SIUE Department of Computer Science

Encrypted Search

Thesis Proposal

Hiroshi Fujinoki, Alex Towell
3-10-2014

ADVANTAGES OF CLOUD STORAGE

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

- **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 (CSP) are efficient (division of labor).
- **Accessible**. Storage can be accessed anytime and anywhere.
- **Sharable**. Each resource (e.g., directory or file) 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 a document's confidentiality when it is hosted *in the cloud*. 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 through the use of 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.

Unfortunately, this approach breaks down if an organization has a large collection of sensitive documents. It is inefficient having to download the collection of encrypted documents before being able to search them. Moreover, the larger the collection of sensitive documents, the less efficient it becomes. This inefficiency is especially evident on resource-limited machines, e.g., smartphones with limited bandwidth—*slow downloads*—and limited power—*decryption is computationally demanding*.

Efficient solution to confidentiality

What is sought is some way to allow the CSP to search the encrypted documents on behalf of clients, 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 (*hidden queries*). In other words, the CSP should be able to perform *oblivious* searches on behalf of users. Additionally, the CSP should not be able to initiate meaningful searches except on behalf of users.

ENCRYPTED SEARCH

The ability to search over a collection of encrypted documents without needing to decrypt them first is known as *encrypted search*. In light of the advantages of cloud storage, encrypted search has gained a lot of traction in the research community.

OUR RESEARCH

See the document on existing research for an overview on the current state of affairs in *encrypted search*. Our research will try to contribute to this work in a few different ways.

Overview of secure index construction

Encrypted search will be facilitated by a *secure index.* To make an encrypted document searchable, a client must construct a secure index for it. The index will only allow someone who has secret information to meaningfully query it.

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) The *document processor* encrypts the document using whatever cryptographic algorithm is deemed appropriate.

(3) First, the *document processor* generates *searchable terms* for the document. Terms can be anything; the most basic terms are the words contained in the document (keyword search). Second, it concatenates a document id and *secret* to each term. Third, the concatenation is fed into a cryptographic one-way hash function. Finally, it generates a list of private searchable terms from the output of the one-way hash function and transmits the result to a *proxy indexer*.

(4) First, the *proxy indexer* concatenates a *secret* to the private terms and feeds them into a one-way hash function. Second, the proxy concatenates the document's id 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 its corresponding encrypted document in a database to facilitate efficient encrypted searches in response to client queries.

