

[9.7 Boolean proximity searching 93](#_Toc392880763)

[9.8 Semantic search 93](#_Toc392880764)

[9.9 Topic searching 93](#_Toc392880765)

[9.10 Optimization: caching results 94](#_Toc392880766)

[10 References 94](#_Toc392880767)

# Motivation for Encrypted Search

## Advantages of cloud storage

Organizations are eager to take advantage of cloud storage for its many advantages.

* Reliable—storage management is delegated to expertise of cloud storage provider (CSP).
* Scalable—as storage needs change, pay more or less as needed.
* Cost-effective—cloud storage providers are efficient (division of labor).
* Accessible—storage can be accessed anytime and anywhere.
* Sharable—every resource (e.g., directories, files) has a URL.

## Disadvantage of cloud storage: loss of confidentiality

A significant disadvantage to cloud storage is loss of control over confidentiality. An organization loses control over an unencrypted document’s confidentiality when it is hosted in the cloud.

In [1], one of the earlier papers presented on Encrypted Search, the author observes that many individuals and organizations wish to exploit cloud storage services, but do not trust the CSP with their sensitive data. That is, organizations trust the CSP with storage logistics but they do not trust the CSP with their need for confidentiality.

## Inefficient solution to confidentiality

The naïve solution to regaining control over confidentiality of cloud-hosted documents is achieved using encryption. Before a document is uploaded into the cloud, it is encrypted. Subsequently, to access this document, clients download it to a trusted machine and decrypt it.

Often, the documents of interest are not known in advance. Consequently, the ability to perform searches over a collection of documents is needed. In the naïve solution, this entails the following sequence of actions:

1. Download the collection of encrypted documents to a trusted machine.
2. Decrypt the encrypted documents.
3. Search through the decrypted documents using any available search facility on the local machine.

This approach breaks down if an organization has a large collection of sensitive documents: being required to download a large collection of encrypted documents before being able to search them can be time consuming and costly in terms of transmission costs (i.e., downloading a large corpus) and energy costs (i.e., decryption is a computationally demanding).

The larger the collection of sensitive documents becomes, the less efficient the naïve solution is. This inefficiency is especially evident on resource-constrained machines, e.g., smartphones with limited bandwidth and power.

## Efficient solution to confidentiality: encrypted search

What is sought is some way to allow the CSP to search the encrypted documents on behalf of clients, and returning only those documents relevant to client queries. Furthermore, this should be done without revealing the contents of documents (data confidentiality) nor the contents of client queries (query privacy). In other words, the CSP should be able to perform *oblivious* searches on behalf of authorized users. Finally, the CSP should not be able to initiate meaningful searches except on behalf of authorized users.

The ability to search over a collection of encrypted documents without needing to decrypt them first is known as Encrypted Search*[[1]](#footnote-1)*. In light of the advantages of cloud storage, Encrypted Search has recently gained a lot of traction in the research community.

Many solutions to this problem have been proposed. Sections 1-6 describe much of the existing work in Encrypted Search with attention paid to the strengths and weaknesses of various proposals.

# Confidentiality

The first issue to address is confidentiality and its implementation. That is, the techniques employed to prevent disclosing information to unauthorized parties, like a server hosting the confidential documents. We consider three primary approaches: compression, obfuscation, and encryption.

|  |  |  |
| --- | --- | --- |
| Methods | Advantages | Disadvantages |
| Compression  [2], [3] | Very space efficient  Fast and easy to implement  Very well understood (Huffman) | To serve as an obfuscator, users must maintain a separate symbol-mapping table[[2]](#footnote-2)  Serves as a substitution cipher; may be broken through cryptanalysis |
| Obfuscation  [4], [5], [6] | Effective against non-skilled attackers | High insider risk  Too weak for cryptographic use |
| Encryption  [7], [8], [9], [10], [11], [1], [12], [12] | Provides strong guarantees on data confidentiality (assuming the key is kept private)  Very well understood (modern cryptography) | Depending on the types of information leaks prevented, certain IR operations problematic, e.g., if query privacy is provided, ranking relevancy of documents can be more challenging  Strong encryption tends to be slow and still subject to information leaks |

## Compression

In [2], the idea of using the theoretically optimal symbol encoder, Huffman [3], is proposed in the context of information retrieval (without consideration given to the unique needs of Encrypted Search), where the symbols consist of words rather than letters. Thus, each word in the document maps to a unique bit string such that the total length of the document is (near) optimally compressed. To obscure the contents of a document, one could substitute the words with Huffman codes to serve as a weak form of encryption (a substitution cipher). That means the symbol table must be kept private to prevent untrusted parties from learning the codewords.

When combined with an inverted index (see section 5.2.1), this is not only space-efficient, but fast also; it only takes time to determine if a word is in a document, where is the size of the vocabulary. Unfortunately, it has far more disadvantages than advantages. First, a local symbol table must be maintained and kept private (this is essentially a large secret key). Second, and more importantly, it is a simple substitution cipher and is thus vulnerable to substitution cipher attacks.

## Obfuscation

The compression section discussed a method to obfuscate by remapping words to Huffman codes. In general, any symbol substitution technique may be used to obfuscate the contents of documents. The primary distinction between obfuscation and compression is obfuscation obscures the contents of documents by mapping symbols in the domain of to other symbols in the domain of [4] [5], whereas compression maps codes in one domain (e.g., character strings) to codes in another domain (e.g., bit strings).

Conceptually, there are two types of obfuscation techniques: encoding-based techniques and index-based techniques. Encoding-based transformations apply general rules to each symbol. For example, remapping the character sequence to by adding to the ASCII code of each character (this is also an example of a simple Caesar substitution cipher). Indexed-based transformations apply unique transformations to inputs, e.g., the one-to-one mapping of *John Smith* to *Person 192353*.

A more general example can be found in automatically generating hash functions; that is, map to .

The symbol remapping instructions—essentially a secret key—must not be disclosed to unauthorized parties. Moreover, these techniques are generally vulnerable to substitution cipher attacks.

## Encryption

All of the previous confidentiality techniques serve the same purpose: convert communicated information into a secret code so that unauthorized people cannot understand it. The more popular and far more secure approach to doing this is through cryptography.

In cryptography, the unencrypted data is called plaintext. To encrypt (convert) plaintext into an unintelligible format, called ciphertext, the plaintext is input into an encryption function along with a secret key. The inverse of the encryption function is the decryption function, which takes the ciphertext and a secret key (either the same key, as in symmetric encryption, or another paired keyed, as in asymmetric encryption), and outputs the plaintext. For each cryptographic key in the keyspace (all possible keys), the encryption function maps a given plaintext to a different and unique ciphertext. Thus, to decrypt the data, one must not only be in possession of the decryption algorithm, but also the secret key.

In designing security, one should assume Kirchhoff’s principle, “only the secrecy of the key provides security”. Specifically, the cryptographic algorithm[[3]](#footnote-3) is presumably already known to untrusted parties. Thus, only the secret key must be kept private.

Without knowing the secret key, an unbroken encryption scheme requires an attacker to do a brute force attack over the entire key space; for a given cipher text , iterate through all keys in the key space, feed the decryption function with the given key and cipher text, and decide on the plausibility of the decoded plaintext . To make brute force attacks intractable, the key space should be extremely large.

### Symmetric encryption

In the context of Encrypted Search, two different types of encryption have been used, symmetric encryption and asymmetric (public-key) encryption. Symmetric encryption uses a single key for both encryption and decryption. Compared to public key encryption, it is less computationally demanding. However, the downside is, a secure channel must be used to communicate the secret key if multiple parties need to be able to use it. The earliest examples of Encrypted Search used symmetric encryption [7] [1].

### Public-key encryption (asymmetric encryption)

Boneh [13] proposed an Encrypted Search scheme based on public-key encryption. Using public key encryption addresses a weakness found in earlier proposals. In symmetric encryption, if wishes to make it so that can search a confidential document, they must first agree on a secret key. The problem with this approach is, how do they agree on a key without revealing it to others? In public-key encryption schemes, no such problem arises: he may use her public key. Thus, only , who has the corresponding private key, may search the confidential document[[4]](#footnote-4).

### Trapdoors -- one-way (cryptographic) hash functions

In a secure index (see offline searching), which is independent of the document it represents, there is no need (nor desire) to be able to reconstruct the document from the information in the index. This particular relaxation offers a more attractive option: do not use a decipherable encryption scheme at all. Rather, use a one-way hash function [14] [15], ideally something that approximates a random oracle, and “filter” [16] the document (plaintext) through it, e.g., insert the one-way hash of each word in the document into the secure index. These one-way hashes are known as trapdoors—easy to compute and very difficult (if not impossible) to invert.

In theory, it is nearly impossible to which terms a document contains simply by looking at the secure index since the trapdoors are practically one-way. Indeed, in many constructions, they are truly one-way (non-invertible), e.g., multiple terms may map to the same cryptographic hash value (collision). In this case, it is impossible to determine with certainty, which terms the document contains.

There are compelling advantages to this approach. First, it is far less computationally demanding; evaluating a one-way hash is far less computationally demanding than executing a decipherable encryption scheme. Second, because one-way hash functions are non-invertible and pre-image resistant[[5]](#footnote-5), even if the secret keys are disclosed to unauthorized users, this does not necessarily compromise the contents of the actual document (except by allowing whatever search facilities the secure index permits).

Note that one or more secret keys, as with encryption, may be used to manage search authorization and to mitigate pre-image attacks (the secrets serve as salts).

# Information leaks

Goh [16] contends that Encrypted Search should ideally reveal no information about the contents of the document unless a secret which is used to construct trapdoors [14] that can be used to search for specific terms in the document)—e.g., is known which allows the secret-holder to search it. The only information that should be revealed through this search operation is whether a given encrypted document is relevant to a query. Thus, even if an untrusted party—like a cloud storage provider—captures an encrypted query, it can neither determine the contents of the query nor the document, affording both data confidentiality and query privacy.

While this is ideal, to enable Encrypted Search in untrusted environments, there are many different subtle ways information can be disclosed—or “leaked”. This remains true even if we assume an unbreakable encryption scheme is being used.

## Document confidentiality

Even if a strong cryptographic scheme is being used, information may still be leaked. Consider the following. For each document in the collection, the words in a given document are passed through a one-way hash and that hash is directly inserted into the index. Since this is a substitution cipher for small blocks (words), it is vulnerable to substitution cipher attacks.

For instance, since it is likely word frequencies in the confidential corpus are similar to other corpora, statistical frequency analysis can be used to construct probable cipher string to plaintext word mappings. For reasons similar to this, Goh [16] argued traditional hash tables are not suitable for use as secure indexes.

## Query privacy

The same argument for data confidentiality also applies to query privacy. In the extreme case, queries may be sent in plaintext. This will be very informative to an adversary; it enables them to construct a model of an encrypted document by submitting many different plaintext queries to it and seeing which are relevant—e.g., in the case of Boolean search, does it contain these keywords? Thus, it is vulnerable to basic dictionary attacks. Indeed, they may be successful at reconstructing close approximations of original documents, especially if phrase searches are supported.

A simple solution is to insert a cryptographic hash of the term, as elaborated elsewhere[[6]](#footnote-6). However, since it is reasonable to assume the hash algorithm will be known to adversaries, they may proceed the same as before. A better solution is to hash the terms concatenated with one or more secret keys. As long as the keys are kept secret, only authorized users may meaningfully submit queries to the secure index.

However, if a cryptographic query for a specific term always looks the same, then an adversary may slowly build up a frequency table of the cryptographic hashes by observing query histories and use this information to mount a substitution cipher attack. For example, using a frequency table of plaintext English words, an adversary may be able to determine an accurate mapping from cryptographic hashes to English words.

A somewhat impractical solution to this problem is found in oblivious RAM [17]. In our research, we will explore other methods.

## Access patterns

To exacerbate matters, even if an Encrypted Search scheme provides robust data confidentiality and query privacy, access patterns may be useful for statistical inference.

A user’s data access patterns constitute a certain kind of information. For instance, what is the distribution of (encrypted) documents the user has retrieved over time? Users may show interest in different documents, in which case clustering algorithms may reveal associations among users.

Another kind of information is revealed by subtler implicit patterns. For example, if a hidden query is followed by another action, like checking stock prices, this correlation may conceivably be used to infer properties about the query and the corresponding documents that are returned in response to it.

To mitigate these subtle forms of information disclosure, Pinkas [17] proposed the use of Oblivious RAM to conceal the history of patterns (observable by the server) associated with a user’s Encrypted Search activities.

## Primary goal of Encrypted Search

The primary goal of Encrypted Search is to provide searching facilities while preventing an adversary from being able to infer anything about confidential documents and private queries except which confidential documents[[7]](#footnote-7) are returned for a given hidden query—and even this information can be minimized.

Most Encrypted Search schemes [16] [1] [13] [18] have this as the primary objective, but only a few solutions [17] considered maintaining this objective in the presence of a determined adversary who has access to user activity histories, e.g., hidden query histories.

# Online and offline searching

| Methods | Advantages | Disadvantages |
| --- | --- | --- |
| Online  [7], [10], [11], [1] | Exact phrase matching is easy—does not increase size like in other solutions  Reasonably space-efficient (if combined with Huffman coding) | Vulnerable to substitution cipher attacks -- encrypted words may be statistically inferred by observing their frequencies and correlations  Sequential search—impractically slow for large-scale use |
| Offline: Inverted index  [8], [2] | Efficiently supports query operations necessary for rank-ordered search heuristics.  Reasonably fast trapdoor lookups—O(log n) for a document with n words (a global index should not be used)  Reasonably space-efficient (if combined with Huffman coding) | Vulnerable to substitution cipher attacks  Potentially vulnerable to preimage attacks |
| Off-line: Bloom filter  [16], [19] | Space efficient (nearly optimal)—can trade accuracy for space-complexity  Reasonably fast trapdoor lookups | k hash functions must be evaluated per trapdoor query  Does not efficiently support query operations necessary for rank-ordered search heuristics. |

## Online searching

Online search performs a sequential search on the document cipher [1] [5] [6] [9]. To be able to perform encrypted searches on such a cipher, a large block cipher may not be used; rather, each term must be encrypted separately to facilitate exact string matches on the encrypted query terms. Thus, this is a simple substitution cipher. As mentioned elsewhere, these may be compromised.

Moreover, as pointed out in [18], proposals based on online searching, in light of their sequential time complexity, are not appropriate except in limited contexts. For example, they may be appropriate if the purpose is to allow an email server to obliviously scan the “subject” field of incoming emails for the keyword “urgent”.

While their disadvantages are many, they do have some advantages. Their primary advantage is related to their simplicity: one simply iterates through all of the terms and applies a cryptographic hash to each. To permit exact phrase searching, an encrypted query phrase need only be iteratively compared to the encrypted terms in the cipher. A comprehensive overview of online searching solutions can be found in [5].

## Offline (index) searching

Offline indexes are data structures that store a representation of the document (or documents) in which rapid, efficient retrieval is facilitated.

Most of the recently proposed Encrypted Search constructions are based on offline indexes [16] [19] like Bloom filters. A comprehensive overview of offline-based solutions can be found in [13].

### Inverted index

In [2], a possible approach to a secure index is given in the form of an inverted index. Previously, the inverted index was discussed in the context of using Huffman codes to serve as an efficient (though insecure) substitution cipher. However, more secure representations are possible.

If words are being stored in the inverted index, then this means an adversary can observe the contents of the document directly and thus no data confidentiality is provided. Moreover, even if encrypted, compressed, or obfuscated transformations of terms are stored in the index, it is still subject to cryptanalysis (e.g., frequency analysis) and pre-image attacks.

The primary advantage of the inverted index is that it is extremely well-understood—it is the most popular index in the field of information retrieval—and thus many algorithms and heuristics have been designed with it in mind. It is also reasonably fast and space-efficient.

### Bloom Filter (Signature File)

A Bloom filter [20] is a probabilistic (approximate) set in which false positives on membership tests are possible. While it does not possess the theoretical optimal space efficiency of bits for a set with a false positive rate and members, it is still reasonably space efficient requiring only bits.

Operationally, it consists of a bit vector of size (all initially set to ) and hash functions. For each member, use the hash functions to map it to (possibly non-unique) indexes in the bit vector, setting each mapped bit to .

To verify an element is a member of the set, check to see if each of its hash positions is set to . On the one hand, If any are , then it is certainly not a member (false negatives are not possible). On the other hand, if all of them are set to , we assume it is a member with a false positive rate. That is, one or more actual members may have caused those bit positions to be set to .

It is straightforward to construct a secure index from a Bloom filter. For each term in the document (words, n-grams, or other searchable terms), insert it into the filter. To prevent unauthorized users from querying the index, do not insert the plaintext terms; rather, insert some transformation of them. Ideally, a trapdoor will be used, which is just a one-way (cryptographic) hash function applied to each term concatenated with one or more secrets, e.g., .

To harden the Bloom filter from cryptanalysis, the same term in separate documents ought to map to different index positions in the Bloom filter. In [16], it is recommended that the document id be appended to the trapdoors during the construction of the secure index. For example*,* . Likewise, during the construction of hidden queries (trapdoors), the user must apply the same transformations. Since is unknown to unauthorized parties, like the server, they are unable to meaningfully query the secure index.

#### Problems with the Bloom filter

However, there is a notable problem with secure indexes based on Bloom filters: they need to evaluate hash functions per trapdoor, where . For large corpora consisting of documents, this could be a significant drawback, requiring hashes per trapdoor.

Another problem with them, as previously described, is they do not efficiently support multiplicities (multi-sets) nor do they efficiently support other types of queries, e.g., where is a member located? This complicates scoring functions like term weighting or proximity weighting.

# Mapping queries to documents

The method in which a query is mapped to (or ranked according by) a set of documents is a very important topic in information retrieval, but it is often neglected in Encrypted Searching. There are two primary ways to perform this mapping function: Boolean keyword searching or ranking documents according to their relevancy to a given query.

|  |  |  |
| --- | --- | --- |
| Methods | Advantages | Disadvantages |
| Boolean keyword search  [7], [11], [1], [13], [21], [22] [23] | Simple  Fast queries | Documents are either relevant or irrelevant to a query (no degree of relevancy), so recall and precision suffer |
| Rank-ordered search  [24], [25], [26], [27], [28], [18] | Much better precision and recall on results  Draws from extensive research in the IR community | Answering queries is more computationally demanding (this may especially important in the context of cloud computing, where every second of CPU time is charged)  More information about documents must be leaked to support degrees of relevancy |

## Definition of a query

A query represents an *information need*. In practice, it a string of terms, where a term is either a keyword or an exact phrase surrounded by quotes. In BNF notation, the syntax for a query is:

|  |  |  |
| --- | --- | --- |
|  |  |  |

Consider the following query:

Volunteer “doctors without borders”

This query consists of two terms: the keyword volunteer and the exact phrase doctors without borders. When conducting a search on a collection of secure indexes, it will look for that exact phrase and keyword. It will not count doctors with borders as a hit since the phrase must exactly match[[8]](#footnote-8).

