Proposal

Motivation for our work: Individuals and organizations would like to use untrusted cloud services to host their sensitive data so that they can access it anytime, anywhere. But how do we secure the data while making it easy for them to search through it?

The naive solution is for users to download their encrypted documents from the cloud and decrypt it on their local, trusted machines. And then perform local searches to find relevant documents. However, this is not an ideal solution. For instance, if users are on devices with limited resources (e.g., bandwidth, storage capacity, etc.), it is unreasonable to expect them to download the entire set of encrypted documents. It becomes especially unreasonable as the volume of sensitive information continues to grow.

What is needed is some way to allow the CSP to search the encrypted documents on behalf of clients, returning only those documents relevant to user queries. Furthermore, this should be done without revealing the contents of documents nor queries. In other words, the CSP should be able to perform oblivious searches on behalf of users.

The ability to search over an encrypted collection of documents without needing to decrypt them first is known as encrypted searching. Encrypted searching has gained a lot of traction in the research community because of its obvious and timely utility.

Our research hopes to contribute to its development in a few different ways.

First, the encrypted searching community has, generally (with recent exceptions), washed over the issue of matching queries to relevant sets of documents. In the field of information retrieval, relevancy is paramount. To that end, they have devised all sorts of 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) relevancy metrics.

To enable term proximity, we can adopt the approach seen in the IR community where they map words to a set of blocks instead of a set of positions, thus the index can quickly answer the question, “in what blocks does this word occur in?” We can use this to provide an approximate proximity score by seeing which blocks the terms of a query are in and using the size of the blocks to estimate how far apart they are.

We will try different approaches to keyword weighting. In tf-idf, both the term frequency within a document and the inverse document frequency within a collection of documents are used. Term frequency is simply how many times the particular term occurs in a document. This can be either be approximate, e.g., how many blocks does it occur in, or exact if we use an index which supports multiplicities. More on that in a bit. Inverse document frequency is just the total number of documents in the collection divided by the number of documents containing the term. This poses some challenges to encrypted searching in the context of information leaking and speed, so we are curious to explore this relationship further.

Second, to facilitate rapid and efficient searches on large collections of documents, indexes of the documents are usually created. A popular one in IR is the inverted file index, but we contend this construction leaks too much information about the encrypted documents. In the encrypted searching community, different indexes have been explored. A particularly interesting index is based on Bloom filters, which support set membership queries on the approximation of the (they permit false positives, but no false negatives). They are interesting because (1) they are reasonably efficient—*O(1)*)—and (2) they can trade accuracy (probability of false positives) for space.

However, as promising as the Bloom filter is, it does not possess theoretical optimal space efficiency for a given false positive rate. The optimal space efficiency is , where ε is the false positive rate, but the Bloom filter is a factor of ~1.44 of that.

So, we will explore the use of what we call a “Minimum Approximate Set” (MAS). This has, as far as have been able to determine, not been explored by anyone in the field of encrypted searching. It has a number of advantages: it ha, theoretically, optimal space efficiency, and it is very fast.

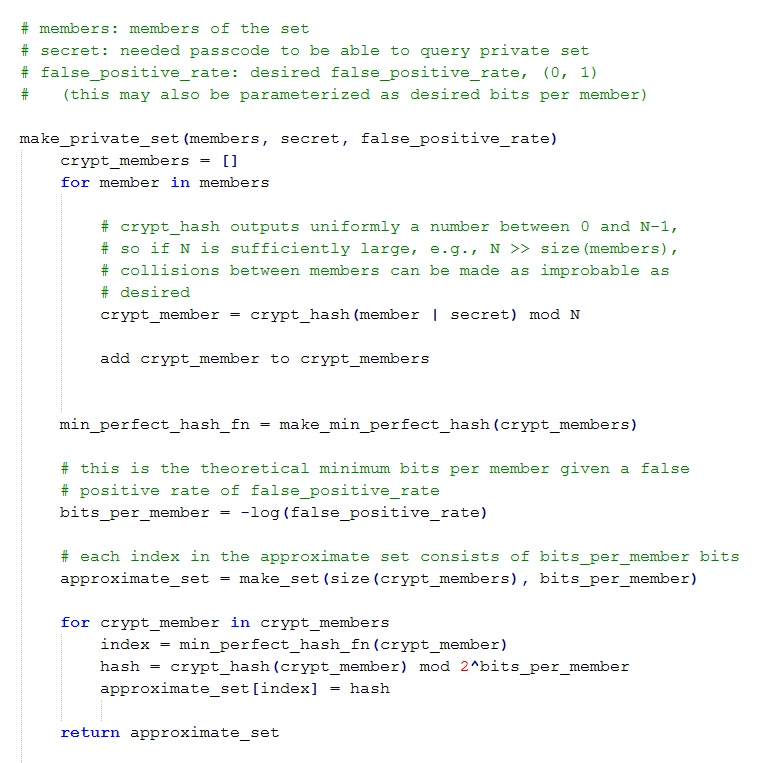
It is constructed by combining a minimum perfect hash with a cryptographic (one-way) hash. Like the Bloom filter, it can trade accuracy for space-complexity, but it does so in a theoretically optimal way. Perhaps more importantly (since a search query may be submitted against an arbitrary number of documents), MAS only requires evaluating two hash functions. Consequently, it has a significant advantage over Bloom filters. In a Bloom filter, the number of hash functions (that minimizes the false positive rate) is, where p is the desired false probability rate (this equation implicitly depends on m and n, where m is size of bit vector and n is cardinality of set). For instance, if p is 0.001—1 out of 1000 false positives—then k = 10.