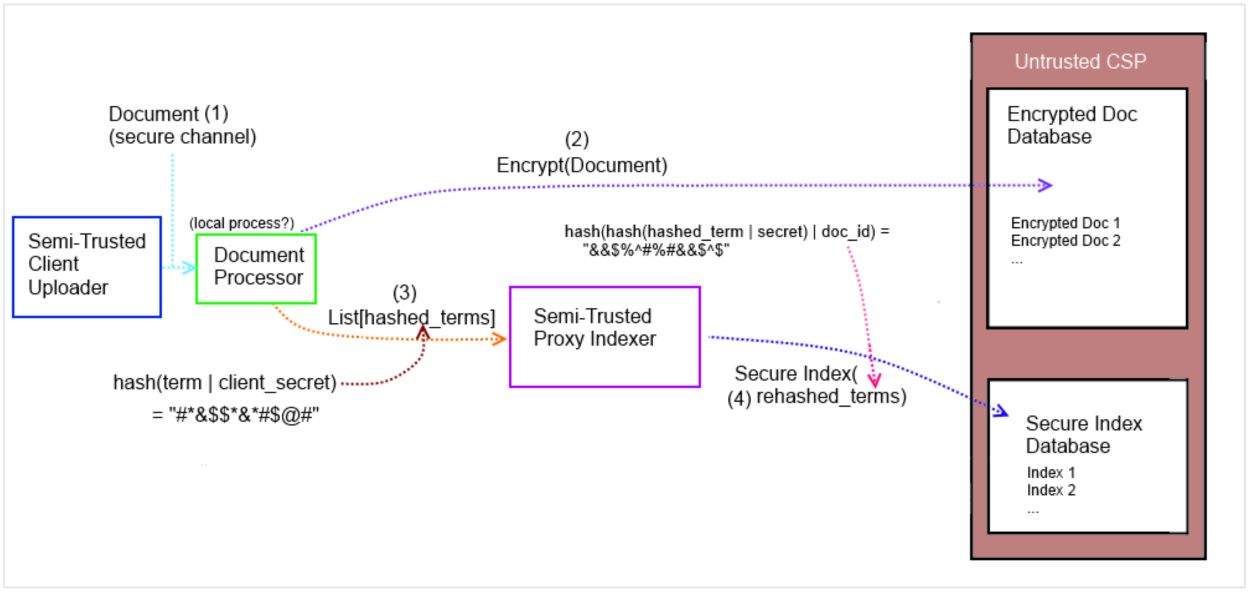

Figure 1

ADDITIONAL NOTES

The motivation for partitioning *secure index* construction into two separate stages, a *document processor* and *proxy indexer*, is to enable user authorization management—namely, user revocation. Assuming the proxy and users do not collude, neither the proxy nor the users will be able to meaningfully query secure indexes, even if they downloaded copies to their local machines. Specifically, users are only provided with partial secrets. To search a secure index, this information must be cryptographically mingled with the partial secrets provided to the proxy. Neither the users nor the proxy have sufficient information on their own.

The reason for the second hash in (3) is to make is to that the same term in one secure index will appear differently than the term in another secure index to prevent correlations between secure indexes from being exposed.

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. (In that case, **Error! Reference source not found.** is significantly simplified.)

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.

Overview of encrypted search

Assuming the *secure indexes* for a collection of sensitive documents have been constructed and transmitted to the CSP, encrypted search proceeds as follows. Refer to **Error! Reference source not found.** 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).

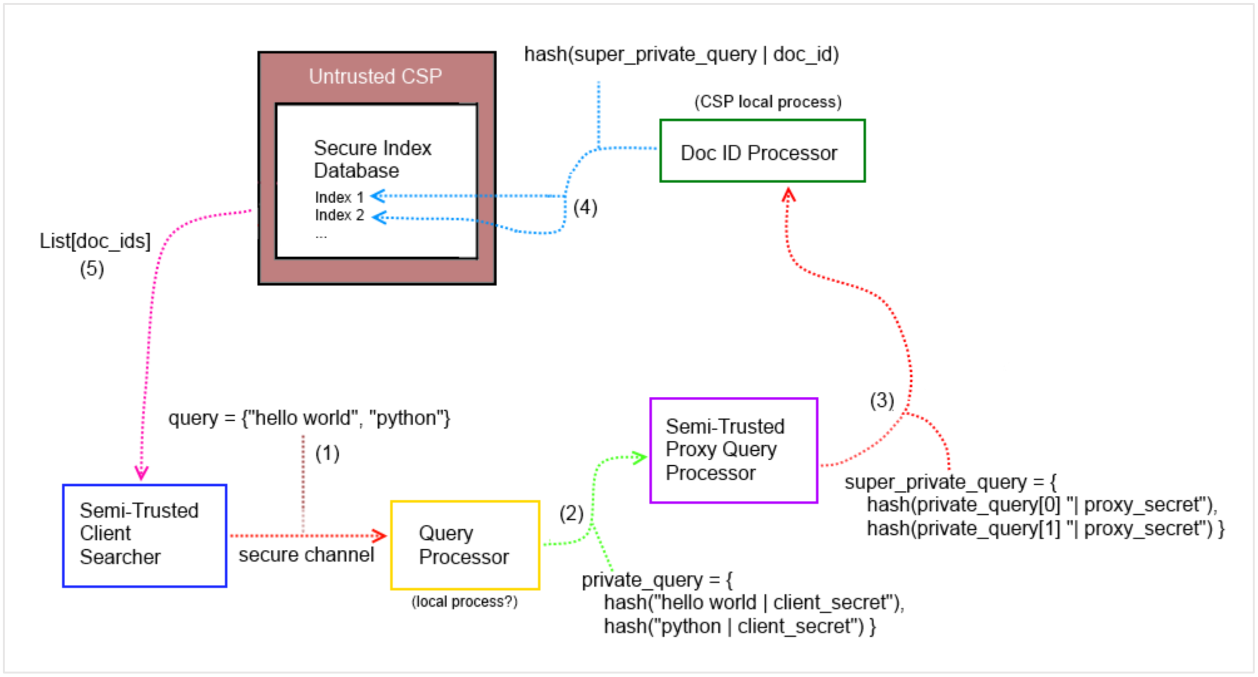
Untrusted CSP
Secure Index Database
Index 1
Index 2
...
hash(super_private_query | doc_id)
(CSP local process)
Doc ID Processor
(4)
List[doc_ids]
(5)
query = {"hello world", "python"}
(1)
Semi-Trusted Client Searcher
secure channel
Query Processor
(local process?)
(2)
Semi-Trusted Proxy Query Processor
(3)
super_private_query = {
hash(private_query[0] "| proxy_secret"),
hash(private_query[1] "| proxy_secret") }
private_query = {
hash("hello world | client_secret"),
hash("python | client_secret") }