A secure index typically only stores the unigrams (keywords) and bigrams found in the document it represents. Thus, any query with an exact phrase consisting of more than two keywords must be converted into a chain of bigrams (biword model).

## Boolean search

Encrypted Boolean search [1] looks for a particular word or words [13] in a set of documents. In Boolean search, a document is either considered relevant (e.g., contains all of the terms in the query) or is considered non-relevant to a given query.

In addition, while there are some Boolean search techniques that allow for approximate matches or are tolerant of errors (like typographical errors), most encrypted search techniques must still rely upon some form of exact string matching due to the way the words in documents are transformed (for confidentiality) using one-way hash functions (trapdoors). This need for confidentiality complicates many standard information retrieval techniques.

For example, if tolerance of typographical errors is desired (as measured by edit distance, see section 6.2.1.2.2), the Levenshtein distance algorithm may be used in non-encrypted searching. However, since the words in a document are cryptographically hashed, this algorithm is useless (i.e., two words with an edit distance of should hash to completely different strings). The solution proposed by Li [11] addresses tolerance of typographical errors by pre-computing and including all error patterns up to errors directly in the index[[9]](#footnote-9), upon which simple exact string matching may thus be performed.

### Extensions

#### Conjunctive keyword search

In [7], the authors propose a system which permits secure conjunctive queries for certain keywords on a given set of fields, like the “From” field in an email. By secure, they mean that given access to a set of indexes for encrypted documents and a freely chosen set of encrypted keywords (trapdoors), adversaries—like an untrusted cloud storage provider—must not be able to learn anything about the encrypted documents except whether it matches those specific trapdoors.

Their work demonstrates a slight improvement over the single keyword searching discussed in [9], but their solution is still rather limited. First, they still only perform exact string matching. Second, their solution inflexibly requires the document creator to tag specific keyword fields for search-ability[[10]](#footnote-10). Finally, like most other solutions, they do not consider untrusted parties that consider historical data [17].

#### Approximate keyword matching

In [12], the authors point out that, for Google, not returning enough results is not a problem. Indeed, Google’s primary problem is finding ways to return fewer, more relevant results so that users do not have to sift through too many results (many users only check the first page of results). Thus, Google is motivated to improve the precision (see section 7.3.1), which is the ratio of relevant documents returned to the total documents returned. However, in vertical search--such as encrypted searching over an enterprise's store of encrypted documents--there is far more concern over not missing or overlooking relevant documents, since there may be so few relevant results to begin with. Thus, vertical search tends to have an objective in direct contrast with Google’s. That is, recall, which is the ratio of relevant documents return to total number of relevant documents, is equally or even important to than precision.

Elsewhere, we discuss relevancy scoring, which provides a more sophisticated approach in that the goal is to rank documents according to how relevant they are to a query as opposed to the simple Boolean "relevant" or "irrelevant" score found in Boolean keyword searching. But, a simple extension to Boolean keyword searching which improves the recall (at the expense of precision) is to do more tolerant matching on the keywords.

##### Locality-sensitive hashing

In locality sensitive hashing, the notion is to map similar (according to some distance measure) items to the same hash. This is an especially good fit in the context of encrypted searching, since in encrypted searching, only exact matches (on the encrypted bit strings) is possible.

To avoid the curse of dimensionality—or in other cases avoid having to explore a space that combinatorially explodes—locality-sensitive hash functions may be used as a form of dimensionality reduction [29]; that is, use hash functions in which the probability of a collision is high for “close” elements and low otherwise (distance preserving). Thus, LSH functions are not at all like cryptographic hash functions; cryptographic hashes, like most hash functions, are designed to minimize the probability of collisions, but in general LSH hashes are designed to maximize collisions in some sense.

For instance, in [30] the authors observe that Bloom filters generally assume cryptographic hash functions, or at least hash functions which uniformly distribute over the domain (bit positions). However, if this requirement is relaxed, then a choice of hash functions can be made which is more likely to test positively for non-members that look like members by using locality-sensitive hash functions. Unfortunately, this will cause more information leakage; for minimizing information leakage, the hash functions should uniformly distribute over the entire domain.

###### Stemming

Stemming may be thought of as another form of locality sensitive hashing. In stemming, morphological variations of a word are mapped to a single base form. By reducing such variations to a single form, in which the different variations have the same essential meaning, recall and precision may both be improved significant.

For example, if a user searches for “computing grades”, it would seem the user would find “computed grade” relevant also. By not including this variation in the result set, the recall and potentially the precision are reduced: the recall is reduced because not all of the relevant documents are returned, and the precision is potentially reduced because a less relevant document may be returned in its place. Stemming has demonstrated itself to be a fast and effective technique to improve precision and recall. [31]

###### Phonetic algorithms

Phonetic algorithms are another form of locality sensitive hashing. The notion is to map words that sound alike to the same hash. Soundex is one of the more popular examples of this; it is an especially useful trick for approximate matches on the names of people.

##### Edit distance

In [11], a mechanism is proposed to address the limitation in which only exact matches on keywords are performed. In particular, they propose a construction that allows for matches on typographical errors or typical spelling variations, e.g., “color” vs “colour”.

To accomplish this, when constructing the index, for each term in the document, add all size k edit distance patterns, where an error is an insertion, deletion, or substitution of a character. For example, for a 1-edit error tolerance, the keyword “age” is expanded to {age, \*age, a\*ge, ag\*e, age\*, \*ge, a\*e, ag\*}, where the \* represents any character. Thus, if “age” fails to match, the query can be automatically expanded to each of those variations in turn until a match is found.

##### Wildcard matching

Wildcard [32] searches can be quite useful. For example, if users are unclear on how to spell a particular word, they can use wildcards to represent their ignorance, e.g., instead of “tomorrow”, they may type “to\*row”. Or, as another example, the user may seek multiple variations of a word, e.g., “\*night” for “night” or “knight”.

The solution proposed in [11] can be repurposed to implement wildcard searching.

### Exact phrase matching (word n-grams)

Most searchable encryption schemes only allow matches on keywords, but in [33], a method for secure exact phrase matching is elaborated on. Phrase searches consist of approximately 10% of web search queries, so this is an important capability. Unfortunately, they require clients maintain a local dictionary on their computers to facilitate the capability. As long as such data must be maintained locally to perform searches, one may reasonably argue that local searchable indexes should be maintained instead. Local indexes, freed from many of the security concerns, would permit any sort of search operation without the need to communicate with server until a specific document is desired (thus less information leaks in the form of access patterns)

#### Biword [26] model

The notion is, to accomplish exact phrase queries, as long as the index supports 2-grams, any n-gram exact phrase search can be expanded to a sequence of 2-grams. For example, to find the exact phrase, “hello dr fujinoki” can be represented as the Boolean keyword query, “hello dr”, “dr fujinoki”. Do note, however, that this opens up the possibility for false positives, as this expanded query will also match any document in which “hello dr” and “dr fujinoki” are present—they do not have to be adjacent to each other.

Simple extensions can exploit k-gram members, like 3-grams, to reduce the probability of a false positive. The biword model would work well with secure indexes, like the Bloom filter or minimum hash constructions.

## Rank-ordered search -- degrees of relevancy

As pointed out in [18], most Encrypted Search research focuses on Boolean search, where a document is either relevant or non-relevant to a given query—that is, there are no degrees of relevancy. For instance, a Boolean search may consider a document a match—relevant—if and only if all of the terms in the query are matched in the document.

As indicated elsewhere, this has a number of problems with respect to precision and recall. The results in this paper represent an important advance over prior encrypted searching schemes in that it ranks documents according to some measure of their degree of relevancy to a given query.

In modern information retrieval systems, scoring the relevancy of a document to a query is one of its most important functions. If standard scoring techniques can be employed in Encrypted Search, its utility could be significantly improved. However, in Encrypted Search, a server obliviously searches over a collection of encrypted documents. The server’s ignorance (which is needed to preserve confidentiality—i.e., confidentiality of queries and documents) complicate many traditional scoring techniques.

For instance, commonly a vector-space model is used in which documents are represented as unit vectors, where each dimension of the vector corresponds to a term’s normalized tf-idf weight (see 6.3.1.1). With this representation, an efficient similarity measure between two documents (or between a document and a vectorized query) is the cosine similarity measure. However, in Encrypted Search, the terms in a document should not be known a priori. Rather, such information should only be approximately learned through user submitted queries (and only in terms of trapdoors—cryptographic one-way hashes). Thus, documents may only be modeled using this more limited information[[11]](#footnote-11).

In this paper, we will explore other more compatible measures, e.g., instead of cosine similarity one can simply measure the score the relevancy of a document just with respect to the terms in the query, e.g., a simple summation.

### Term importance weighting

Term (keyword) weighting [27] is based on two fundamental insights. First, some of the terms in a query will occur more frequently in one document compared to another document. When scoring the relevancy of documents and to a query term , if appears more frequently in than then should be considered more relevant all other things being equal.

where is a monotonically increasing function. An offline secure index, like one based on Bloom filters (probabilistic sets), may be adapted to work with these frequency (multiplicity) metrics [34].

The second insight is that some of the terms in the query will be in a larger proportion of the documents in the corpus. These terms, therefore, carry less meaning—that is, they have less discriminatory power since they appear in a larger fraction of the documents. Conversely, some of the terms in the query will be very rare or even unique in a corpus, and thus they have more discriminatory power.

For example, the term “the” is in nearly every document—it serves as linguistic glue— but the term “acatalepsy” is in very few documents. The more discriminatory power a term has, the more weight it should be given when scoring a document’s relevancy to the query. Mathematically:

where is a monotonically decreasing function (e.g., the log of the inverse).

Combining both of these insights, a measure that is sensitive to both of the rarity of a term in a corpus and the frequency of a term in an individual document may be devised. Mathematically:

where and , e.g., a function that takes the product of the two inputs.

#### tf-idf term weighting heuristic

A notable example of a term weighting heuristic is tf-idf and its variants. The tf-idf heuristic—term frequency, inverse document frequency—accounts for both the term frequency within a document and the inverse document frequency within a document collection when estimating the importance of a term:

### Term proximity weighting

In [35], the importance of proximity of terms in keyword searches is considered. The fundamental principle can be demonstrated by considering the following: given two documents, and , should be more relevant than for the query even though they both contain the two keywords and with the same frequency. Put simply, all other things being equal, the closer the query’s terms are in a document, the more relevant the match.

### Semantic search

In [25], the authors maintain that most search techniques—from simple Boolean search to vector-space tf-idf weighted scoring—are variations of syntactic search in which some form of string matching is performed combined with a method to estimate how important particular string matches are.

There are two major problems with the syntactic approach. First, different terms may be used to express similar meanings (depending on the context). This is referred to as synonymy. Second, the same term may be used to express different meanings (depending on the context). This is referred to as polysemy. Both of these problems may degrade recall and precision of search results.

Semantic search attempts to mitigate these problems by modeling the meaning of text, with the aim of more intelligently mapping an information need—as represented by a query—to a set relevant rank-ordered list of documents that satisfy the information need, e.g., does the meaning of the query correspond to any similar meanings in a document?

Modeling semantics is a more complicated problem than string matching. It may involve natural language processing to perform word-sense disambiguation, part of speech tagging, and named entity recognition. When combining this with ontological and semantic knowledge, the information retrieval system may begin to process search queries in a way that resembles a human's ability to understand the meaning of text.

There are also statistical techniques to model semantically related terms, like latent semantic indexing (LSI). There has been very little progress on either of these fronts in relation to Encrypted Search*[[12]](#footnote-12)*.

# Research objectives

Our research will contribute to Encrypted Search (and more generally, oblivious search), in several different ways.

Research focus:

1. Using established/standard information retrieval scoring techniques
   1. Accuracy
      1. Precision/recall (Boolean) – for a given fp rate?
   2. Computational efficiency
      1. Compare to BSIB
   3. Space efficiency
      1. Compare to BSIB
2. Query privacy against attacker who had access to query histogram data
   1. Effectiveness/efficiency of secrets
   2. Effectiveness/efficiency of obfuscations
   3. Effectiveness of crypto hash collisions
3. Data/document confidentiality – simulation is future work, analysis already done
   1. Effectiveness/efficiency of trapdoors/cryptographic hashes
   2. Effectiveness/efficiency of location uncertainty
   3. Effectiveness/efficiency of false positive rate (per member—unigram/bigram)
   4. Effectiveness of crypto hash collisions
   5. Effectiveness of secure index poisoning
      1. Adding fake terms
      2. Adding frequency error for secure indexes which can store perfect frequency information
         1. PSIP, PSIF

## Secure indexes

Encrypted Search will be facilitated by one or more secure indexes. To make an encrypted document searchable, a client must construct a secure indexfor it. The secure index will only allow someone who has secret information meaningfully query it.

There are several different kinds of secure indexes analyzed and explored in our research. As far as we are aware, with the exception of BSIB none of these secure indexes have been previously investigated.

### Overview

#### Secure index construction

What follows is the sequence of actions needed to construct a secure index. Once constructed, the secure index may be stored on untrusted systems, such as a CSP. Refer to **Error! Reference source not found.** for a visualization of the steps.

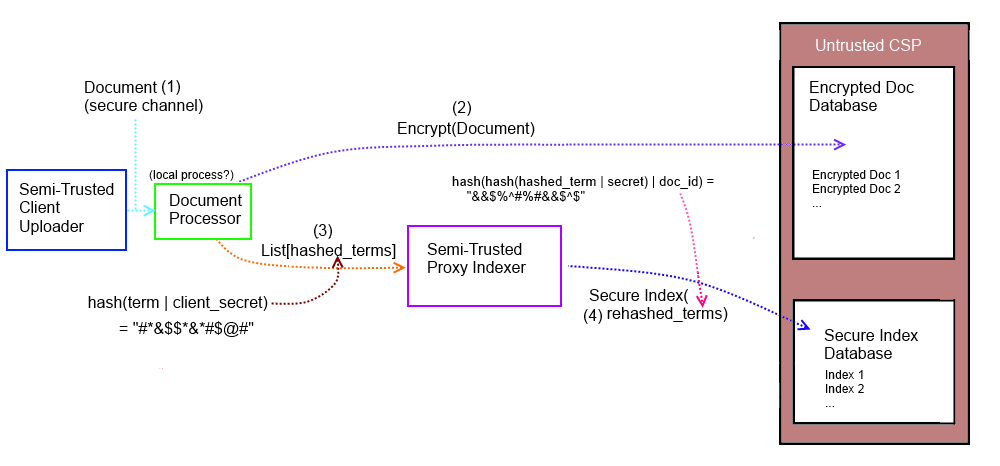
1. A client transmits a sensitive document over a secure channel to a *document processor*. This is most likely a process running on the client’s local machine, but in theory it can reside anywhere. There are compelling reasons to decouple the document processor from the client, e.g., centralized control to enforce uniformity of secure index construction.
2. Optional: The *document processor* encrypts the document using whatever cryptographic algorithm is deemed appropriate and transmits it to an untrusted channel, e.g., the CSP or a system decoupled from the CSP. For Encrypted Search purposes, only a reference to the document needs to be provided as output.
3. First, the *document processor* generates *searchable terms* for the document. These terms can be anything—Soundex hashes, trigrams, etc.—but in our implementation they are lower-case unigrams and bigrams (optionally stemmed) contained in the document (biword model). Second, the concatenations are fed into a cryptographic one-way hash function. Finally, it generates a list of hidden terms from the output of the one-way hash function and transmits the intermediate results to a *proxy indexer*.
4. First, the *proxy indexer* concatenates one or more *secrets* to the intermediate hidden terms and feeds them into a one-way hash function. Second, the proxy concatenates the document’s id (reference) to the outputs of the previous hash function and feeds them into another one-way hash. Third, a secure index is constructed from the hash function’s output. Finally, it transmits the *secure index* to the CSP. At this point, the CSP stores the secure index and, optionally, its corresponding encrypted document in a database to facilitate efficient Encrypted Search operations in response to hidden queries.

Figure 1

##### User revocation

In [37], the authors observe that most Encrypted Search implementations assume only one person will be able to perform the searches; or, if multiple people, then they all must share the same secret, and that secret by itself will allow them to query the secure indexes. However, it may be desirable to be able to revoke a user’s ability to query a secure index. For example:

* makes a secure index for a document using
* is trusted to query secure index by providing her with
* loses trust in and wishes to revoke her ability to query the secure index, even when the raw contents of the secure index is otherwise accessible to .

The motivation for partitioning *secure index* construction into two separate stages in **Error! Reference source not found.**, a and , is to enable user revocation. Assuming the proxies ( and ) and users do not collude[[13]](#footnote-13), neither the proxies nor revoked users will be able to query secure indexes, even if they downloaded copies of the secure indexes to their local machines.

In our architecture, this can be implemented through partial secrets. The following steps are required for secure index construction:

* A fully trusted provides a partially trusted with set of secrets .
* provides partially trusted with another set of secrets .
* constructs a sequence of cryptographic hashes from the ordered unigrams in the document concatenated with secrets in ——and transmits the ordered sequence[[14]](#footnote-14) to .
* constructs a secure index from the sequence of cryptographic hashes (intermediate trapdoors) using secrets in —.

Likewise, the following steps are required for submitting queries to the secure index:

* provides a partially trusted with set of secrets .
* provides partially trusted with .
* submits hidden queries using any secret in to
* transforms the intermediate hidden queries using any secret in and transmits the resulting hidden query to the CSP.

To revoke Encrypted Search authorization for , instruct the not to honor her queries. Even if had downloaded a local copy of a secure index, she cannot meaningfully query it since she does not know any secrets in .

##### Additional Notes

The prospect of constructing a secure index for the entire collection of documents sounds tempting. This *master index* would allow the CSP to avoid the cost of independently querying each secure index in the database. However, there is a significant problem with this idea: the same term in different documents will look the same. The purpose of the *Doc ID Processor* is to eliminate corpus-wide patterns of this nature to mitigate attacks based on frequency analysis.

An alternative to the master index is a cache. See section 9.10.

#### Secure index hidden query

Assuming the *secure indexes* for a collection of confidential documents have been constructed and transmitted to the CSP, Encrypted Search proceeds as follows. Refer to . for a visualization of the steps.