Furthermore, if conjunctive queries are supported, then the false positive rate for a conjunction of k queries is . For example, if each independent query has a false positive rate of p = 0.001, then for a conjunction of 10 queries, the false positive rate is 0.01. A similar analysis can be done for disjunctions. Thus, even though a false positive rate of 0.001 may seem quite small, in practice it may not be sufficient.

If the ability to count multiplicities is desired for the keyword weighting relevancy measures, this is easily 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 multiplies of terms in an encrypted document, these four discrete values can either map to direct frequency counts from 1 to 4, or they can map to four ordinal values, e.g., {“only one”, “between two and four”, “between five and ten”, “more than ten”}. We can use this to inform the relevancy of our results in whatever ranking algorithm we choose.

There are, however, some disadvantages compared to the Bloom filter index.

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

Here is how an encrypted search index using a MAS can be constructed in pseudo-code:

To perform a query membership test on the MAS index, the user first hashes each of the searchable terms in the document,like in the *crypt\_member* pseudocode, and sends them to the cloud storage provider. The cloud storage provider, unable to see the terms of the query, then applies the minimum perfect hash function to each cryptographically hashed term to retrieve the index into the approximate set. The server then applies the cryptographic hash function on each of these (already) cryptographically hashed terms. If this hash matches the hash at that index, we assume there is a match (with a false probability rate as previously described).

It should be noted that, while in theory the perfect hash is optimal with respect to bits per member for a given 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 searches in which an entire database of documents may need to be queried.

Third, to exacerbate matters, even if an encrypted search scheme provides robust data confidentiality and query privacy, 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.)

To counter this subtler form of information leakage, we may explore ways to mitigate this form of information leak by allowing a single query term (encrypted/private) to have N independent bit string representations such that the user could issue the same query N times and the server would see a different encrypted query each time (but the user would get the same relevant documents back). How? Quite simple, really: for each term, hash it with different known concatenations, e.g., numbers, or combinations of multiple secret keys.

This approach, however, will take up more space, so this turns into another trade-off. (The exact amount of space can be expressed formulaically.)

Other ways we can explore this trade-off include locality-sensitive hashing; that is, similar terms hash to the same value (fewer terms to insert). This will allow us to do cheap approximate matching, like stemming (in which the morphological forms of a word reduce to the same base form, e.g., {“computing”, “compute”, “computed”} all map to “comput”.

Wildcard searching is another trade-off we can make, e.g., when inserting the word “age”, also insert its 1-edit distance wildcard patterns (“\*ge”, “a\*e”, “ag\*”, “a\*ge”, “ag\*e”, “age”). Now, a client may search for “a\*e” to get a match on “age”, or “ate”. A similar approach can be used for matching against any whole words, e.g., the query “planes \* automobiles” matches both “planes trains automobiles” and “plains or automobiles”.

Finally, we will pursue these trade-offs in the context of exact phrase searching. Here, though, things aren’t as bad as they seem. To enable exact phrase searches of any length, in theory we need only insert the 1-grams and 2-grams of the document. So, to search for “hello doctor archer” we need only search for “hello doctor” AND “doctor archer”. This is called the biword model. (Note that this approach, like the minimum approximate set approach, allows for false positives.)

Moreover, to increase the accuracy (decrease the false positive rate) on m-gram queries, where m >= 2, we observe the following: for an m-gram query to be a true positive, all of the 2-gram subqueries and 1-gram subqueries that the m-gram query is composed of must also be true positives. So, we can check each of these subqueries independently; for the m-gram query to be considered a positive, all of its sub-queries must test positive too. If none of the queries and subqueries are true positives, then the probability of a false positive is simply p^(2m-1).

Whether this is a fruitful avenue to explore, we cannot yet say. Clearly, this will increase the computational burden on the server since more matches must be performed. In addition, since the server cannot decompose an exact phrase search into sub-phrase searches as required, due to query confidentiality, the user submitting the query must do it for the server, which increases the transmission cost.

That’s it for now. We are in the process of working out our experimental design.