Figure 2

ADDITIONAL NOTES

The prospect of constructing secure indexes for the entire collection of documents is tempting. This *master index* would allow the CSP to avoid the cost of iterating over all of the secure indexes in the database. However, there are a few concerns. First, the purpose of the *Doc ID Processor* (see *proxy processor* in the previous section) is to reduce aggregate patterns. The master index, in spirit, conflicts with this objective.

However, the master index is a far more approximate (document-level instead of block-level) and uncertain (higher false positive rate on queries) data structure, so it leaks very little information. Second, the master index may be quite large, although its size can be reduced by allowing high false positive rates on queries; after all, the master index is only used to narrow down the list of candidate secure indexes.

Saving the results of previous hidden queries can also reduce the cost of searching. For example, a least recently used (LRU) cache of hidden query, ranked result pairs could conceivably service a sizable fraction of queries. Note that the CSP can do this without technically requiring any permissions from the customer; while it is a concern that a CSP may collect hidden query statistics, there is little that can be done about it (with the exception of oblivious RAM-like constructions or homomorphic encryption; more on the former later).

Secure Offline Indexes

INVERTED INDEXES

To facilitate efficient searching over large collections of documents, offline search indexes must be constructed. A popular choice in information retrieval (IR) is the inverted index. Unfortunately, this data structure leaks too much information. Consequently, alternative indexes are being explored.

BLOOM FILTERS

A particularly interesting choice is the Bloom filter, which support set membership queries on an approximation of the set (i.e., they permit false positives—but no false negatives). They are interesting because (1) they are reasonably efficient, (2) they can trade accuracy (false positive rate) for space, and (3) they leak very little information about the documents they represent.

However, as promising as the Bloom filter is, it does not achieve optimal space efficiency for a given false positive rate. The optimal space efficiency is $-n\ log_2 \varepsilon$, where ε is the false positive rate, but the Bloom filter has a space complexity of $-1.44n\ log_2 \varepsilon$. Moreover, while the time complexity is *O(1)*, the Big-Oh notation is hiding a large coefficient.

PERFECT FILTER

In place of the Bloom filter, we propose the **Perfect filter** (*Pf*). As far as we can determine, this approach has not been explored in encrypted search. It has two primary advantages over Bloom filters: optimal space efficiency and speed.

A *Pf* is constructed by combining a minimum perfect hash with a cryptographic (one-way) hash. 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 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 $k = -\frac{\ln p}{\ln 2}$, 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 $p' = 1 - (1 - \mathrm{p})^r$. 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 *p* = 0.001 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*.