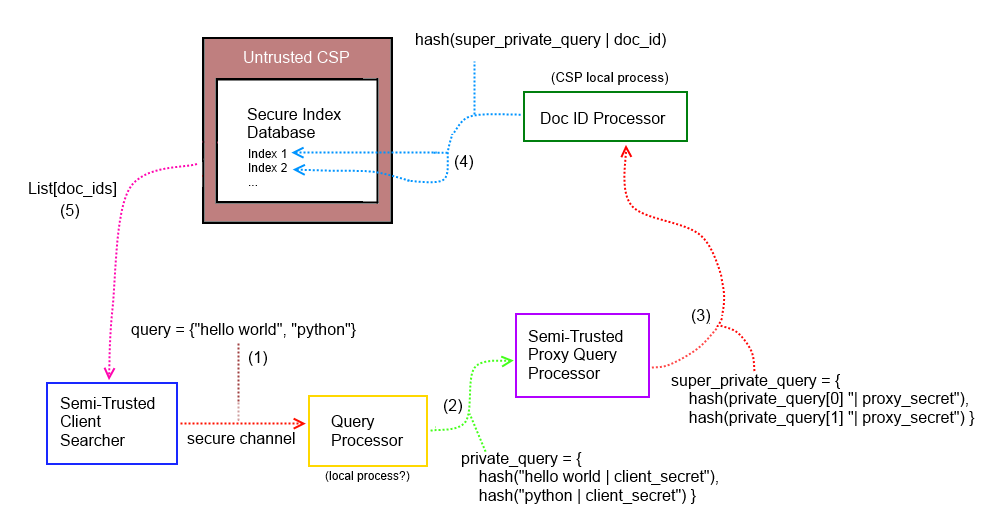
1. Clients construct queries to find sensitive (encrypted) documents relevant to their information needs. Queries will be sent over a secure channel to the *query processor.*
2. First, the *query processor* concatenates a *secret* to each term. Second, it feeds each concatenated term into a one-way hash function. Finally, it generates an intermediate *hidden query* from the output of the one-way hash function and transmits the result to a *proxy query processor*.
3. The *proxy query processor* concatenates a secret to each term in the intermediate hidden query with another secret and transmits the resulting *hidden query* transformation to the CSP. Note that the proxy only observes intermediate *hidden queries* (it does not know what the client is searching for), thus the proxy need not be fully trusted.
4. The CSP iterates through the secure indexes in the DB and ranks them according to how relevant (with respect to *relevancy measures*) they are to a hash of the hidden query concatenated with each respective document’s id.
5. The CSP returns the top-k ranked list of document IDs (e.g., URLs).

Figure 2

##### Additional notes

The reason for the second hash in (3) is to make it so that the same term will appear differently in separate secure indexes to prevent correlations between secure indexes from being exposed and to mitigate attacks based on frequency analysis.

The *document processor* and *proxy indexer* may reside on the same machine; indeed, if the client is trusted (rather than semi-trusted) both of them may reside on the client’s local machine.

Moreover, the encrypted documents and the secure indexes need not reside on the same server. A secure index and its corresponding encrypted document are independent. The encrypted document, having an independent life from the secure index, can be handled in whatever way is deemed appropriate. Multiple secure indexes may be constructed for a single encrypted document to provide tiered access, e.g., the lowest tier may consist of very coarse blocks and 1-gram terms (to facilitate only keyword queries with coarse term proximity), whereas the highest level may include exact phrase matching and wildcard queries.

### Interfaces

#### SecureIndex

getType: SecureIndexType

Return the type of secure index. Presently, this includes PSIB, PSIF, PSIP, PSIM, and BSIB.

getReference() -> String

Return the reference (name) of the document the secure index represents, e.g., an encryption of the document’s URI.

The secure index may be decoupled from the actual document.

getApproximateWords() -> Integer

Return the approximate number of words in the document the secure index represents.

getLocations(hiddenTerm) -> Vector[Integer]

getFrequency(hiddenTerm) -> Integer

getMinPairwiseDistance(hiddenQuery) -> Map[Pair[hiddenTerm, hiddenTerm], Integer]

contains(hiddenTerm) -> Boolean

containsAll(hiddenQuery) -> Boolean

containsAny(hiddenQuery) -> Boolean

### Perfect filter

In place of the Bloom filter, as proposed in [16], we propose the Perfect filter.

It is a conceptually simple probabilistic set data structure. First, use a perfect hash to map members to unique integers. If the N integers mapped to are in the set , then it is a minimum perfect hash. Using a hash function that maps members to M bit strings, Use the N integers as indexes into an array, where each element in the array is an bit string.

Like the Bloom filter, it can trade accuracy for space-complexity, but it does so in a theoretically optimal way. More importantly, it only requires computing two fast hash functions. Thus, it has a significant computational advantage over Bloom filters. Specifically, with the Bloom filter, the number of hash functions (that minimizes the false positive rate) is , where *p* is the desired false probability rate. For instance, if *p* = 0.001 (1 out of 1000 queries are false positives), then approximately *k* = 10 hash functions are prescribed.

Note that the false positive rate of a conjunction of *r* membership queries is . For example, if each individual query has a false positive rate of *p* = 0.001, then a conjunction of *r* = 10 queries has a false positive rate of approximately *p’* = 0.01. Thus, even a seemingly low false positive rate of may be inadequate, in which case more than the theoretically optimal number of hash functions may be recommended (which also entails a larger bit array for the Bloom filter).

Figure 3 details secure index construction. First, the document is broken up into N blocks. The choice of N represents a trade-off. Larger blocks are more space efficient, but as a consequence *proximity measures* are less accurate. Alternatively, if the blocks are too small, hidden queries reveal compromisingly accurate information.

Next, the blocks are sent to the *Searchable Terms Extractor*. At this stage, decisions are made with respect to which elements (searchable terms) are extracted from the document and inserted into the *Pf* in order to facilitate different search modes. For instance, what combination of the following will be supported?

* Approximate searching. (E.g., stemming, Soundex*,* typographical error tolerance.)
* Exact phrase searching.
* Wildcard searching.

Note that the index itself is agnostic to these questions. Moreover, the size of the *Pf* is determined only by the number of members and the desired false positive rate, e.g., in the *Pf* a large string occupies the same (fixed) space as a small string.

Finally, for each block, a *Pf* for the *searchable terms* is constructed. Upon completion, the *Pf* is added to the s*ecure index*. After this process has completed for every block, the secure index computes metadata about itself and the corresponding encrypted document. For instance, a hash of the corresponding encrypted document will be stored in the secure index as metadata. Thus, if the encrypted document is changed, a secure index representing the older version of it will have a mismatched hash with respect to the newer version. After the metadata has been computed, the secure index is integrated into the CSP’s database to enable it to obliviously search the encrypted documents in response to *hidden queries*.

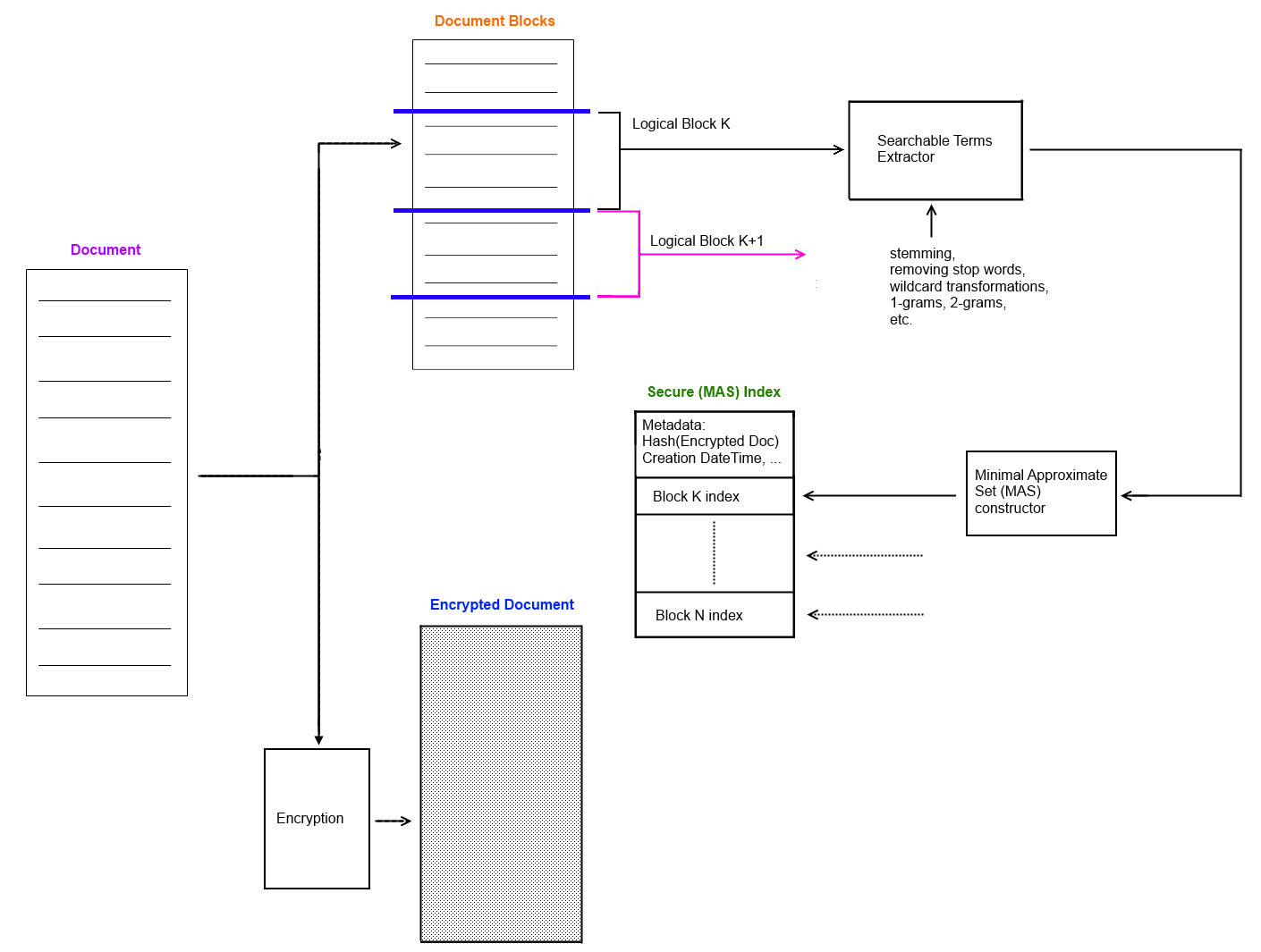


Figure 3

#### Disadvantages

The *Pf* has some disadvantages compared to the *Bloom filter*.

* The Bloom filter leaks less information in some respects since there is no possibility of collisions between members in a *Pf*. However, like with the Bloom filter, each member in a *Pf* collides with an infinite set of non-members with a false probability rate , where *m* is the number of bits allocated per member. Moreover, by constructing fake, random *dummy* members, the *Pf* can be seeded with *obfuscating* garbage. This garbage will only cause false positives on non-member queries equal to the false positive rate decided upon when constructing the *Pf*. This false positive rate can be reduced as much as needed; each additional bit allocated halves the false positive rate.
* The time to construct a *Pf* is slower compared to a Bloom filter, although it is still linear in the number of members.
* The *Pf* is a static (immutable) data structure. Unlike the Bloom filter, once it has been constructed no new elements may be added (or deleted). However, this is not as problematic for *encrypted search* since most modifications to the original document would entail a Bloom filter reconstruction also.

#### Additional notes

If more fine-grained term frequencies are desired (e.g., for more accurate keyword weighting) without decreasing the block size, then a *multiset* is a more appropriate data structure. A multiset tracks the multiplicities of its members.

Fortunately, it is trivial to extend the *Pf* to a multiset. This is accomplished by using the minimum perfect hash to index into another vector consisting of *r* bits per element, which can represent discrete values. For example, for *r* = 2, four unique values can be represented. Thus, if it is desired to track the multiplicities of terms in a block, these four discrete values may map to some category. For example, they could map to the four ordinal categories: values, 0 => *only one*, 1 => *between two and four*, 2 => *between five and ten*, and 3 => *more than ten*. The extra multiplicity information may be used to improve the relevancy of search results.

While in theory the minimum perfect hash has optimal space efficiency given a false positive rate, in practice this optimality is not achieved. In practice, the space-complexity savings over the Bloom filter are less significant, although the reduced time complexity is still a big selling point.

### Bloom filter secure index

### Bloom filter secure index-block (BSIB)

A secure index capable of answering approximate frequency and location requests for query terms. BSIB uses a Bloom filter as the underlying data structure.

Similar to PSIB, BISB is a block-based secure index. A document is segmented into N blocks, and for each block a Bloom filter (another kind of approximate set) is constructed such that the unigrams and bigrams residing in that block are inserted into it.

The frequency for a query term is approximated by counting how many of the N blocks it tests as positive in. If the query term is a unigram or a bigram, a test for membership in the block’s Bloom filter is performed directly. Otherwise, a test for the presence of all the n-gram query’s n-1 bigrams is performed, e.g., if the query term is “A B C D”, then test for “A B”, “B C”, and “C D”. If any do not test as positive, that block does not contain the query term. Otherwise, it is said to contain it.

The location for a query term is approximated by storing which blocks it tests as positive in, and returning a random location for each such block range. For example, if each block is of size m words and query term *t* exists in blocks 0 and 5, then two locations will be returned; one in the range and the other in the range .

### Perfect filter secure index (PSI)

**P**erfect-hash **S**ecure **I**ndex. A secure index capable of answering approximate Boolean queries. PSI is based upon the perfect filter.

1. Each unigram or bigram in the target document D is concatenated with *n* secrets—. Every unigram and bigram in the document will thus be searchable with *k* different secrets.
2. Each is cryptographically hashed—[[15]](#footnote-15)
3. Each is concatenated with the document’s identifier (e.g., hash of its filename) and is then re-hashed—. This prevents the same cryptographically hashed term in different documents from mapping to the same hash value[[16]](#footnote-17).
4. Each is uniquely hashed by a perfect hash function s.t. . That is, no collisions among any of the cryptographic hashes are possible. If using a minimum perfect hash, then
5. Let U be a bit array with at least indices, where each index maps to M contiguous bits. Then, .

Note that because an M bit hash of is stored in U rather than the actual value, false positives on non-member strings *x* are possible, i.e., if , then with probability , . This equality denotes a false positive. Therefore, PSI represents an approximate set in which false positives occur with conditional probability .

There is a different way in which a false positive may occur when processing n-gram query terms, n > 2. If a document contains the words “A B B D”, then the PSI will conceptually represent it as the set {“A”, “B”, “D”, “A B”, “B B”, “B D”}. To determine if “A B” exists in the document, a single set membership test will suffice.

However, determining whether the query term “A B B” exists in the document is more complicated (it does not exist in the set if only unigrams and bigrams are members). To support exact phrase searches larger than bigrams, as in the trigram “A B B”, a biword model is used in which n-gram query terms, , are decomposed into a set of *n-1* bigram tests, e.g., testing if “A B B” exists is transformed into a a conjunction of membership tests for “A B” and “B B”. If all bigrams test as positive, the n-gram term is said to exist in the document. In this case, “A B B” will correctly test positively. But if the term is “A B D”, then it will test positively both for “A B” and “B D” but nowhere in the document is the trigram “A B D” found. Therefore, this query term would cause a false positive to occur.

Notes:

1. In my PSI-based experiments, I allow for any false positive rate of the form , where M can be any positive integer. In a more practical implementation, it would be sensible to optimize the special case where M represents a byte-aligned number of bits, e.g., 8-bits or 16-bits, to take advantage of much faster parallel bit-wise operations.
2. PSI is not used in isolation in any of the experiments. Instead, PSIB, PSIF, and PSIP—which build on top of PSI—are tested. In hindsight, it is regrettable I did not include any experiments exclusively for it.
3. I also regret not including a PSI-based index more directly comparable to BSIB, i.e., a secure index which constructs a perfect hash for each block-of-word segments as is done in BSIB.

### PsiFreq (PSIF)

PSIF uses the PSI interface to, for instance, check for the existence of query terms, and on top of that provides an interface capable of answering approximate frequency requests for query terms.

To construct a PSIF, first a PSI is constructed. Then, the frequency of each member (unigram and bigram) is calculated. These frequency counts are then stored in a bit vector in a memory efficient way.

If exact frequency information for positive examples of unigrams and bigrams is not desirable, then during the construction phase an approximation of the exact frequency may be used instead, e.g., .

To service frequency requests for unigrams or bigrams, the PSI interface is used to index into the PSIF’s frequency bit vector. If the query term is not a unigram or bigram, then the frequency is considered to be the minimum frequency of all of the bigrams making up the query term.

### PsiPost (PSIP)

PSIP uses the PSI interface to, for instance, check for the existence of query terms, and on top of that provides an interface capable of answering approximate frequency and location requests for query terms.

To construct a PSIP, first a PSI is constructed. Then, a postings list (a list of positions) for each unigram and bigram in the document is constructed. Finally, the positions in the postings list are randomly changed according some random variate, i.e., . In the experiments, I use a triangular distribution (with a mode equal to the exact position) instead of a uniform distribution. The triangular distribution has less variance and therefore preserves more information about location information.

Frequency requests are serviced in the same way as PSIF using the size of a term’s postings list as the frequency. Note that this means the PSIP does not exploit location information to eliminate false positives. This is done for the sake of speed.

If exact frequency information for positive examples of unigrams and bigrams is not desirable, then insert random positions into the postings lists and/or calculate the average position of adjacent positions for a term and use that average position in place of the two positions. Do this as many times as necessary to achieve the desired level of approximation, e.g., if a term appears times, repeating this times would result in storing its mean position. Of course, this also effects location accuracy.

To service location requests for unigram or bigrams, the PSI interface is used to index into the PSIP postings lists. For an n-gram query term, a greedy algorithm is used to construct non-overlapping sets each with a diameter less than or equal to some constant that depends on the way in which the postings list was changed by the random variate (e.g., when a triangular distribution is used with a maximum offset of positions in either direction, then is the diameter). The positions of the query term are taken to be the center of each such non-overlapping set.

Note that since a greedy algorithm is used, this operation is fairly quick as evidenced by experiments consisting of queries with a large terms. Moreover, the use of a more sophisticated algorithm, e.g., an algorithm which produces the maximum number of such non-overlapping sets, is not obviously an improvement in the context of greater accuracy.

### PsiBlock (PSIB)

PSIB uses the PSI interface to, for instance, check for the existence of query terms, and on top of that provides an interface capable of answering approximate frequency and location requests for query terms.

To construct a PSIB, first a PSI is constructed. Then, the document is segmented into blocks, and a bit vector of size is constructed such that there are N bits assigned to each unigram and bigram in the document. Finally, if a term resides in a block, the corresponding index representing that block in the bit vector for the given term is set to 1. Otherwise, it is set to 0. The larger the N, the more precisely PSIB can locate terms.

Notes:

1. Bit vectors are used so that byte-alignment in memory is not necessary. For all of the PSI and PSI-based secure indexes, this approach is used to minimize the size of the secure index resident in memory without compression.
2. The bit vector representing the blocks a term resides in can become very sparse as the number of blocks increase. This can be very inefficient. However, sparse bit vectors can be easily compressed—see compressed bit vectors. I elected not to do this in my experiments, some of which do clearly reveal the need for compressed bit vectors or some other representation.   
     
   Moreover, I allow for an arbitrary number of blocks per document. However, like with the PSI, a more practical implementation could see a significant performance boost if byte-aligned sizes were used instead, e.g., parallel bit-wise AND operations could determine which blocks contain all the bigrams in a k-gram query term.

The frequency for a query term is approximated by summing the binary digits of the bit vector representing that term’s approximate block locations. Note that for unigram and bigram query terms, the pre-computed bit vector may be used, but if the term is an n-gram, , then the bit vector representing locations for the term is derived from an AND operation on the corresponding bit vector entries for all of the bigrams of the n-gram query term.

The location for a query term is approximated in much the same way as the frequency, except instead of summing the binary digits of the bit vector, a list of approximate locations is returned. For example, if each block is of size m (each block has m words, except the last which may have fewer) and query term *t* exists in and , then two locations will be returned, one in the range and the other in the range .

On false negatives  
Unlike the PSI, false negatives are possible because we use the approximate location information to eliminate positive matches that are most likely false positives. However, if an occurrence of a true positive spans two blocks, that occurrence will be eliminated by the algorithm. One solution is to check for whether the bigrams of an n-gram query term exist in adjacent blocks, e.g., for the query term “A B C”, if “A B” exists in , check for “B C” in either or . For sufficiently long query terms, a chain of adjacent blocks may also be acceptable.

### PsiMin (PSIM)

PSI Min-Pair. To mitigate the threat from the hypothetical attacker elaborated on in {REF}, ideally we would like to avoid storing location information. However, if proximity-aware searching is desired, this is a problematic suggestion.

However, what if we instead stored some sort of relative position, where it is impossible (or at least much harder) to determine a word's' absolute location (e.g., word wi is in word position range k-n to k+n)? this is the idea behind PSIM. instead of storing location information, for word, store the minimum distance to every other word in the document, as long as the minimum distance is less than a specified threshold distance. consider a document consisting of 10 words:

*doc = "word0 word1 word2 word0 word3 word4 word1"*

if we let the threshold distance be 3, then PSIM represents this doc as:

minimum pair-wise distances which are less than or equal to . In general, PSIM is upper-bounded by , where is distance threshold and is the number of words.

Thus, for large k, PSIM is not very practical. However, PSIM is best

used in the context of providing existing secure indexes with much

more precise proximity information for very nearby words, e.g., k ~ 6.

For instance, it may be combined with PSIB; PSIB can be used

to provide frequency information and use PSIM for fine-grained proximity

information. If two words are too far away (min distance is past the

threshold distance), the coarse location information in PSIB can be used

as an approximation. So, PSIB becomes a fall-back option. PSIM is the

preferred method because it provides accurate min-pairwise distance

information, but does so in a way that discloses less information. Note

that approximate pair-wise distance information may also be used in

PSIM if this level of accuracy is vulnerable to attack.

PSIM also may act as a way to guard against false positives due to the

biword model, i.e., if PSIB uses a PSIM with a distance threshold of

k and the user is searching for an n-gram, then at the very most word i

in the phrase must be preceded by word{i-j} by a min-pair distance of j for

j = 1 to min(k, i-1). For even small k, this would make false positives

extremely unlikely (although false frequencies can still occur). Indeed, it

may work so well that if a secure index uses a PSIM for this, it may disable

its own checks on guarding against

false positives due to biwords. (PSIB and BSIB use a check where every

word must fall in the same block, which can cause false negatives since

a phrase can span multiple blocks, and PSIP doesn't even bother to use

its proximity information to guard against false positives due to the

biword model, although it could).

Note that while PSIM is not very space efficient, it can replace a secure

indexes bigrams, e.g., only insert unigrams into the secure index, and PSIM

will serve as a superset of the bigram/biword model.

Optimization for PSIB:

Do not include bigrams (or the generalization PSIM) in individual

blocks. Only include unigrams in the blocks. To see if a phrase exists,

check that all of the keywords (unigrams) exist in at least one of the

blocks, then use PSIM to check that these keywords exist at most at

the distances demanded by the phrase (two words may be even closer

in the document than they are in the phrase, of course). if not, then

it is not a true match; if so, then it's still possible it''s a false

positive, but the larger the PSIM k value, the less likely a false

positive will get through.

note however that this solution is not good for when we want to deal

with multiple occurrences of the same phrase.

## Definition of a hidden query

A hidden query is a cryptographic transformation of a query (see section 6.1).

Unigrams (Keywords)

Bigrams (Biword model)

Trapdoors

Obfuscations

Secrets

Benefits/drawbacks of hashing unigrams/bigrams to N bits

## Relevancy metrics

Encrypted Search efforts have largely ignored the thorny though necessary (for commercial appeal) task of trying to match queries to relevant sets of documents in more sophisticated ways that predictably effect confidentiality. That is, rank-order documents by a measure of how close they match a hidden query.

In information retrieval, finding effective ways to measure relevancy is of fundamental importance. To that end, they have devised many clever algorithms and heuristics to rank-order documents by their estimated relevancy to a given query. We will explore term weighting (6.3.1) and term proximity weighting (6.3.2) heuristics in the context of our secure index constructions.

### Precision and recall

Precision and recall are relevant metrics for Boolean searches; they do not rank retrieved documents like BM25 or MinDist\*; a document is either considered relevant (contains all of the terms in a query) or non-relevant.

Precision measures the proportion of retrieved documents that are relevant to the query. It is defined as:

Precision has a range of [0, 1]. Recall measures the proportion of relevant documents that were retrieved. It is defined as:

Recall also has a range of . It is trivial to achieve a recall of by retrieving every document in the corpus. This, however, comes at the cost of decreased precision. Thus, in general, there is a trade-off between precision and recall.

### Estimating relevancy with term weighting

The tf-idf heuristic—term frequency, inverse document frequency—is a family of heuristics in which both the term frequency within a document and the inverse document frequency within a document collection are used to estimate how important a term is in a query.

The tf-idf heuristic is based on two insights. First, some of the terms in a query will occur more frequently in one document compared to another document. When scoring the relevancy of documents and to a query with a single term , if appears more frequently in than then should be considered more relevant. In other words, the higher the frequency a term t in a document, the more weight it should be given when scoring its relevancy to the query.

where is a monotonically increasing function.

The second insight is that some of the terms in the query will be in a larger proportion of the documents in the corpus. These terms, therefore, carry less meaning—that is, they have less discriminatory power since they appear in a larger fraction of the documents. Conversely, some of the terms in the query will be very rare or even unique in a corpus, and thus they have more discriminatory power.

For example, the “the” is in nearly every document—it serves as linguistic glue— but “acatalepsy” is in very few documents. The more discriminatory power a term has, the more weight it should be given when scoring a document’s relevancy to the query:

where g is a monotonically decreasing function (generally, the log of the inverse).

In tf-idf, both the term frequency within a document () and the document frequency within a document collection () are used to score the relevancy of documents with respect to a given query term .

where h is typically a function on the product of the two given functions, and .

#### BM25

BM25, a well-established tf-idf variant, is one of the relevancy measures chosen for our experiments. Mathematically:

where is the average size (in words) of documents in .

Parameters and are free parameters. In the experiments, is set to and is set to . These are typical values, although ideally these parameters would be automatically tuned for each secure index. The function stands for term frequency and simply returns the number of occurrences of term in document . The function stands for inverse document frequency. Mathematically:

where is the number of documents in and is a function which returns the number of documents in which had one or more occurrences of term .

When using BM25 to rank search results, each document in is ranked according to query by the function , where is a term in query .

After giving every document a BM25 rank, the results are sorted in descending order of rank as the final output.

Note that the scoring method is used differently in our system than in most real-world information retrieval systems. For instance, IR systems employing a vector space model may construct a vector for each document in , where each dimension represents a word with a weight equal to . Subsequently, a document can be ranked according to a query by taking the cosine similarity of their respective vectors.

### Relevancy of proximity

#### MinDist\*

MinDist, like BM25, ranks documents according to their proximity relevance to a given query. It is a less established ranking heuristic than BM25, but in experiments [36] it had performed well compared to other proximity heuristics. In our experiments, we add additional tunable parameters to MinDist and call it MinDist\*.

It is a proximity heuristic in which the minimum distance between each existent pair of terms are summed over. Thus, it needs location information. So, for example, if query Q = {A, B, C}, where A, B, and C are the terms of Q, and document D = “A B D D A D C”, then the minimum pair-wise distances are: (A, B) = 1, (A, C) = 2, and (B, C) = 5. The summation of these distances is simply s = 1+2+5=8. MinDist\* is a scoring function of *s*.

The intuition behind the min-pairwise summation is, the more concentrated the query terms are in a document, the more relevant the document is to the query, but only up to a certain point. For example, consider a query Q consisting of two keywords, Q = {“computer”, “science”}. Given two documents, A and B, where A contains both “hello” and “doctor” on page 7, and B contains “computer” on page 7 and “science” on the page 20. It is obvious that A should be considered much more relevant to Q since the two keywords of interest are much closer together. However, consider a third document, C, in which “hello” appears on page 7 and “science” appears on page 100. Intuitively, this is not much worse than B; both documents are simply not that relevant, and B is at best only marginally more relevant.

Mathematically, these intuitions are implemented in the following way. Let *Q* be the set of query terms, *Q’* be the subset of *Q* that exist in the given document, and *s* be the sum of the minimum pair-wise distances between terms in *Q’* as previously described.

where .

To see if this function matches our expected intuition—a strictly decreasing function that flattens out as *s* increases—it may be instructive to consider the limits and partial derivative of *MinDistScore* with respect to *s*.

As *s* converges to 0, *MinDistScore* converges to . As *s* converges to ∞, *MinDistScore* converges to . To see if these end points are the maximum and minimum values respectively, let us consider the partial derivative with respect to *s*.

This function, for all positive values of *s*, is negative. The smaller *s* is, the more negative it is; the larger *s* is, the less negative it is—it approaches as s approaches 0 and it asymptotically approaches 0 as *s* approaches ∞. This matches the desired intuition; for small *s*, a small increase in *s* corresponds to a large decrease in *MinDistScore*; and, for large *s*, a small increase in *s* corresponds to small decrease in *s.* In other words, its graph flattens out as *s* increases.

It is also reassuring to note that for large |Q’|, the function will decrease less rapidly than for small |Q’|, which is the desired behavior. Recall that |Q’| corresponds to matching more of the terms in the query. Thus, we do not wish to unduly penalize a document which contains more of the query’s terms but spread out over a larger region.

In the experiments, for the secure indexes we bind the parameters to *MinDistScore*(Q’, s; . For the canonical index (index with perfect information), all but the β parameter is the same; β has been set to instead of . This causes the canonical index’s *MinDistScore* to decrease more rapidly as *s* increases. However, since the experiments measuring MinDist\* output do not interact with any other scoring functions, like BM25, the ordering of how documents are ranked is the same for any MinDistScore binding as long as β > 0, γ > 0, and they have the same value for θ. Since this is the only aspect of the output that matters in the MinDist\* MAP experiments, this difference for β has no effect on the results.

MinDist\* is most appropriately used as a way to add proximity sensitivity to already established scoring methods, like BM25. For example, a linear combination of their scores can be used as the final output of a scoring function that is both sensitive to proximity and term frequencies:

Since training data is abundant—it is the output from the canonical index for a set of queries on a given corpus—ideally the free parameters (e.g., the parameters in MinDist\*) for each secure index would be independently optimized using a supervised learning algorithm to minimize some error measure on actual output versus the expected output. This would make for an interesting area of future research.

### Measuring performance of relevancy scores

#### Mean average precision (MAP)

MAP is a way to measure a secure index’s BM25 and MinDist\* output scores. It does this by measuring how closely its outputs matches a canonical (expected) output, as determined by a non-secure index that provides perfect location and frequency information.

The more approximately a secure index represents a document, the less information one can infer about the document from the secure index. Thus, to what extent the secure index can approximate a document while still achieving high MAP scores is an important question.

MAP is calculated by taking the mean of the average precisions on over 30 queries. The precision at k is:

The average precision for the top n documents is:  
The mean average precision (MAP) for the top n documents over Q queries is:  
Consider the following. Suppose the ranked list of relevant documents to a query is [3, 0, 1, 2, 4], and the retrieved ranked list (by a secure index) is [2, 4, 3, 0, 1]. The precision at is ; the precision at k=2 is ; the precision at k=3 is , the precision at is , and the precision at k=5 is. Thus, the average precision is . The mean average precision would simply be the mean of the average precisions for queries.

Note that the average precision for the last value of is necessarily 1 if, by that iteration of , the relevant set and the retrieved set contain the same elements. However, in general, this is not the case; for instance, if the relevant ranked list of documents to a query is (A, B), and the retrieved ranked list is (D, C, B, A), then if the mean average precision goes from to , the average precision is 0. In my simulation, I do a variation of this.

Suppose the relevant ranked list of documents to a query is (A=0.9, B=0.85, C=0, D=0), and the retrieved ranked list is (A=0.9, C=0.85, D=0.5, B=0). Then, I calculate the average precision for the top instead of the top or top . In this example, document B is not included in any of the precision at to calculations.

Finally, in one of the experiments, I conduct a “page one” MAP test, i.e., I find the mean average precision using only the top 10 results. The randomized algorithm does much more poorly in this instance, e.g., with over 85% probability, the mean average precision will be less than or equal to 0.05.

## Information leaks

To mitigate document confidentiality (4.1) and query privacy (4.2) information leaks, several different techniques will be explored.

To mitigate query privacy leaks, two approaches (that can work in tandem) will be considered: obfuscations and secrets.

In particular, a single term may have up to N random bit string assignments in a hidden query

One can poison the hidden query with extraneous (random) hidden terms – when I do this, it doesn’t seem to effect the rankings of documents much (sometimes not at all – it will if a false positive happens on one of the extraneous terms, since the extraneous terms are just random junk rather than an actual term that would be found in a document). A histogram analysis can still be performed, but it complicates things.

### Simulating an attack using query history

There are many possible ways an attacker could compromise the confidentiality of the secure indexes and the hidden queries. There is the obvious case where a secret is disclosed to a would-be attacker. For instance, once a secret is known and the attacker has acquired authorization to query the secure indexes, it may systematically probe the secure indexes in such a way as to try to classify the secret documents using a trigram language model.

However, we limit our attention to attacks on one of the more vulnerable parts of the system: hidden queries[[17]](#footnote-21). I consider below two general strategies to compromise hidden query confidentiality, cryptographic hash attacks and maximum likelihood attacks. I only consider cryptographic hash attacks, but I conduct experiments using maximum likelihood attacks.

#### Cryptographic Hash Attacks

Cryptographic hash functions take as input an arbitrary-length string and output a fixed-length string (hash value). In general, cryptographic hash functions have the following properties:

1. Pre-image resistance.  
     
   Given a hash , finding an s.t. should be intractable. Lacking this property, an attacker may observe and find one or more candidate *.*  
     
   On the one hand, in the context of hidden queries, lacking pre-image resistance, an attacker may be able to discern which keywords or phrases a target has an interest in finding.  
     
   On the other hand, it may be productive to increase the collision rate so that it is trivial to find an s.t. . This is especially relevant when the attacker knows one or more secrets—in that case, simple dictionary attacks become possible. The collision rate can be controlled such that a dictionary attack will produce some set s.t. . In this case, the attacker may not have enough information to determine which the user was actuall interested in[[18]](#footnote-22).  
     
   it is trivial to find an m s.t. a effective collision resistance means that any collisions found suggests both that (a) the guessed secret is correct (if not already known) and (b) implies that the searcher was indeed looking for .  
     
   Thus, there is a case to be made that pre-image resistance is undesirable in this context. Indeed, since most queries will consist of common terms, if the attacker knows any secrets he can hash a dictionary of common terms to discover what other users are searching for.[[19]](#footnote-23)  
     
   However, if too many collisions on legitimate queries occur, then this may have a negative effect on the accuracy of search results, like BM25 ranking of documents. So, collision resistance represents a trade-off between privacy and accuracy of search results for the kind of attack mentioned above.  
     
   We do not explore this trade-off experimentally, but a simple approach to exploring consists of changing the size of the fixed-length output of the cryptographic hashes (trapdoors); if a hash function maps all input to bits, then a smaller corresponds to a larger collision rate. For example, if there are 64 query terms which are mapped to 4 bits each, then on average (assuming a good uniform hash function) each term will collide with other terms in the population. How this will in practice effect outputs of interest is difficult to estimate without performing experiments.
2. Collision resistance.  
     
   Finding strings and s.t. . A general purpose cryptographic hash function should make this infeasible. Once an adversary learns (or is provided with) the secrets used to construct the hidden queries, if finding such strings

#### Maximum Likelihood Attack

Instead of mounting an attack that depends on finding collisions, I simulate a different kind of attack.

Suppose there is a sample of independent and identically distributed observations – that is, a history of query terms – coming from some distribution , where f is a pmf denoting how probable a randomly sampled query term is.

Thus, the probability of seeing a particular history of terms t1, t2, …, tn is[[20]](#footnote-24):

Each term is mapped to a hidden term . The objective of the attacker is to find a function which maps each hidden term to a term .

To accomplish this goal, I simulate an attacker for which the distribution is known (and, in the simulation, is a Zipf distribution); since this distribution can be estimated by examining queries in an IR system that does not hide the query terms, assuming the attacker can arrive at a reasonable approximation of the true distribution is not unreasonable.

For a given in , the probability of seeing a particular history of hidden terms is:

To discover the most likely mapping function in , the attacker will use maximum likelihood estimation; that is, it will explore the space of and choose a which maximizes the probability[[21]](#footnote-25) of seeing .

Since the space of is , a subset of the space must be explored which has a high likelihood of finding local maxima. In my simulation, I use a hill-climbing algorithm, in which the neighbors to a point in this space are defined as the interchange any two and in (in other words, a swap).

Note that an excellent initial starting point in this space, especially given a sufficient number of samples, is to collect all of the hidden terms, sort them by frequency, and pair them up to the terms in sorted by probability. However, I do not use this initial estimator in my simulation. If I did, this would give an even greater advantage (with respect to mitigating maximum likelihood attacks) to simulations involving multiple secrets or obfuscations.