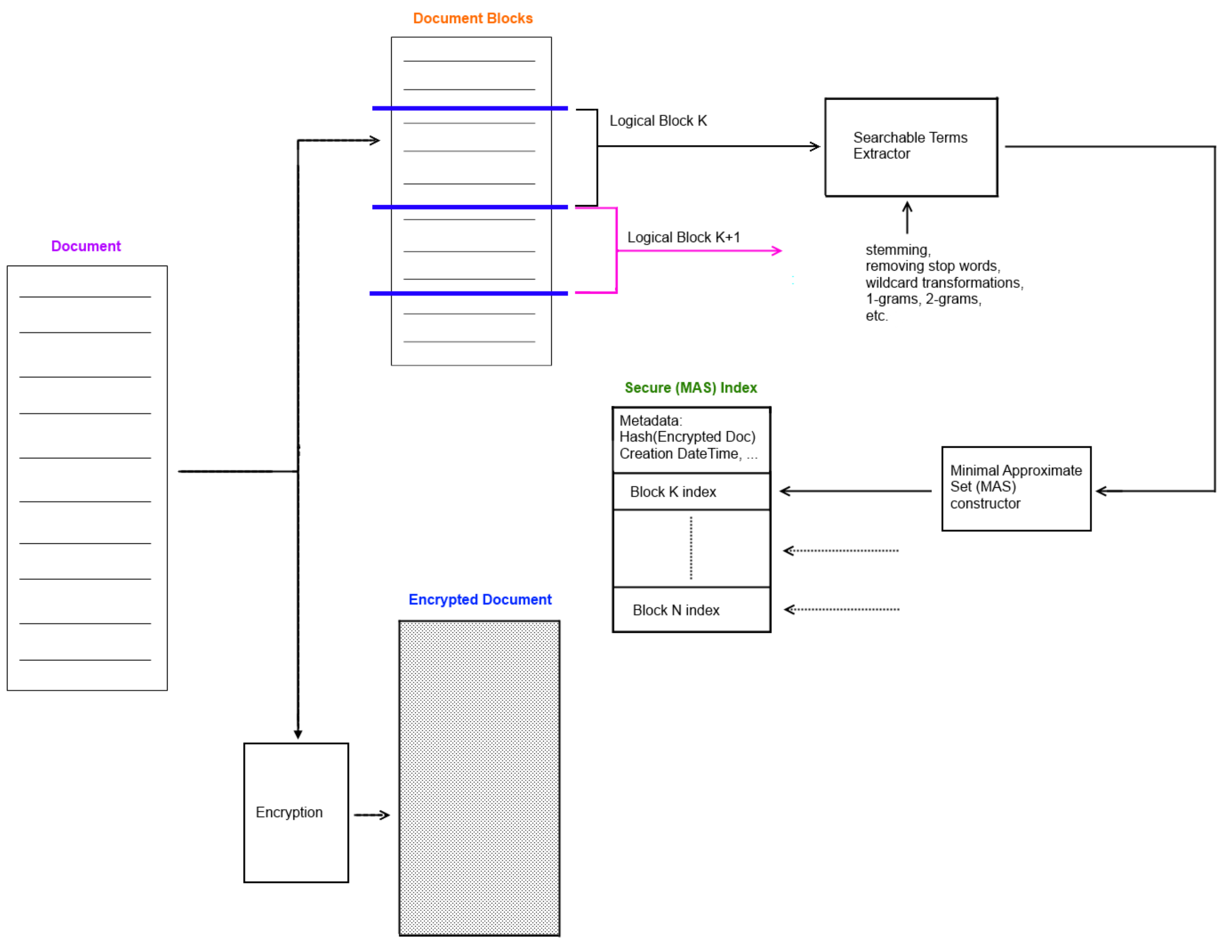
Document Blocks
Logical Block K
Searchable Terms Extractor
Logical Block K+1
Document
stemming, removing stop words, wildcard transformations, 1-grams, 2-grams, etc.
Secure (MAS) Index
Metadata: Hash(Encrypted Doc) Creation DateTime, ...
Block K index
Block N index
Minimal Approximate Set (MAS) constructor
Encrypted Document
Encryption

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 $\varepsilon = \frac{1}{2^m}$, 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 for whatever reason (e.g., for more accurate keyword weighting metrics) 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 2^r 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.