When we add secrets per term, i.e., hidden terms for term *t* consist of the set {h(*t*|*secret1*, *secret2*, …, *secretm*}, then has the form s.t. each plaintext term maps to hidden terms. Thus, the space of g is now instead of , and it is expected that more samples of hidden terms will be needed for a given level of accuracy (where accuracy is defined as the percentage of hidden terms which have been correctly mapped).

Finally, when we add obfuscations to the vocabulary of hidden terms (without secrets), takes the form:

In the above function , to do not actually map to any plaintext term; they all map to class *obfuscation*. Thus, if a specific set of hidden terms map to class *obfuscation*, there is only one way for each of those hidden terms to be mapped to it. Thus, the space for is ). This is equal to or larger than for all non-negative integer values of and ; as a degenerate case, when (no obfuscations), it reduces to .

Obfuscations introduce additional unknowns that either must be given or estimated. As with the distribution of plaintext terms being given, the probability that a random hidden term is an obfuscation term will also be given, i.e., .

There are many ways to complicate matters for the attacker when dealing with obfuscations, e.g., making it so that the distribution of individual obfuscation hidden terms are similar to the distribution of non-obfuscated hidden terms. However, in the design experiment, each obfuscation term has a uniform probability.

When combining both obfuscations and secrets, the space of is ). In any case, the space of explodes as or grows (the original space was already exponential with respect to ). I only consider increasing or separately, i.e., if I increase , I fix at . Alternatively, if I increase , I fix at .

### Simulating an attacker reconstructing documents from secure index information

Once the attacker has successfully decoded (or has been provided with) the mapping from hidden terms to plaintext terms, it may be possible to reconstruct a document from its secure index.

A secure index contains an approximation of a document’s word[[22]](#footnote-26) frequencies without revealing which words are in the document. In addition, it may contain location information about each of the words.

If the secure index only provides approximate frequency information, then a line of attack may consist of the following steps. First, sample from a word distribution to automatically determine (via queries, assuming the attacker has access to the secure indexes) some fraction of the unigrams or bigrams in the given document and their respective frequencies (multiplicities). Then, using a bag-of-words model, classify the document, e.g., P[medical document | word histogram]. More specific classes are possible also. For example, a single plaintext document can represent a class.

However, more sophisticated attacks may be done. For instance, note that bigrams are more informative feature classifiers than unigrams. Moreover, trigrams are more informative than bigrams. Indeed, the larger the n-gram, the more informative it may be as a feature. The limiting case for this is an n-gram the size of an entire plaintext document. Finding a match on this in a secure index would indeed be very informative.

So, with this insight as motivation, an attacker could use a secure index’s bigrams in conjunction with a language model to partially reconstruct a document from the information contained its secure index. First, the attacker can use a generative language model, like the trigram language model, to probe the secure index for plausible n-grams. For instance, if the attacker finds a positive hit on the bigram “A B”, this information can be used to generate plausible trigram phrases, e.g., sample word *x* from the conditional distribution, P[*x* | “A”, “B”]. If “C” is plausible given the previous two words were “A” and “B”, then check for a hit on the trigram phrase “A B C”. If this this trigram tests positively in the secure index, then generate and test plausible 4-gram phrases by sampling from the distribution, P[x | “B”, “C”].

Repeating the above steps over and over again, a large set of n-gram phrases (that test positive in the secure index) may be constructed. Furthermore, some of the discovered n-grams may overlap in some way, in which case the attacker can automatically stitch the pieces together in various ways. The plausibility of a stitching can be estimated using the language model, especially if multiple consistent stitchings of the same size are possible.

Already, this may reveal a significant amount of details about the document. However, if the secure index also provides location information, the attacker has a much easier job.

If exact location information is provided for each word, then as the attacker finds words as previously described, he puts them in their proper place. As words are placed, larger and larger n-grams are constructed, and the language model can be used to generate plausible candidates for the missing words. Or, since the problem has been vastly simplified, the attacker may choose to exhaustively check every word in a dictionary.

Since false positives are possible, each position may have multiple candidates. To deal with this eventuality in a reasonably straightforward way, the attacker can find an assignment of candidates that tries to maximize the likelihood (given a language model) of a given assignment of candidates to each position.

Given the likely effectiveness of this attack, the reported positions for a word should have some degree of uncertainty—e.g., only reporting that a word falls within some range (block), as PSIB and BSIB do, or scrambling the positions in some random way, as PSIP does.

#### Problems with the block-based approach used by PSIB and BSIB

The block-based approach used in PSIB and BSIB reduces the problem to treating each block as a small document, and solving each one independently without location information using the techniques described at the beginning of this section. Since the document is much smaller, the reconstruction effort will be significantly easier than trying to do this for the entire document.

To paint a clearer picture, if the document consists of words, and the words are segmented into blocks, then there are words per block. If all words in the block are discovered (and ignoring word multiplicities), then there are ways to order them. So, for each permutation, the attacker calculates its likelihood given the chosen language model (e.g., trigram language model), and saves the permutations with the highest likelihoods. Note that the space of permutations is a factor of the original space; for sufficiently large the search space may even be exhaustively explored.

#### An alternative solution that overcomes many of the problems for the block-based approach

A significant problem with the block-based approach is the attacker’s ability to treat each block as a separate, independent problem—this has the effect of reducing the attacker’s curse of dimensionality. PSIP is designed, in part[[23]](#footnote-27), to overcome this problem. In PSIP, there is no such block delineation—instead, words are offset from their true position according to some random variate. This makes it harder to treat the document as a set of smaller independent problems.

For instance, suppose document D = “A B C D E F G H” and to simply matters for D’, the secure index approximation of D, suppose we can swap any word in D with any other word in D as long as the words final position is within two units of its starting position. Then, let scrambled document D’ = “B A E D C G F H”. Is it possible to break this larger problem down into two smaller independent problems?

d1’ = “B A E D” and d2’ = “C G F H” will not work since, in the original document, the first set should contain elements from {A, B, C, D}, but it is missing B and has an additional E. Any 4-gram ordering on these two sub-problems cannot match the ordering in the original document; indeed, in this case, the only sub-ordering that matches the original ordering is “A B” and “F G H”. These two sets cannot be stitched together since they have no overlapping components.

Another possible division is d1’ = “B A”, d2’ = “E D C”, and d3’ = “G F H”. This is a legitimate way to reduce the larger problem into a set of smaller independent problems, but the attacker has no way of knowing this beforehand. For instance, if instead A had been swapped with C, this would no longer be a legitimate partition.

This, I would argue, will significantly blow up the search space for the attacker. The attacker may still use the location information to do things like eliminate impossible stitchings, but it is more difficult to use the location information to create independent sub-problems. Of course, it may be acceptable to reduce the original document into sub-problems with a size dependent upon the location uncertainty and settle for more erroneous solutions. It is also possible to parameterize the segmentation points and include those as additional parameters to optimize, but this has the effect of blowing up the search space.

#### On the effect of false positives

As shown in the previous section, false positives create a problem for a hypothetical attacker attempting to (at least partially) reconstruct a document from the information in its secure index.

Given a secure index of with words, each unique (in order to simplify the discussion), there are permutations. The attacker wishes to find some words (which will test as positive in the index) and then find a permutation that maximizes the likelihood of observing that sequence of words given a chosen language model.

An exact solution is already computationally intractable—. Adding false positives complicates matters even more for the attacker, although it is still in . Suppose false positives occur at a rate of , and the attacker wishes to perform an exhaustive search on the secure index by iterating through a dictionary consisting of words, where the words in the document are a subset of the words in the dictionary. Then, to find the words in the document, of those words from the dictionary will necessarily be true positives and it is expected that there will be false positives.

In total, it is expected that words will positively match. Each one of these words is a candidate, and thus instead of the attacker needing to explore a space consisting of possibilities, the attacker must explore a space of possibilities. The degenerate case evaluates to , but as grows it quickly diverges from for a given .

Thus, we see that on the one hand, a high false positive rate mitigates reconstruction attacks. On the other hand, as some of the experiments were designed to probe, a high false positive rate may cause the searching apparatus to return unacceptably poor results if it is affected unduly by the false positive hits. Ultimately, this is trade-off; the least disclosing is , and the most accurate (for searching) is .

#### Secure index poisoning

The intent of poisoning a secure index is to make it so that the hypothetical attacker will be less successful at (at least partially) reconstructing a document from the information in its secure index.

Experiments have explored poisoning in three general ways:

1. Inserting fake terms (unigrams and bigrams) into a secure index. They seem to have a similar effect to increasing the false positive rate, but unlike increasing the false positive rate, it can be done in a way that should theoretically not affect search accuracy (as experimentation has demonstrated).  
     
   Poisoning often represents a trade-off between accuracy and space efficiency (and time, for some secure indexes). However, perfect hash-based secure indexes may be able to accomplish this at very little (if any) cost since the choice between having fake terms and not having fake terms is essentially a choice between using a minimum perfect hash or a perfect hash. A perfect hash can be far more space efficient—indeed, in hindsight, by default a perfect hash is preferable to the minimum perfect hash even if poisoning is not desired.
2. Adjusting frequencies. For a given term, alter the secure index s.t. the frequency is altered. See secure indexes on page 20.
3. Adjusting locations. For a given term, alter the secure index s.t. the locations are altered. See secure indexes on page 20.

### Access pattern leaks

As described in section 4.3, information leaks can take many other forms. To mitigate such information leaks, in general we can look to Oblivious RAM [17] for inspiration. Oblivious RAM may naively be thought of in the following way: to prevent meaningful statistics from being gathered about a user’s activities, whenever an action—a read or write—is performed, include other randomly chosen actions (e.g., fake queries) to obscure the user’s actual interests or activities.

#### Fake queries

Instead of transmitting a single query for each actual query a user is interested in, a fake query rate parameter causes an appropriate number of additional fake queries to be generated[[24]](#footnote-28). Fake queries are strictly intended to obfuscate the user’s actual queries of interest, just as query obfuscation is intended to obfuscate the actual terms of interest in a single query.

Fake queries may consist of terms chosen from a distribution that is likely to result in a plausible distribution of hits (i.e., relevant to an appropriately large set of documents), e.g., sample the terms in fake queries from a Zipf distribution. In this way, it may be impossible for an attacker to separate fake queries from real queries. Of course, this comes at the cost of increased resource consumption, e.g., the server must process more queries per legitimate query, thus increasing processing and network transmission requirements.

#### Fake secure indexes and multiple secure indexes per document

For each document, construct multiple secure indexes – each one will look different because it will (a) have a random seed value, and (b) have additional known terms that are designed to distinguish it from other docs that are the same as it – e.g., seed the first secure index of the doc with “::variation one::”, the second with “::variation two::”, and so forth. Then, when submitting a query, sample from these variations so that each time a query is submitted, up to N different looking sets of the same actual doc references will be returned per query, assuming we have N different varations of secure index per document.

For this to work, the actual document reference that the secure index itself has in it will also need to be encrypted – but this is trivial, just use standard encryption algorithms to decode it.

ii. One can insert each searchable term multiple times into the document, but with different “secrets”, or even the same secret but concatenated to the term multiple times. If N variations of each searchable term is used, then each 1-gram and 2-gram has N variations, and each phrase search of K words has ????-1 variations. Again, a histogram can still be made, but fewer and fewer correlations can be made, especially when combing (i) and (ii) together. I don’t have a good way to measure how much information is being leaked; intuitively, it’s clear that this is the result, but I don’t quantify it. So, I won’t be analyzing how effective it is at preventing information leakage, but I will (I think) be analyzing to what extent these factors affect the output of our program, e.g., when adding variations of each search term, how much does this effect time complexity/space complexity, or when “poisoning” the query how much does this effect time complexity/relevancy of results?

iii. There is still the problem of a user’s history of data access patterns, e.g., which documents do the users queries typically return as relevant to their hidden queries? This can be dealt with to an extent by adding terms to hidden queries that will probably change the results (e.g., common words), and thus make relevant results (with respect to the untransformed hidden query) further down the list, which means that the users, if they want the same level of recall (the ratio of the number of relevant documents to the query to the actual document returned that are also relevant), will need to increase the number of results to process to get at what they are interested in. This may or may not be a problem. I will not be exploring this in my research, other than to note it in passing. (I haven’t seen this line of research done elsewhere

# Experiments

Experiments are intended to explore how one or more inputs relates to one or more outputs. To avoid over-complicating the experiments, techniques like multiple linear regression were avoided. Instead, we used designs in which one input is changed (while the other inputs are held constant) and one or more outputs are observed with respect to this change.

## Inputs

* Secure index. The type of secure index. It is either PSIB, PSIF, PSIP, or BSIB. In most of the experiments, multiple secure indexes and their respective outputs are compared to one another.
* Documents (documents/corpus). Number of documents in the corpus. A variable corpus size should effect most outputs in a linear way, e.g., MinDist\* lag time should depend linearly on the number of documents (assuming documents are of fixed size). However, MinDist\* scoring and BM25 scoring may be effected in a non-linear way, thus I make this variable to see how such outputs respond.
* Pages. The number of pages in each document in the corpus. A variable page count will be used to see how each *secure index* scales with document size with respect to a number of parameters.
* Terms/query. The number of terms in a query, where a term is either a keyword or an exact phrase.
* Words/term. The number of words in each term.
* Secrets. Number of secrets that can be used to search for query terms in the secure index database.
* Obfuscations. This input is used in two different senses. In the context of the attack simulation, this input refers to the number of unique obfuscations; otherwise, it refers to the number of obfuscated terms added to a query.
* Obfuscation rate. In history attack simulations, obfuscation rate refers to the probability that a random term in the history set will be an obfuscated term.
* Location uncertainty. Unigram or bigrams in the document have exact positions. Exact positions reveal too much information about the contents of the document; thus, positions should only be known approximately. See section 7.4.2.
* False positive rate. A word not in a secure index document will have a probability (the false positive rate) of testing positively as belonging to it. This probability can be controlled by increasing or decreasing the input for false positive rate. On the one hand, a low false positive rate should improve search accuracy; on the other hand, a high false positive rate should improve confidentiality (e.g., more difficult for an attacker to reconstruct the document). See section 7.4.2.3.

## Outputs

* Secure index size. The size (e.g., bytes) of the secure index database for a corresponding corpus (collection of documents).
* Build time. Time taken to build the secure index database for a given corpus.
* Load time. Time taken to load a secure index database for a given corpus.
* Boolean search precision. Proportion of retrieved documents relevant to the Boolean search (all of the terms must be in a relevant document). See section 7.3.1.
* Boolean search recall. Proportion of relevant documents retrieved. See section 7.3.1.
* BM25/MinDist\* rank-ordered MAP. See section 7.3.4.1.
* BM25/MinDist\*/Boolean search lag time. Time taken for the corresponding kind of query to complete.

## Platforms

|  |  |
| --- | --- |
| *Machine A* |  |
| Operating System | Windows 7 Service Pack 1 |
| Processor | AMD A6-6400K APU 3.9GHz |
| Installed memory (RAM) | 8.00 GB |
| Storage Device | Kingston SSDNow V300 Series SV300S37A/60G 2.5" 60GB SATA III SSD |
| Compiler | Visual Studio 2013; 32-bit target; command line = ” /MP /GS /GL /W3 /Gy /Zi /Gm- /O2 /fp:precise /GF /GT /WX- /Zc:forScope /arch:SSE2 /Gd /Oy- /Oi /MD” |

## Global Parameters

* The number of unique words per corpus (corpus dictionary) is fixed at 10,000 words. The unique words in the dictionary follow a Zipf distribution, and they are randomly generated with an average length of 6.5 alphabetic characters. Such a dictionary is uniquely constructed for each trial of every experiment.
* For BM25 scoring, parameter b is set to 0.75 and parameter k1 is set to 1.2. See Background Information for more details.
* Each document of size n (n words) in the corpus separately samples m= unique words from the corpus dictionary, conforming to Heap’s law with K=12 and β=0.5. Once m unique words are sampled, they are renormalized to make them into a proper distribution. It is this distribution that is used to generate the sequence of words for a document.  
    
  I did this with the intention of making each document approximately follow a Zipf distribution, but with a different subset of words to account for different authors with different but overlapping vocabularies. In hindsight, it would have been sufficient (and perhaps preferable) to have simply sampled n words directly from the corpus dictionary.
* When calculating the outputs for a given input, e.g., false positive rate, location uncertainty, etc., I always use a query set consisting of 30 queries. I then average the outputs over all of those queries where appropriate.
* For each query term in a query in a query set, I seed a document in the corpus with that term with probability p = 0.2 except where otherwise noted. Thus, for a query with k terms, the probability that one or more of its terms occurs in the document is . For k = 1, P[at least one] = p = 0.2; for k = 2, P[at least one] = . Thus, if a query consisting of k terms seeds a corpus of size N, then on average documents will be seeded with the query term.   
    
  In general, this should mean when doing a mean average precision calculation, the expected number of documents relevant to a query with k terms will be . The remainder should be non-relevant, i.e., MAP scores of 0, which means it is irrelevant how they are ranked and can thus be ignored in the mean average precision calculation. Of course, if the secure index does score them with a score of non-zero, that is a sign of a false positive, and should and does subsequently degrade the MAP score. This happens with probability for those documents which are nonrelevant—on average, for a query with k terms, this will happen . For k = 3, N = 1000, p = 0.2, and fp = 0.001, it is approximately 1.5 times.  
    
  To clarify, when calculating a MAP score, I include as many documents for scoring in the ranked list as there are non-zero scores for retrieved ranked documents.  
    
  Once a document is targeted to be seeded by a given query term, all such occurrences of the term will occur within a window size of w ~ U(min(size(doc), 2000), max(size(doc), 6000)) except where otherwise noted.  
  Finally, the number of occurrences of the term within that window will be n = }.
* When measuring precision, MinDist\* MAP, or BM25 MAP, each query in the query set (as previously mentioned, there are 30 queries per query set in total) is submitted 10 times for the block based indexes, and the average of those 10 is taken to be the output.

## Results

### BM25 MAP “Page One” Results

Figure 4

|  |  |  |
| --- | --- | --- |
| Experiment Details | | |
| Input | location uncertainty vs BM25 Top 10 MAP (first page of results) |
| Output | BM25 Top 10 MAP | |
| Constants | 1 secret  0 obfuscations  0.001 false positive rate  1000 documents (documents/corpus)  1 or 2 words/term  3 terms/query  16 pages | |
| Testbed | machine A | |

This is the “page one” Google test. Search users do not want to dig through multiple pages to find what they want. Indeed, studies have shown that Google’s second page of results only receives 1.5% of click-through rate.

In this experiment, we get the top 10 results according to the canonical index, and then get the top 10 results each secure index and restrict the mean average precision to only those top 10. This is a much more demanding measure than taking the mean average precision over all of the results.

In Figure 5, I return 10 random documents out of 250 documents and then calculate its MAP score (note that the real experiment is even more unforgiving since it draws the top 10 results from 1000 documents). Here is a histogram of the results; note that over 90% of the results have a MAP between 0.0 and 0.1.

Compare the random results with the results returned from the secure indexes. They all do remarkably well—clearly much better than random (no trial out of the millions tested had a score higher than 0.4 in the random tests, while no trial had a score less than 0.7 on the secure index tests). PSIP and PSIF came out on top, as expected, since they can optionally preserve perfect frequency information for words (unigrams and bigrams) in the document (although false positives are still possible). Indeed, they rarely scored under 95%. Also, note that PSIP and PSIF are independent of location uncertainty—PSIF does not even store location information, and PSIP’s frequency information is independent of the location uncertainty.

The block-based indexes, PSIB and BSIB, also do quite well, although their scores expectedly trail off as the location uncertainty increases.

Figure 5

### MinDist\* “Page One” Results

Figure 6

|  |  |  |
| --- | --- | --- |
| Experiment Details | | |
| Input | location uncertainty vs MinDist\* Top 10 MAP (first page of results) |
| Output | MinDist\* Top 10 MAP | |
| Constants | 1 secret  0 obfuscations  0.001 false positive rate  1000 documents (documents/corpus)  1 or 2 words/term  3 terms/query  16 pages | |
| Testbed | machine A | |

Once again, we see PSIP pulling ahead. However, discouragingly, no matter which secure index is chosen, location uncertainty must be quite modest for MinDist\* to achieve competent scores.

The MinDist\* measure is more sensitive to location uncertainty than BM25 is to frequency uncertainties. This makes sense. When two terms are only separated by a couple of words, they are likely mutually relevant, but as the distance between them grows they rapidly decrease in mutual relevance. MinDist\* captures this intuition: it only scores documents high when they contain the terms (in the query) at a sufficiently close distance.

However, when location uncertainty is moderately large, two terms that are approximated to be only a couple words apart may actually be much further apart (even pages apart). This can have a large, negative impact on the MinDist\* ranked output. Unfortunately, large location uncertainties are desirable for confidentiality—see Simulating an attacker reconstructing documents from secure index information on page 42 for more on this.

Despite these observations, the secure indexes—especially PSIP—still do reasonably well (e.g., they do much better than random chance, as demonstrated by Figure 5). Moreover, encrypted search users will probably be more willing to dig deeper into the results to find what they are searching for.

Note that the intuition behind MinDist\* proximity sensitivity is the primary motivation for PSIM (PsiMinPair), which can (optionally) preserve perfect min-pairwise distance information for terms in the document that are up to words apart.

### Attack Simulation: Obfuscations vs Accuracy

Figure 7

|  |  |  |
| --- | --- | --- |
| Experiment Details | | |
| Input | unique obfuscations (unique strings in the uniform distribution being sampled from) |
| Output | accuracy (proportion of hidden terms correctly mapped to the corresponding plaintext term) | |
| Constants | 1 secret  50 word search vocabulary (every query is composed from the same 50 unique search terms)  50,000 query term history (to be used as data points in MLE)  150,000 samples (in the Monte Carlo simulation to approximate MLE) | |

In this experiment, I am interested in seeing how accurately a hypothetical attacker, using maximum likelihood estimation, can learn a mapping from hidden terms to plaintext query terms with respect to the unique number of obfuscations for several obfuscation rates (which is just the probability that a random query term will be an obfuscated term). Whenever an obfuscation is injected into a query, I sample the obfuscated term from a discrete uniform distribution consisting of N unique strings. See Maximum Likelihood Attack on page 40 for background information on this.

From the graph, one may conclude that for a given obfuscation rate, there comes a point at which increasing the number of unique obfuscations has little effect on mitigating the attacker. The lower the obfuscation rate, the sooner this point is reached. Additionally, the lower the obfuscation rate, the larger the attacker’s limiting accuracy as n goes to infinity.

For high obfuscation rates, it is a mistake to let the number of unique obfuscations N be small. I imagine the reason for this is related to the fact that the obfuscations will tend to have a higher frequency than most of the real terms if is larger than the probability for most real terms. Thus, mapping a non-obfuscated hidden term to the obfuscation class will cause the likelihood of seeing the given history much lower. This presents another area to explore. Instead of sampling the obfuscated terms from a uniform distribution, sample them from a distribution which is designed to resemble, in some way, the real distribution s.t. incorrect mappings are less penalized in the MLE calculation.

Figure 8

In Figure 40, we see that (with the same fixed constants as before) the optimal combination of number of obfuscated terms and obfuscation rate—the combination that minimizes the attacker’s accuracy—is 50 obfuscated terms and 0.2 obfuscation rate.

### Attack Simulation: Secrets vs Accuracy

Figure 9

|  |  |  |
| --- | --- | --- |
| Experiment Details | | |
| Input | secrets |
| Output | accuracy (proportion of hidden terms correctly mapped to the corresponding plaintext term) | |
| Constants | 0 obfuscations  50 word search vocabulary (every query is composed from the same 50 unique search terms)  50,000 query term history (to be used as data points in MLE)  150,000 samples (in the Monte Carlo simulation to approximate MLE) | |

In this experiment, I am interested in seeing effective secrets are at mitigating the hypothetical MLE attacker. Arguably, it can be effective; moreover, its effectiveness comes at no cost to MAP accuracy and query lag time (except, possibly, for BSIB). However, it does cost in terms of inflating the secure index size.

Additionally, the marginal value of secrets has diminishing returns. Eventually, there comes a point where it hardly makes a difference at all, but you are likely to run out of memory space before that happens.

It is worth pointing out that the secrets for a given term are sampled from a discrete uniform distribution. This probably limits the effectiveness of having secrets. The attacker may be able to infer the underlying plaintext distribution using big data and statistics, but the attacker cannot know (just as with obfuscations) the distribution of an individual user’s secret distribution which may be randomly re-defined periodically (not only an unknown distribution, but a moving distribution). Indeed, each user can have their own way of sampling secrets (and the same is true for sampling obfuscated terms).

I expect that any experimental results on this would look quite promising.

### Attack Simulation: History Samples vs Accuracy

Figure 10

Figure 11

|  |  |  |
| --- | --- | --- |
| Experiment Details | | |
| Input | secrets, history size |
| Output | accuracy (proportion of hidden terms correctly mapped to the corresponding plaintext term) | |
| Constants | 0 obfuscations  50 word search vocabulary (every query is composed from the same 50 unique search terms)  150,000 samples (in the Monte Carlo simulation to approximate MLE) | |

As the number of query samples (history) increases, so too does the attacker’s accuracy. This, of course, is expected; the more data the attacker has to learn the model (the mapping from hidden terms to plaintext terms), the more accurate the model should be.

Indeed, for a history size of 512k, if only using one secret the attacker has a 93% accuracy rate. Increasing the number of secrets to 16 reduces the attacker’s accuracy rate to 36%. Lower is certainly better, but preferably it would be lower yet. Granted, this is a toy problem; there are only 50 words in the user’s search vocabulary, for instance. However, according to equation for the curve representing 512k history samples, we would need over 700 secrets to reduce the attacker’s accuracy to 10%. Since secrets inflate the size of the index, this is not a viable option.

However, as discussed elsewhere, if the secrets were not sampled uniformly, much better results could probably be realized.

### Secrets vs Compression Ratio, Build Time, and Load Time

Figure 12

Figure 13

Figure 14

|  |  |  |
| --- | --- | --- |
| Experiment Details | | |
| Input | secrets |
| Output | compression ratio (ratio of secure index size to document size), load time, build time | |
| Constants | 12 pages  256 location uncertainty | |

The only outputs secrets affected were build time, load time, and secure index size. For PSIB and PSIF, load time is nearly constant with respect to secrets. However, all of the secure indexes flatten out as the secrets increase (as they do for build time, also).

The compression ratio output is linear with respect to the number of secrets; this certainly makes sense, as each secret variation of each term will be dedicated a constant number of bits.

### False Positives vs BM25 MAP and Precision

Figure 15

|  |  |  |
| --- | --- | --- |
| Figure 16  Experiment Details | | |
| Input | pages (~250 words/page) |
| Output | secure index size (bytes) | |
| Constants | 12 pages  1 term/query  1 or 2 words/term  1 secret  0 obfuscations  128 location uncertainty  0.001 false positive rate  1000 documents (documents/corpus) | |
| Testbed | machine A | |

While not much space is saved by decreasing the false positive rate, the primary advantage in having a high false positive rate is its effect on confidentiality. The higher the false positive rate, the less certain the information in the secure index is. See On the effect of false positives on page 44 for more analysis on this.

Thus, **Error! Reference source not found.** paints a fairly positive picture for BM25 scoring. Indeed, false positives occurring even a quarter of the time on negative examples still result in a BM25 MAP of around 0.7. And this is only for 1 term/query and 1 or 2 words/term; BM25 tends to perform better when given more terms to work with, as other experiments have shown.

Figure 10 is less encouraging. If false positives occur a quarter of the time here, only 50% accuracy is achieved. Compared to precision, BM25 is far less sensitive to the false positive rate. This makes sense; if a false positive happens when measuring precision, it will admit a term that should not be included in the result set, which will certainly effect its precision negatively. However, BM25is ranking the documents. Thus, even if a document is falsely hitting on a search term, it is the order that counts—how it ranks the document. Other documents will also falsely hit on the term (with probability 0.25), which will cause the false hit to not be very discriminating—that is, it is not rare in the corpus since 25% of the documents have it. In practice, it may do better than this experiment suggests since I did not include particularly rare terms in the query set.

### Obfuscations vs BM25

Figure 17

Figure 18

Figure 19

|  |  |  |
| --- | --- | --- |
| Experiment Details | | |
| Input | Obfuscations (per query) |
| Output | BM25 MAP, BM25 lag | |
| Constants | 1 secret  256 location uncertainty  12 pages  0.001 false positive rate  1000 documents (documents/corpus)  1 term/query, 1 or 2 words/term *or* 6 terms/query, 6 words/term | |
| Testbed | machine B | |

In Figure 17, we plot obfuscations versus BM25 MAP on a rather large type of query—6 terms/query, 6 words/term. PSIP and PSIF perform very close to 100% and each additional obfuscation per query only reduces BM25’s MAP score by 0.005%. PSIB and BSIB also do well and remain largely unaffected by the obfuscated terms as well.

Figure 18 reveals that every obfuscated term added to the query increases the BM25 lag time by 0.014 milliseconds. Note that this lag time cannot be directly compared with the lag time reported in other BM25 experiments, as this experiment was conducted by a different machine (see Platforms on page 47 for more information on the machines). However, it does not seem unreasonably slow. Also, note that every one of the queries is slow compared to smaller, more typical queries performed in other experiments.

Finally, Figure 19 shows BM25 MAP on a more modest set of queries consisting of 1 term/query and 1 or 2 words/term. This results in an across the board reduction in the BM25 MAP score. However, as discussed elsewhere, in that BM25 generally does better on more complicated queries. Unfortunately, each additional obfuscated term injected into the query also has a larger negative impact on the BM25 MAP score, i.e., -0.013 vs -0.005.

### Obfuscations vs MinDist\*

Figure 20

Figure 21

|  |  |  |
| --- | --- | --- |
| Experiment Details | | |
| Input | Obfuscations (per query) |
| Output | MinDist\* MAP | |
| Constants | 1 secret  256 location uncertainty  12 pages  0.001 false positive rate  1000 documents (documents/corpus)  6 terms/query, 1 or 2 words/term | |
| Testbed | machine B | |

Earlier attack simulations demonstrated the effectiveness of obfuscations in mitigating attacks. Judging by Figure 20, increasing obfuscations have almost no effect on MinDist MAP scores. This is certainly welcome news—we can exploit obfuscations without incurring much, if any, loss in MinDist\* accuracy.

Figure 21 shows a linear relationship between obfuscations and MinDist\* lag time. This makes sense; it is essentially the same increase in lag time expected from any additional query terms—obfuscated terms or otherwise.

Note that PSIP is pulling ahead in a majority of the benchmarks measuring lag time or mean average precision. This is not unexpected; subsequent experiments will expand on why this is happening.

### Pages vs Secure Index Size

Figure 22

Figure 23

|  |  |  |
| --- | --- | --- |
| Experiment Details | | |
| Input | pages (~250 words/page) |
| Output | secure index size (bytes) | |
| Constants | 1 secret  256 location uncertainty  0.001 false positive rate  1000 documents (documents/corpus) | |
| Testbed | machine A | |

As discussed elsewhere, PSIB is optimized for smaller documents. For PSIP and BSIB, every page is approximately 1700 and 1500 bytes respectively; their secure index sizes are linearly dependent upon their page counts.

However, as demonstrated by Figure 23 and Figure 24 , the sparse bit vector representation used in the PSIB does well for small to moderate pages, but explodes as the pages increase past a certain point (~100 pages). It is quadratic with respect to page count rather than linear. For small page counts, the squared component is dominated by the linear component, but for large page counts the squared component dominates.

The point of intersection between PSIB and BSIB is ~50 pages. This is the size of a relatively large document; for larger documents (e.g., books) with more than 50 pages, it may be advisable to (automatically) segment them into smaller chunks.

### Blocks vs Secure Index Size

Figure 24

Figure 25

|  |  |  |
| --- | --- | --- |
| Experiment Details | | |
| Input | blocks (block segments per document) |
| Output | secure index size (bytes); compression ratio (ratio of secure index size to document size) | |
| Constants | 1 secret  256 location uncertainty  0.001 false positive rate  1000 documents (documents/corpus) | |
| Testbed | machine A | |

In the previous experiment, we examined how page count affected secure index size while location uncertainty was held constant at 256. This had the effect of increasing the number of blocks per PSIB and BSIB as the page count increased. This motivates us to consider how the block count per PSIB and BSIB are affects secure index size.

In Figure 24, we see that the lines for PSIB and BSIB cross at ~47 blocks. If the primary metric of interest is secure index size (as measured by memory allocation size), this point of intersection represents a dividing line; to the left of the line PSIB is preferable, and to the right of the line BSIB is preferable. Note, however, that PSIB has other advantages that continue to exist regardless of the block count, e.g., query lag times.

In Figure 25, we see that for a small number of blocks, PSIB is a small fraction—a quarter—the size of the actual document. However, it grows linearly as the block count increases. On the other hand, BSIB converges (to a first approximation) to a constant factor of ~0.75 the size of the document as the block count increases.

A high block count is ideal for MinDist\* and BM25 MAP accuracy—it reduces the location uncertainty—but there is a trade-off between such accuracy and the amount of information leaked about the document.

PSIP does not represent a document as blocks; it represents a document as a postings list. Thus, location uncertainty can be adjusted to any desired value and PSIP’s file size (and query lag times) will remain the same.

Note that, as discussed later, it would be possible to use a more efficient representation for denoting which terms appear in which blocks. For example, since the bit vector becomes increasingly sparse as the block count increases, a compressed bit vector representation would result in significantly higher space efficiency at the expense of computational overhead.

### Documents (per corpus) vs Corpus Secure Index Size

Figure 26

|  |  |  |
| --- | --- | --- |
| Experiment Details | | |
| Input | documents (per corpus) |
| Output | corpus secure index size (bytes) | |
| Constants | 1 secret  16 pages (per document)  250 location uncertainty  0.001 false positive rate | |
| Testbed | machine A | |

For a reasonably large document consisting of 16 pages (4000 words, 250 words/page), we see that the average document is ~10.5 kilobytes for PSIB, ~16.8 kilobytes for PSIP, and ~23.3 kilobytes for PSIP. Note that, with that many pages and with that location uncertainty, the blocks per document is 16 for PSIB and BSIB; this is under the threshold of ~47 blocks under which PSIB is superior to BSIB.

For a corpus of nearly 20,000 documents, the total corpus size is a little over 200 MB; the original corpus was 593 MB.

### Pages vs Build Time

Figure 27

Figure 28

|  |  |  |
| --- | --- | --- |
| Experiment Details | | |
| Input | pages (~250 words/page) |
| Output | build time (milliseconds) | |
| Constants | 1 secret  256 location uncertainty  0.001 false positive rate  1000 documents (documents/corpus) | |
| Testbed | machine A | |

In this experiment, I am interested in seeing how page count (~250 words/page) affects secure index build time. The byte-size of the document is less important than its page count. BSIB is nearly twice as slow as PSIB and PSIB, but even BSIB is only 2.4 milliseconds per page.

None of the secure indexes are unreasonably slow; even a document consisting of ~300 pages takes only a fraction of a second to build. Indeed, the PSIB can build a ~800 page document in only a second. And, as discussed later, there are significant performance improvements that could be easily realized, e.g., replacing unnecessary cryptographic SHA256 re-hashes with non-cryptographic hash functions.

### Documents (per corpus) vs Build Time

Figure 29

|  |  |  |
| --- | --- | --- |
| Experiment Details | | |
| Input | documents (per corpus) |
| Output | corpus build time (milliseconds) | |
| Constants | 1 secret  16 pages (per document)  250 location uncertainty  0.001 false positive rate | |
| Testbed | machine A | |

The time to build a corpus consisting of reasonably large 16 page documents is 24 milliseconds per document for PSIB, 27 milliseconds for PSIP, and 36 milliseconds for BSIB.

In the previous experiment, we plotted page size vs build time. The line of best fit for PSIB build time ≈ 1.24 ∙ pages + 4.1; for PSIP, the line of best fit was build time ≈ 1.36 ∙ pages + 5.5; finally, for BSIB, the line of best fit was build time ≈ 2.4 ∙ pages - 2.4. Each one of these lines of best fit competently predicts the slopes in Figure 29.

### Pages vs Load Time

Figure 30

|  |  |  |
| --- | --- | --- |
| Experiment Details | | |
| Input | pages (~250 words/page) |
| Output | load time (milliseconds) | |
| Constants | 1 secret  256 location uncertainty  0.001 false positive rate  1000 documents (documents/corpus) | |
| Testbed | machine A | |

In this experiment, I am interested in seeing how page count affects secure index load time. Interestingly, PSIB (and to a greater extent although not shown here, PSIF) is nearly constant when representing documents from 1 page to 300 pages; 300 page documents take only 2.86 milliseconds to load raw from disk.

The other two perform less impressively. Concerning PSIP, I did not make much of an effort to optimize it. For instance, I load a term’s postings list as a vector of varints[[25]](#footnote-29), which incurs significant vector construction overhead as the number of terms in the document increases. A likely more efficient representation of a posting lists is a list consisting the word gaps between adjacent positions of a term. Since this list does not need to facilitate random access, i.e., operations on it can efficiently be performed on it in a sequential manner, the gaps may even be compressed, e.g., Huffman coded. Moreover, inspired by PSIB and its block-based approach, instead of storing the exact number of gaps N between adjacent positions of a term, store, where k is an integer denoting the block granularity size. And, of course, positions that map to the same block may be dropped to save even more space and processing time (but at the cost of a loss of frequency information, which may be desirable anyway).