It should be noted that, while in theory the perfect hash has optimal space complexity given a false positive rate, implementations in practice do not achieve this limit. In practice, the space-complexity savings over the Bloom filter are less significant, although the reduced time complexity is still a big selling point, especially in the context of encrypted search in which an entire database of documents may need to be queried.

Relevancy metrics

Encrypted search efforts have (with recent exceptions) ignored the thorny though necessary (for commercial appeal) task of trying to match queries to relevant sets of documents. That is, ranking documents by a measure of how “close” they are to matching a user’s search hidden query. In the field of information retrieval, finding effective ways to measure relevancy is probably considered their fundamental objective. To that end, they have devised many clever algorithms and heuristics to rank documents by their estimated relevancy to a query. We will explore enabling some of these relevancy measures within our secure index construction (discussed next)—namely term proximity and keyword weighting (e.g., tf-idf) heuristics to improve accuracy of relevancy scores.

TERM PROXIMITY

To calculate term proximity, words are mapped to a set of blocks (rather than a set of positions). Thus, the index quickly and efficiently answers the question, *in what blocks does a word occur in?* An approximate proximity score can be calculated by determining which blocks the terms of a query reside in combined with the size of the respective blocks.

KEYWORD WEIGHTING

Keyword weighting is based on two insights. First, some of the terms occur more frequently in a document than other terms. Thus, when scoring the relevancy of a document, if a frequent term in the document matches a term in the query, it should be given more weight than a less frequent but matching term.

$$term\_weight\_in\_document(t, d) = some\ function\ on\ the\ frequency\ of\ term\ t\ in\ d$$

Term frequency can either be approximate, e.g., how many blocks does it occur in, or exact if an index which supports multiplicities is used. Since the Perfect filter (*Pf*) trivially supports multisets, multiplicity information may be explicitly stored in the secure index if desired.

Second, terms that appear less frequently in the collection of documents have more *discriminatory* power. For example, the word ***the*** is in nearly every document—it serves as linguistic glue— but the word ***acatalepsy*** will be found in very few, if any, documents. The more discriminatory power a term has, the more weight it should be given when scoring a document's relevancy.

$$term\_weight\_in\_collection(t, D) = f\left(\frac{\|D\|}{\|\{t \in d, d \in D\}\|}\right)$$

A long-established keyword weighting heuristic, *tf-idf* (term frequency, inverse document frequency), will be adopted. In tf-idf, both the term frequency within a document and the inverse document frequency within a collection of documents are used to estimate how important a query term is.

$$tf\_idf(t, d, D) = tf(t, d) \times idf(t, D) = f(t, d) \times \log\left(\frac{\|D\|}{\|\{t \in d, d \in D\}\|}\right)$$

Where *f* is the estimated frequency of term t in document d. A straightforward implementation that uses this heuristic is to simply sum the weights of each query term found in the document.

Information leaks

Even if an encrypted search scheme provides robust *data confidentiality* and *hidden queries*, access patterns and implicit information useful for statistical inference may still be leaked. For instance, if the same term in a private query always appears the same, an adversary can build up an *encrypted term* to *encrypted document* table. If the adversary has another distribution (a simple text corpus) that models the frequency of plaintext queries, then he or she can assign the encrypted terms to plaintext terms in the distribution in a way that maximizes the likelihood of the assignment, e.g., if the most common word is "apple" then it is likely that "apple" should be assigned to one of the encrypted query terms that has a high frequency to maximize the likelihood. (There are simpler examples, however, e.g., whenever "Bob" does a specific encrypted query, he usually follows that up with a look at some stocks.)

With the exception of a paper on oblivious RAM (which is too costly), no encrypted search program sought to mitigate the risk from this subtler form of information leakage. To try to remedy this, experiments will be conducted to mitigate this kind of information leak. In particular, a single query term may have up to have N independent random bit string representations such that a client could issue the same query term N times and the CSP would see a different hidden query term each time (but the user would get the same relevant documents back). This complicates the CSP's task of generating statistical data about frequencies of hidden query patterns

One approach to this problem is that, for each term, have the proxy indexer hash it with different known concatenations, e.g., repetitions of the secret. However, this approach will consume more space.

Experimental design

To be determined.