For BSIB, as the document size increases, the overhead of de-serializing a larger number of Bloom filters may take a toll. However, the serialization seems fairly efficient.

### Documents vs Load Time

Figure 31

|  |  |  |
| --- | --- | --- |
| Experiment Details | | |
| Input | documents (per corpus) |
| Output | corpus load time (milliseconds) | |
| Constants | 1 secret  16 pages (per document)  250 location uncertainty  0.001 false positive rate | |
| Testbed | machine A | |

The time to load a corpus consisting of reasonably large 16 page documents is 0.26 milliseconds per document for PSIB, 2.13 milliseconds for PSIP, and 1.55 milliseconds for BSIB.

In the previous experiment, we plotted page size vs load time. The line of best fit for PSIB load time ≈ 2E-05 ∙ pages2 + 0.002 ∙ pages + 0.22. Plugging in pages = 16, we get a prediction of 0.26 milliseconds, which accurately matches the actual slope in Figure 31. The same is approximately the same for the other two secure indexes as well.

### Pages vs MinDist\* Lag Time

Figure 32

|  |  |  |
| --- | --- | --- |
| Experiment Details | | |
| Input | pages (~250 words/page) |
| Output | MinDist\* lag time (milliseconds) | |
| Constants | 1 secret  0 obfuscations  256 location uncertainty  0.001 false positive rate  2 terms/query  1 or 2 words/term | |
| Testbed | machine A | |

In this experiment, I am interested in seeing how page count affects MinDist\* lag time. BSIB is unique among the secure indexes in that MinDist\* lag time is linearly dependent upon page count; every additional page incurs ~0.0004 milliseconds.

For large documents (more specifically, for documents with a large number of block segments), BSIB performs poorly on this measure. This is the expected outcome. For a fixed location uncertainty, as the page count increases the document must be segmented into more blocks and therefore, because every block is assigned a Bloom filter, more Bloom filters must be queried (i.e., more hash functions must be evaluated). Since each Bloom filter hash function evaluation requires a constant amount of time, all query lag times—MinDist\* included—are dependent upon the number of hash functions that must be evaluated per document.

For a ~300 page document, the lag time is nearly 0.14 milliseconds. If the corpus consists of a million such documents, this operation would require nearly 140 seconds to complete. This is certainly impractical.

The PSI-based secure indexes, to a first approximation, take only a small constant amount of time with respect to page count. However, the constant—while small—will have scalability issue as corpus size grows to many thousands of documents. For instance, a corpus consisting of a million documents would require nearly ~20 seconds to complete. Even PSIP, the fastest secure index on this benchmark, would require ~7 seconds to complete the query.

None of them are below the 1 second mark for the million document example; one second response times are often considered the maximum delay a typical user will tolerate, and the network latency time is not even being factored into this measure. Of course, secure indexes do not represent the typical use case—the services provided by secure indexes are not free. Moreover, the slowed response times are far better than the alternative of downloading the entire corpus, decompressing the documents, and conducting local searches on them.

There are a two immediately obvious things that could significantly speed up the query operations like MinDist\*. First and foremost, each secure index in the database re-hashes the hidden query’s unigram and bigram terms with SHA256. While the re-hashing operation is desirable to ensure that a cryptographic hash of a term in one secure index looks nothing like the cryptographic hash of the same term in any other secure index, using SHA256 to perform the re-hashing is overkill; after all, the hidden query itself has already been transformed using SHA256.

According to my benchmarks, each evaluation of SHA256 takes ~0.0024 milliseconds on machine A. This is fairly significant; a single SHA256 hash around one-third the total time taken, on average, to complete a PSIP MinDist\* query (per secure index) consisting of two terms per query and one or two words per term. Moreover, ~0.0024 milliseconds is required for each hidden unigram or bigram in the hidden query. While the secure indexes short-circuit processing queries where appropriate (in this experiment consisting of two terms per query, they can at most avoid processing one term per query per document), it is clear that significant savings could be realized by using an orders-of-magnitude faster non-cryptographic hash function without any loss in confidentiality.

The second way to speed up query processing is through parallel programming techniques. Each query can be independently queried so this is an *embarrassingly parallel* problem and performance scales linearly with core count. Given N cores, PSIP MinDist\* query lag time would be ~0.0078/N milliseconds. It could complete the MinDist\* query operation against a million documents in less than a second given N=8 cores.

BSIB would require around N=140 cores to get under the one-second mark on a million documents consisting of ~300 pages per document. Using Bloom filters in the ways that have been previously proposed does not seem to be practical at scale. When we include some optimizations, like replacing the slow cryptographic SHA256 re-hash with a fast hash and incorporate memorization or caching (as discussed in **Background Information**), it may still work at scale in practice.

### Simulation: Location Uncertainty vs Location Error

Figure 33

|  |  |  |
| --- | --- | --- |
| Experiment Details | | |
| Input | location uncertainty |
| Output | average absolute location error | |
| Testbed | machine A | |

MinDist\*, discussed at length on page 36, relies on a secure index’s approximate location information. The less uncertain the location is, all things else being equal, the more accurate MinDist\* output will be (compared to the canonical index with perfect information).

However, if the location information is too precise, a hypothetical attacker will have more success at infer the contents of the document. Thus, the reported positions for a word must be uncertain—e.g., only reporting that a word falls within some range (block), as PSIB and BSIB do, or scrambling the positions in some random way, as PSIP does. See *simulating an attacker reconstructing documents from secure index information* on page 42 for a thorough analysis on this.

In this experiment, I am interested in observing the expected location error for block-based secure indexes and scrambled postings in which the scrambled positions of terms are randomly offset from their true position.

Figure 33 clearly shows that the block-based approach results in the greatest loss of accuracy. This was expected since each term is assigned a block range that is not centered on its true position (except for the term with a true position in the middle of the assigned block range). The other two (used by PSIP), centered\_uniform and centered\_triangular, do effectively do this by offsetting a term from its true position. They are named after the PDFs they sample their position offsets from: centered\_uniform uniformly samples an integral offset from , where is equal to the location uncertainty, and centered\_triangular samples from the triangular distribution with a mean and mode equal to the true position and a base also of length . The triangular distribution has a higher probability of being closer to the mean, so naturally it should have the least variance of the three.

The block-based approach has around 100% more error than the triangular approach for a given location uncertainty, and nearly 33% the error than the uniform approach. This seems fairly significant, especially in light of my analysis on page 42 which suggests that the block-based approach is also easier for the hypothetical attacker to compromise.

### Simulation: Location Uncertainty vs Minimum Pairwise Distance Error

Figure 34

|  |  |  |
| --- | --- | --- |
| Experiment Details | | |
| Input | location uncertainty |
| Output | average minimum pairwise distance error | |
| Testbed | machine A | |

In this simulation, instead of estimating the expected absolute location error for a term, I calculate the expected average minimum pairwise distance error. The error is now expected to be greater since there are two terms with uncertain locations instead of one, but the ratio of the errors between the three outputs are the same.

### Location Uncertainty vs MinDist\* MAP

Figure 35

Figure 36

|  |  |  |
| --- | --- | --- |
| Experiment Details | | |
| Input | location uncertainty |
| Output | MinDist\* MAP | |
| Constants | 1 secret  0 obfuscations  1000 documents (per corpus)  0.001 false positive rate  3 terms/query  1 or 2 words/term | |
| Testbed | machine A | |

As location uncertainty converges to 0, all of the secure indexes converge to the same MinDist\* MAP, which is approximately a score of 0.95. However, as location uncertainty increases, PSIP quickly diverges from the other two. These results reinforce the analysis in Simulation: Location Uncertainty vs Minimum Pairwise Distance Error.

Moreover, PSIB and BSIB do not scale well to large numbers of blocks (recall that location uncertainty is inversely proportional to the number of blocks in PSIB and BSIB). If the chosen parameters for MinDist\* flatten out its curve, e.g., MinDist\* relevancy curve is flattened s.t. it is made to be less sensitive to small changes in location uncertainty or the documents are reasonably small, PSIB and BSIB are still reasonable choices for MinDist\*. Otherwise they are not well suited to it compared to PSIP. However, note that they may effectively enable other search criteria, e.g., see Boolean proximity searching.

### Pages vs BM25 Lag Time

Figure 37

Figure 38

|  |  |  |
| --- | --- | --- |
| Experiment Details | | |
| Input | pages (~250 words/page) |
| Output | BM25 lag time (milliseconds) | |
| Constants | 1 secret  0 obfuscations  256 location uncertainty  0.001 false positive rate  2 terms/query  1 or 2 words/term | |
| Testbed | machine A | |

In this experiment, I am interested in seeing how page count affects BM25 query lag time. Pages vs BM25 lag time shows a similar pattern to pages vs MinDist\* lag time. However, note that BM25 is even slower. The reason for this is related to a technicality in the implementation of the BM25 algorithm: the implementation queries secure indexes twice for each query term; once for calculating the number of documents which contain a given query term, and once for calculating the frequency of the given query term per document. The implementation of the BM25 algorithm can be streamlined to only require a single frequency per query term request per document, rather than the two different queries used presently. Moreover, as mentioned in the section on caching, many of these computations can be memoized or cached. This could in practice be expected to save a significant amount of time.

As shown by Figure 37 and Figure 38, BM25 lag time for PSIB and PSIP are, to a first approximation, independent of page count.

Both MinDist\* and BM25 are scoring functions used to rank documents according to their relevance to a given query so that the results returned to the user can be ordered from most relevant to least relevant. As discussed in **Background Information** on MinDist\*, MinDist\* is most appropriately used as a way to give proximity sensitivity to established scoring measures like BM25. A straightforward way to do this, as discussed later, is to linearly combine them:

This calculation can be efficiently done with a single expensive operation used by MinDistScore—*GetLocations*. In light of convenient but inefficient implementation of BM25Score, it is plausible that this combined scoring function could be nearly as fast as *MinDistScore* by itself.

### Location Uncertainty vs BM25 MAP

Figure 39

Figure 40

|  |  |  |
| --- | --- | --- |
| Experiment Details | | |
| Input | location uncertainty |
| Output | BM25 MAP | |
| Constants | 1 secret  0 obfuscations  0.001 false positive rate  3 terms/query (Figure 39), 2 terms/query (Figure 40)  1 or 2 words/term  1000 documents (per corpus)  16 pages (per document) | |
| Testbed | machine A | |

In this experiment, I am interested in seeing how location uncertainty affects BM25 MAP. The results are not unexpected.

In Figure 39 and Figure 40, PSIP and PSIF track each other perfectly, as do PSIB and BSIB. PSIP and PSIF preserve, if desired, perfect frequency information (for unigrams and bigrams) for positive examples (false positives on negative examples are still possible), and this is independent of location uncertainty in this two secure index types.

However, PSIB and BSIB approximate a term’s frequency by counting how many of the blocks it appears in. The more blocks (the lesser the location uncertainty), the more accurately it approximates the true frequency. Indeed, in Figure 39, we see that as location uncertainty converges to 2, all of the secure indexes converge to the same BM25 score.

Also, notice that compared to Figure 39, Figure 40 is less accurate for a given location uncertainty for all of the secure index types. The reason for this has to do with the fact that Figure 39 has more terms per query, and thus when a document has all three of the terms the BM25 scoring algorithm is able to more accurately score the documents despite any frequency approximation errors. In other words, it seems to be the case that the more terms per query, potentially the less sensitive BM25 is to frequency approximation errors.

### Pages vs Boolean Lag Time

Figure 41

Figure 42

|  |  |  |
| --- | --- | --- |
| Experiment Details | | |
| Input | pages (~250 words/page) |
| Output | Boolean query lag time (milliseconds) | |
| Constants | 1 secret  0 obfuscations  256 location uncertainty  0.001 false positive rate  2 terms/query  1 or 2 words/term | |
| Testbed | machine A | |

As with every other output related to lag time, BSIB performs comparatively poorly. Boolean queries do not rank documents; a document is either relevant to the query (in this case, for a document to be relevant it must contain all of the terms in the query—a Boolean AND operation) or it is non-relevant. This is the quickest kind of query. (Indeed, the time required to perform the excessive SHA256 re-hashing operation takes up the most significant portion of time.)

PSIB, PSIP, and PSIF (PSIF is not shown but it tracks PSIP) can complete this operation in approximately 0.003 milliseconds, at least 0.0024 milliseconds of which is consumed by computing unnecessary SHA256 re-hashes. In other words, this is a ~500 nanosecond operation. This should allow a corpus of two million secure indexes to be searched by queries of this form in a second (not including the round-trip network delay).

With some of the other proposed performance enhancers, like caching and parallel computing, the simple Boolean query operation is extremely scalable. For documents of a typical length, e.g., less than 50 pages, even BSIB performs acceptably.

### Compression Ratio (Secure Index Size to Document Size) vs MinDist\* MAP

|  |  |  |
| --- | --- | --- |
| Experiment Details | | |
| Input | compression ratio: secure index size (bytes) to document size (bytes) |
| Output | MinDist\* MAP | |
| Constants | 1 secret  0 obfuscations  0.001 false positive rate  3 terms/query  1 or 2 words/term  1000 documents (per corpus) | |
| Testbed | machine A | |

In this experiment, I am interested in seeing how the ratio of the secure index size (in bytes) to the original document size (also in bytes) affects MinDist\* MAP accuracy.

PSIP has the highest unconditional MinDist\* MAP score. However, it does not come close to PSIB and BSIB in terms of having the smallest compression ratio. In fact, PSIP is a fairly constant size—the only way to change its size is by changing the false positive rate or by poisoning it. Depending on how the poisoning is done, it can be made smaller (e.g., replacing multiple positions with a single mean position in a postings list) or larger (adding positions in a postings list or adding fake terms). This is both a positive and a negative, as PSIP is nearly (in these examples) always the same fraction of the original document’s size but is consistently the highest performer on MAP accuracy and lag time.

# Future work

## experimental simulation of an adversary

## Compound set-theoretic queries

Compound set-theoretic queries are a simple extension of Boolean queries. Let the results (list of references) returned from Boolean search be sets, then the intersection (AND), union (OR), and complement (NOT) of them may be taken, e.g., .

All that a secure index needs to do in order to support set-theoretic query operators is to implement the trait. Thus, for each atomic query in a compound query, it will use this interface to retrieve the result set for it and then apply set-theoretic operations on the result sets to implement AND, OR, and NOT.

For instance, to implement , find all documents in the corpus that are relevant to the atomic query and then take the complement of this result set, i.e., only include a document (from the corpus) in the complement set if it is not in the result set.

Furthermore, all other set-theoretic operators, like set difference, can be expressed in terms of AND, OR, and NOT[[26]](#footnote-30), e.g., . Note that to support arbitrary set-theoretic operations, the language should be defined by a recursive grammar. This will allow for applying operators to nested results. For a company like Google, this is not necessarily a practical option since they must support millions of queries per second on billions of documents, but it may be a practical option for an Encrypted Search cloud server.

### Compound query grammar

An example of a recursive grammar, expressed in BNF notation, for compound queries is:

|  |  |  |
| --- | --- | --- |
|  |  |  |

## Fuzzy set-theoretic search

Fuzzy set-theoretic queries are an extension of classical (crisp) set-theoretic queries. Instead of operating on Boolean values, they operate on degree of membership values; a degree of membership value represents the degree (in the range ) to which something is a member of a set. At one extreme, a value equal to is equivalent to —e.g., a document is completely relevant to a query. At the other extreme, a value equal to is equivalent to —e.g., a document is completely irrelevant to a query.

Fuzzy operator equivalents to , , and are:

We can also apply hedges to degree of membership values. Hedges modify the degree of membership of a fuzzy value in a way that conforms to common intuition, e.g., a proposition can be somewhat true but not very true.

Indeed, somewhat and very are two examples of hedge functions. Let and . Then, for instance, , , iff . For that to be true it must be the case that or . Since . This means that, for a given q and K, must be larger for very(q)>K.

Note that reaches a maximum at and a minimum at , just as does. However, .

it is the furthest apart from identify(q) at q = 1/2. If our defuzzification test for whether a fuzzy membership value is true is, say, K = 1/2, then q is true whenever q > 1/2, but very(q) is true only when q > sqrt(1/2) = 1 / sqrt(2) ~ 0.7. This makes since because we are asking if "q is very true" not just "q is true".

Fuzzy set-theoretic queries may use the normalized output of any scoring algorithm or heuristic that does not simply output a binary score[[27]](#footnote-31). The non-binary output from BM25 and MinDist\* are obvious candidates for this. However, before they may be used, their output must be normalized s.t. the maximum value is and the minimum value is .

Assuming an appropriate normalizer has been found, return degree of membership values in the range [0, 1], we can apply fuzzy operators to the returned values.

## Letter n-grams and word n-grams

In the experiments, word unigrams and bigrams are atomic, indivisible units inserted into the secure indexes. However, this is not necessarily the case. One could go in either direction, either using larger word n-grams (e.g., inserting trigrams) to support faster searching on larger phrases, or smaller letter n-grams.

Consider the string "hello world". If storing letter trigrams, then the following transformation takes place, where \* denotes whitespace.

To search for the word “hello”, check for the existence of "hel", "ell", and "llo" in the set. If all three letter trigrams exist, that word is said to exist. As in the biword model, false positives are possible.

With letter n-grams, partial word matches are automatically possible. For instance, if the user wishes to find any words matching “ello”, then simply check for the existence of “ell” and “llo”. In fact, any substring that is three characters or larger can be matched.

## Wild-card searching

It is also possible to support wild-card searching, e.g., match any pattern “hello \* doctor”, where the asterisk represents a word wildcard. To support word wild-cards, in addition to inserting word unigrams and word bigrams, strings of the form “first \* third” must also be inserted, i.e., skip the word in the middle. For example:

This increases the size of the secure index fractionally while facilitating rapid word wildcard searching. This can, at increasing cost to size, be extended to support multiple word wildcards, e.g., “hello \* \* doctor”. Character wildcards can be supported by letter n-grams in the same way that word wildcards are supported by word n-grams. All of this, of course, may inflate the secure index beyond what is desirable.

## Approximate searching and error tolerance

Approximate searching is a superset of wild-card searching. Other ways approximate searching can be accommodated include hashing the terms of the document such that similar terms hash to the same value, e.g., locality sensitive hashing, phonetic coding (e.g., Soundex), or stemming[[28]](#footnote-32). Stemming reduces each word to its stem or root form, e.g., computer, computing, computes, computed, compute, computation, computations, and computational all map to the stem comput.

These kind of transformations generally take place during a preprocessing step so that the secure index constructor only sees these final transformations. Note that a preprocessor may include any combination of these transformations, i.e., it may include both the stem and the Soundex hash of a given word. Another form of approximate searching can be accomplished through query expansion, e.g., including synonyms, fixing spelling errors, and so forth.

However, these tricks—stemming, synonym expansion, etc—must be used with caution lest they come at too high a cost in terms of trading precision for recall. Ideally, both recall and precision can be improved since documents that are relevant to the user’s information need (as represented by the query) may be retrieved where otherwise they would not have been.

## Boolean proximity searching

In our experiments, we use MinDist\* as a way to rank-order documents according to a minimum pair-wise distance metric. However, another perhaps more useful—and far more straightforward—way to use the proximity information in secure indexes is to require that all of the terms in a query be within a minimum proximity of each other.

An algorithm to enable this functionality has already been essentially implemented for the MinDist\* heuristic. It is less complicated (both conceptually and computationally) than the MinDist\* scoring function. Furthermore, MinDist\* and/or BM25 can be used in tandem with Boolean proximity requirements, e.g., rank-order only those documents which contain all the terms in the query within a minimum proximity of each other.

## Semantic search

In section 6.3.3, semantic searching was discussed. In the context of Encrypted search, this can be implemented through standard natural language processing techniques.

For instance, if a user’s information need is represented by the query "carnivore hunting prey," one may assume he or she is also interested in more specific concepts, like "dog chasing cats" and "lions hunting antelopes". Using part of speech tagging, it can be determined that "carnivore" is the subject, "hunting" is the verb, and "prey" is the object. Using word-sense disambiguation, the word senses can be accurately determined, e.g., "carnivore" maps to "carnivore-1" (word sense 1 of carnivore in a dictionary). Using an ontology (like *WorldNet*), it can be determined that "carnivore-1" is a concept which includes more specific concepts like "dog-1" and "lion-2". Following this, we can expand the query “carnivore hunting prey” into a set of queries that better represents the information need:

{“carnivore-1 hunting-3 prey-4”, “dog-1 chasing-1 cat-1”, “dog-1 chasing-1 feline-2”, “lion-2 hunting-3 antelope-1”, …}

The original query has been expanded into a set of queries. This is a sophisticated example of query expansion. In addition, during the preprocessing stage of the secure index construction, similar techniques are used—e.g., instead of storing a biword model of “carnivore hunting prey,” store the word-sense disambiguated set of words {hunting-3, chasing-1, …}. And, using the biword model, instead of storing “hunting prey

## Topic searching

Using Bayes rule, and an assumption of independent, identically distributed unigrams, we have the following Naïve Bayes simplification:

Since the denominator is a constant given a document as evidence, it can be ignored when one is only interested in determining what the most likely topic is. Additionally, we can replace with the log of the conditional probability.

For example, one topic may be medical science. Operationally, sample unigrams from a medical science corpus to estimate and apply the formula. If cannot be estimated, a uniform distribution may be assumed (i.e., remove it from the formula).

Another type of search query can be articulated as, "How closely does a document of interest match a given secure index?" This is just a variation of topic searching in which the conditional probability, and .

Also, note that if the secure index contains bigrams, Markov chains of order may be used to model the true underlying distribution for a given topic with greater accuracy.

## Optimization: caching results

Caching previously calculated results could result in significant savings. For example, whenever a term in a Boolean search is mapped to a set of documents, store the mapping in a cache so that in subsequent Boolean searches involving the term may be serviced in near constant time.

I had initially planned to use an LRU cache to memoize computations like the above, but stripped the code out for the experiments for more predictable query lag times. In a practical implementation, a simple LRU cache or some other memorization technique may be used to avoid duplicating previous work. Since queries will be heavily biased towards a small subset of terms, this would result in significant savings.

Note that the CSP may technically do this without user permission; while it is a concern that a CSP may secretly collect such statistics, there may be little that can be done about it (with the exception of oblivious RAM-like techniques).

# References

|  |  |
| --- | --- |
| [1] | D. X. Song, D. Wagner and A. Perri, "Practical Techniques for Searches on Encrypted Data," *Proceedings of the 2000 IEEE Symposium on Security and Privacy,* no. Dawn Xiaodong Song, David Wagner, and Adrian Perrig, pp. 44-55, 2000. |
| [2] | G. Navarro, E. S. de Moura, M. Neubert, N. Ziviani and R. Baeza-Yates, "Adding Compression to Block Addressing Inverted Indexes," *Information Retrieval,* vol. 3, no. 1, pp. 49-77, 2007. |
| [3] | G. G. Langdon, "Huffman codes," 2000. |
| [4] | M. Mowbray, S. Pearson and Y. Shen, "Enhancing Privacy in Cloud Computing via Policy-Based Obfuscation," *Journal of Supercomputing,* vol. 61, p. 267–291, 2012. |
| [5] | C. T. a. D. L. Christian Collberg, "A Taxonomy of Obfuscating Transformations," 1997. |
| [6] | D. Hofheinz, J. Malone-lee and M. Stam, "Obfuscation for cryptographic purposes," *TCC,* pp. 214-232., 2007. |
| [7] | P. Golle, J. Staddon and B. Waters, "Secure Conjunctive Keyword Search over Encrypted Data," *Applied Cryptography and Network Security, Lecture Notes in Computer Science,* vol. 3089, pp. 31-45, 2004. |
| [8] | Z. Wei, Z. Dan-Feng, G. Feng and L. Guo-Hua, "On Indexing and Information Disclosure Measure for Efficient Cryptograph Query," *Proceedings of the World Scientific and Engineering Academy and Society International Conference on Computers,* pp. 476-480, 2009. |
| [9] | C. Dong, G. Russello and N. Dulay, "Shared and Searchable Encrypted Data for Untrusted Servers," *Data and Applications Security XXII, Lecture Notes in Computer Science,* vol. 5094, pp. 127-143, 2008. |
| [10] | M. R. Asghar, G. Russello, B. Crispo and M. Ion, "Supporting Complex Queries and Access Policies for Multi-User Encrypted Databases," *Proceedings of the ACM Workshop on Cloud Computing Security Workshop,* pp. 77-88, 2013. |
| [11] | J. Li, Q. Wang, C. Wang, N. Cao, K. Ren and W. Lou, "Fuzzy Keyword Search over Encrypted Data in Cloud Computing," in *Proceedings of IEEE INFOCOM*, 2010. |
| [12] | M. Celikik and H. Bast, "Efficient Fuzzy Search in Large Text Collections," *ACM Transactions on Information Systems,* vol. 31, no. 2, pp. 1-59, May 2013. |
| [13] | D. Boneh, G. D. Crescenzo, R. Ostrovsky and G. Persiano. , "Public-key encryption with keyword search," *Proceedings of Eurocrypt, Lecture Nodes in Computer Science,* May 2004. |
| [14] | W. Diffie and M. E. Hellman, "New directions in cryptography," 1976. |
| [15] | M. Naor and M. Yung, "Universal One-Way hash functions and their cryptographic applications," pp. 33-43, 1989. |
| [16] | E.-J. Goh, "Secure Indexes," *Trust, Privacy, and Security in Digital Business, Lecture Notes in Computer Science,* vol. 3592, pp. 128-140, 2005. |
| [17] | B. P. a. T. Reinman, "Oblivious RAM revisited". |
| [18] | A. Swaminathan, Y. Mao, G.-M. Su, H. Gou, A. Varna, S. He, M. Wu and D. Oard, "Conﬁdentiality-Preserving Rank-Ordered Search”". |
| [19] | S. Artzi, C. Newport and D. Schultz, "Encrypted keyword search in a distributed storage system". |
| [20] | A. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey," *Internet Mathematics,* vol. 1, no. 4, pp. 485-509, 2002. |
| [21] | N. Cao, C. Wang, M. Li, K. Ren and W. Lou, "Privacy-Preserving Multi-keyword Ranked Search over Encrypted Cloud Data," *Proceedings of IEEE INFOCOM,* April 2011. |
| [22] | Y.-c. Chang and M. Mitzenmacher, "Privacy preserving keyword searches on remote encrypted data," *Lecture Notes in Computer Science,* vol. 3531, pp. 442-455, 2005. |
| [23] | Q. Liu, G. Wang and J. Wu, "An Efficient Privacy Preserving Keyword Search Scheme in Cloud Computing," *Proceedings of International Conference on Computational Science and Engineering,* vol. 2, pp. 715-720, August 2009. |
| [24] | H. Cao, D. Jiang, J. Pei, E. Chen and H. Li, "Towards Context-Aware Search by Learning a Very Large Variable Length Hidden Markov Model from Search Logs," *Proceedings of the International Conference on World Wide Web,* pp. 191-200, 2009. |
| [25] | F. Giunchiglia, U. Kharkevich and I. Zaihrayeu, "Concept Search: Semantics Enabled Syntactic Search," *Proceedings of CEUR Workshop,* 2008. |
| [26] | R. A. Baeza-yates, "Text retrieval: Theory and practice," *In 12th IFIP World Computer Congress,* vol. I, pp. 465-476, 1992. |
| [27] | C. Buckley and G. Salton, "Term-weighting approaches in automatic text retrieval," *INFORMATION PROCESSING AND MANAGEMENT,* vol. 24, pp. 513-523, 1988. |
| [28] | H. Cao, D. H. Hu, D. Shen, D. Jiang, J.-T. Sun, E. Chen and Q. Yang, "Context-Aware Query Classification," *Proceedings of the International ACM SIGIR Conference on Research and Development in Information Retrieval,* pp. 3-10, 2009. |
| [29] | P. Indyk and R. Motwani, "Approximate nearest neighbors: Towards removing the curse of dimensionality," pp. 604-613, 1998. |
| [30] | Y. Hua, B. Xiao, B. Veeravalli and D. Feng, "Locality-Sensitive Bloom Filter for Approximate Membership Query," *IEEE Transcations on Computers,* vol. 61, no. 6, pp. 817-830, 2012. |
| [31] | R. Krovetz, "Viewing morphology as an inference process," pp. 191-202, 1993. |
| [32] | R. Brinkman, P. Hartel, W. Jonker and C. Bösch, "Conjunctive Wildcard Search over Encrypted Data," *Proceedings of the VLDB International Conference on Secure Data Management,* pp. 114-127, 2011. |
| [33] | Y. Tang, D. Gu, N. Ding and H. Lu, "Phrase Search over Encrypted Data with Symmetric Encryption Scheme," *2012 32nd International Conference on Distributed Computing Systems Workshops (ICDCSW),* pp. 471-480, 2012. |
| [34] | A. Kumar, J. J. Xu, J. Wang and L. Li, "Space-Code bloom filter for efficient traffic flow measurement," *Proceedings of the 2003 ACM SIGCOMM conference on Internet measurement,* pp. 167-172, 2003. |
| [35] | R. Schenkel, A. Broschart, S. Hwang and M. Theobald, "Efficient Text Proximity Search," *Lecture Notes in Computer Science,* pp. 287-299, 2007. |
| [36] | T. Tao and C. Zhai, "An Exploration of Proximity Measures in Information Retrieval," in *SIGIR '07 Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval Pages 295-302*, 2007. |
| [37] | Z. Kissel and J. Wang, "Verifiable Symmetric Searchable Encryption for Multiple Groups of Users," *Proceedings of SAM 2013,* 2013. |
| [38] | C. Gentry, "Fully homomorphic encryption using ideal lattices," *Proc. STOC,* pp. 169-178, 2009. |
| [39] | Q. Liu, G. Wang and J. Wub, "Secure and Privacy Preserving Keyword Searching for Cloud Storage Services," *Journal of Network and Computer Applications,* vol. 35, no. 3, pp. 927-933, May 2012. |
| [40] | S. Pearson, Y. Shen and M. Mowbray, "A Privacy Manager for Cloud Computing," *Cloud Computing, Lecture Notes in Computer Science,* vol. 5931, pp. 90-106, 2009. |
| [41] | Y. Lu and G. Tsudik, "Enhancing Data Privacy in the Cloud," *Trust Management V, IFIP Advances in Information and Communication Technology,* vol. 358, pp. 117-132, 2011. |
| [42] | R. Latif, H. Abbas, S. Assar and Q. Ali, "Cloud Computing Risk Assessment: A Systematic Literature Review," *Future Information Technology, Lecture Notes in Electrical Engineering,* vol. 276, pp. 285-295, 2014. |
| [43] | S. Mehrotra, C. Li, B. Iyer and H. Hacigümüş, "Executing SQL over Encrypted Data in the Database-Service-Provider Model," *Proceedings of the ACM SIGMOD International Conference on Management of Data,* pp. 216-227, 2002. |
| [44] | B. Zhu, B. Zhu and K. Ren, "PEKSrand: Providing Predicate Privacy in Public-Key Encryption with Keyword Search," *Proceedings of IEEE International Conference on Communications,* pp. 1-6, 2011. |
| [45] | R. Curtmola, J. Garay, S. Kamara and R. Ostrovsky, "Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions," *Proceedings of the ACM Conference on Computer and Communications Security,* pp. 79-88, 2006. |
| [46] | B. Xiang, D. Jiang, J. Pei, X. Sun, E. Chen and H. Li, "Context-Aware Ranking in Web Search," *Proceeding of the International ACM SIGIR Conference on Research and Development in Information Retrieval,* pp. 451-458, 2010. |
| [47] | Y. Shen and J. Yan, "Sparse Hidden-Dynamics Conditional Random Fields for User Intent Understanding," *Proceedings of the International Conference on World Wide Web,* no. Shuicheng, Lei Ji, Ning Liu, Zheng Chen, pp. 7-16, 2011. |
| [48] | J. Chen, H. Guo, W. Wu and W. Wang, "iMecho: a Context-Aware Desktop Search System,” Proceedings of the International ACM SIGIR conference on Research and development in Information Retrieval," pp. 1269-1270, 2011. |

1. Encrypted Search may be considered a subset of the far more computationally demanding field of fully homomorphic encryption (FHE) [2]. [↑](#footnote-ref-1)
2. Since users must already maintain a separate symbol-mapping table, they could just query this structure instead. [↑](#footnote-ref-2)
3. [↑](#footnote-ref-3)
4. In this naïve scheme, only can search the encrypted document; in more sophisticated approaches, multi-user encrypted searching schemes are possible. [↑](#footnote-ref-4)
5. Given a hash, it should be difficult to find an input for the hash function that outputs the given hash. [↑](#footnote-ref-5)
6. See section 3.3.3 on one-way hash functions [↑](#footnote-ref-6)
7. Note that the reference to a confidential document can be encrypted to further mitigate leaking information about the document’s contents. [↑](#footnote-ref-7)
8. Unless approximate searching or error tolerance is allowed. See section 9.6. [↑](#footnote-ref-8)
9. Given the space complexity of this approach, it may be more useful to include only probable typographical errors. [↑](#footnote-ref-9)
10. Moreover, these field names—although not the actual values—are revealed to adversaries (information leak). [↑](#footnote-ref-10)
11. The untrusted server could store the results of queries to learn more about the contents of documents over time, but this information should be both (1) approximate (e.g., false positives on terms existing in documents) and (2) incomplete. See section 4 for more information. [↑](#footnote-ref-11)
12. Semantic search in the context of Encrypted Search may have an additional advantage: remove as many specifics as possible from the secure index but include its more general concepts. This may both improve relevancy of search results while erasing potentially compromising specifics. [↑](#footnote-ref-12)
13. A directed acyclic graph (e.g., a chain) of proxies may be used to mitigate the risk of collusion, but this introduces significant overhead. [↑](#footnote-ref-13)
14. Note that the sequence of trapdoors (cryptographic hashes) transmitted to the leaks significantly more information than the secure index representation since it is a simple substitution cipher. Thus, must be reasonably trusted. Since the will not be used nearly as frequently as the , which can be less trustworthy, the may be more tightly controlled without as much cost. [↑](#footnote-ref-14)
15. Default implementation uses SHA256 and non-invertibly maps the hashes to N hexadecimal digits. [↑](#footnote-ref-15)
16. Limits statistical inference to sampling from a single secure index rather than an entire corpus of secure indexes since each secure index has a unique and random way of mapping its unigrams and bigrams to hashes. [↑](#footnote-ref-17)
17. It is one of the more vulnerable parts of the system because it is more susceptible to substitution cipher attacks; the secure index itself has many safe-guards against such attacks: (1) each secure index uniquely hashes its members with a salt such that the same unigram or bigram in one secure index will look completely different than in any other, (2) each Psi-based secure index maps every string in the universe of strings to a much small bit string (e.g., 10 bits), (3) frequency and location information is only approximate, and (4) entries in the secure index may be fake. [↑](#footnote-ref-21)
18. Multiple secrets in this case becomes an attack vector, since [↑](#footnote-ref-22)
19. Note that this assumes the attacker has access to the hidden query stream. If the hidden query steam is taking place over a secret channel, the secure index server and the attacker must share information to make this an effective kind of attack. [↑](#footnote-ref-23)
20. Since order is irrelevant (i.i.d. distribution), the actual probability is , where ki represents how many times ti appears in the query history set. However, is a constant for a given history of n and k, so we can safely ignore it in our maximum likelihood attack. [↑](#footnote-ref-24)
21. When simulating the attacker, the log of the maximum likelihood will be used instead. [↑](#footnote-ref-25)
22. Actually, a secure index using a biword search model stores the unigrams (words) and bigrams in a document. [↑](#footnote-ref-26)
23. It is also designed to provide more accurate location information by allowing the mean error of approximate word positions to be 0—i.e., PSIP changes (or at least it is possible) each words position about its true mean. [↑](#footnote-ref-27)
24. This may take the form of the user’s client sending fake queries per real query, where is a discrete random variable, or it may consist of a fake query bot providing a plausible flow of fake queries independent of real queries. [↑](#footnote-ref-28)
25. A way of storing small integers in fewer bytes. This is sort of a cheat, since in general I attempted to ensure size(secure index data structure in memory) ~ size(secure index serialization on disk), but a varint is converted into an unsigned integer once loaded into memory. However, the cheat is justified in that it is designed to mimic a more efficient solution that does meet the size(serialization) = size(secure index data structure) objective. [↑](#footnote-ref-29)
26. Technically, only AND and NOT (or OR and NOT) are needed, but for computational efficiency both AND and OR should be efficiently supported s.t. they may be short-circuited as early as possible. Indeed, this may justify implementing additional operators, like DIFFERENCE, without reducing them to combinations of AND, OR, and NOT. [↑](#footnote-ref-30)
27. Fuzzy set-theoretic queries reduce to classical set-theoretic queries if the scoring algorithm only outputs binary scores. [↑](#footnote-ref-31)
28. Stemming, as a preprocessing step during secure index construction, has already been implemented. However, it was not enabled for any of the tests. [↑](#footnote-ref-32)