Ok, this is much longer than I anticipated. I'll be sending you something tomorrow night about the metrics and parameters, although I have already discussed this somewhat in this document.

(1) In "PERFECT FILTER", there is a statement, "More importantly, it only requires computing two hash functions. Thus, it has a significant computational advantage over Bloom filters", while in "DISADVANTAGES", there is another sentence, "The time to construct a Pf is slower compared to a Bloom filer, although it is still linear in the number of members". These two sentences sound antagonistic to each other. Did I misunderstand anything?

The time to construct a *Pf* filter may be a little slower as opposed to the Bloom filter. However, this is the time to make the filter. Once the Pf is made, to do query membership tests, there are only two hash functions that must be evaluated in comparison to a Bloom filter's k hash functions for membership queries.

So, we trade a bit of extra construction time for a savings in membership query times. I would argue this is a good trade-off to make, since there will be significantly more membership queries than reconstructions of filters.

(2) I try not to complicate your thesis, but do you plan to cover relevance searches for phrases that consist of multiple words? I believe relevance searches for phrases are easy to implement and scoring and ranking documents based on relevance searches for phrases should be straight forward (please correct me if I am wrong for this assumption).

Yes, you are correct. The searchable *terms* of a document can be anything – the index itself doesn't care what you insert into it.

So, to enable searching for exact phrases, my thought is I will include not only the 1-grams of a document, but also its 2-grams. That is, I will use a ***biword*** model for exact phrase searching. For instance, suppose we have a document block that reads:

doc = "This is an exact phrase. Fujinoki says hello to world."

First, in the secure index, I will make the following 1-grams members of the Perfect filter:

{ hash("this"), hash("is"), hash("an"), hash("exact"), hash("phrase"), hash("fujinoki"), hash("says"), hash("hello"), hash("to"), hash("world") }

I will **also** make the following **2-grams** members:

{ hash("this is"), hash("is an"), hash("an exact"), hash("exact phrase"), hash("phrase fujinoki"), hash("fujinoki says"), hash("says hello"), hash("hello to"), hash("to world") }

So, to enable exact phrase searching, consider the following query:

query = {"this is an exact phrase", "hello world", "fujinoki"}.

This is a query consisting of four terms. Two of the terms are the phrases, "this is an exact phrase" and "hello world". We do not want to search for the occurrence of each individual word in those two phrases, we want to search for occurrences of those exact phrases as given.

So, if you want to find the phrase “this is an exact phrase”, you could look for a conjunction of the following 2-grams: {“this is”, “is an”, “an exact”, “exact phrase”}. Since we are only interested in a match on all of those terms within the same block we devise a simple query language in JSON so that we can specify that the search doesn’t simply look for occurrences of each of the 2-grams separately.

So, let’s transform the query {“this is an exact phrase”, “hello world”, “fujinoki”} into:

query = {

chain: { hash(“this is”), hash(“is an”), hash(“an exact”), hash(“exact phrase”) },

hash(“hello world”),

hash(“fujinoki”)

}

A *chain* means that a match on all of the hashed terms in the chain set must be found in the same block for it to count as a match (or adjacent blocks, if we want to add the ability to look for phrases that span multiple blocks). Note that this will, like with the Bloom filter and the Perfect filter, never report a false negative, but it can report a false positive (independently of the filter’s false positive rate) if a block has each of the 2-grams “this is”, “is an”, “an exact”, “exact phrase” but not as a chain.

Conceptually, at this point the search will go through each block looking for occurrences of the 3 terms, “this is an exact phrase”, “hello world”, and “fujinoki”. It will calculate how important each term is using standard methods in information retrieval, e.g., tf-idf. It will then measure the proximity of all those 3 terms, e.g., having all three terms in the same block—“this is an exact phrase”, “hello world”, and “fujinoki”—will have a better proximity score than if the smallest block interval containing those terms is 2 or more blocks.

Additional thoughts on approximate phrase searching

For approximate phrase matching, the easiest approach is to just make the user specify the approximation, e.g., the user could search for “hello * fujinoki”, where * is any word. In this way, “hello doctor fujinoki”, “hello dr fujinoki”, etc. would each match. To support this kind of query, consider the document, doc = “Hello Dr. Fujinoki. How are you?”

When constructing the secure index for this doc, insert the 1-grams and 2-grams like before. But, also, insert the following 2-grams:

{ “hello * fujinoki”, “dr * how”, “fujinoki * are”, “how * you” }

So, the user could just search for “hello * fujinoki”. This pattern is in the secure index, so a match is possible. What about: “hello * fujinoki how * you”?

We have to break it down, similar to the bigram model before. “hello * fujinoki”, “fujinoki how”, “how * you”. This transformation can be done in a single pass.

What about the query “hello dr * * are you”? “hello dr”, “dr * * are”, “are you”. As you can see, to support this, we would have to skip two words, and insert the resulting bigrams from that, e.g., insert { “hello * * how”, “dr * * are”, “fujinoki * * you” }.

The cost of doing this isn’t too bad. If N is the number of unigrams, then to support exact phrases of any size, we only need ~2N members total. To also support approximate phrase searching with the ability to match k word wildcards in a row, the total is ~ 2N + kN = (2 + k)N.

The same thing can be done for typographical errors, e.g., when inserting “hello”, also insert its 1-edit distance error patterns, “*ello”, “h*llo”, “he*lo”, “hel*o”, “hell*”, “h*ello”, “he*llo”, “hell*o”, “hell*o”.

What about combining this with the word wildcards mentioned previously? It can be done.

“Hello Dr. Fujinoki, good day!” ->

{
“hello”, “dr”, “fujinoki”, “good”, “day”,
“hello dr”, “dr fujinoki”, “fujinoki good”, “good day”,
“hello * fujinoki”, dr * good”, “fujinoki * day”,
“hello * * good”, “dr * * day”,
“*ello”, “h*llo”, “he*lo”, “hel*o”, “hell*”, “h*ello”, “he*llo”, “hel*lo”, “hell*o”,
…
“*ello dr”, “h*llo dr”, “he*lo dr”, “hel*o dr”, “hell* dr”, “hello*dr”, “hello *r”, “hello d*”,
“h*ello dr”, “he*llo dr”, “hel*lo dr”, “hell*o dr”, “hello* dr”, “hello *dr”, “hello d*r”,
…
“*ello * fujinoki”, “h*llo * fujinoki”, “he*lo * fujinoki”, …
}

As you can see, however, the number of members starts blowing up. So, we will likely place further limitations on what sort of approximate matching can be done to get this under control while still allowing for a lot of useful wildcard searching.

For more approximate matching, I can also include stemmed terms and removed stop words along with their 2-grams for a biword model, including the wildcard words. All in all, we will try to maintain a total secure index size equal to some factor (between 0 and h) of the encrypted document size. Note that **h** can be another parameter we play with.

Note:

> Since *hidden queries* are used, e.g., the CSP cannot determine what a client is searching for, the CSP’s hands are tied with respect to finding derivations or approximations to terms, like typographical error tolerance (e.g., 1-edit distance character patterns). Instead, the client must embed this information—metadata—in the query. I’ll create a basic query grammar (probably just JSON) to represent a sufficiently rich language to allow for this.
>
> Things would be much easier if the CSP could be trusted with plaintext queries, but plaintext queries leak far too much information to be useful for encrypted search.
>
> (As an aside: However, if plaintext queries were allowed, the CSP itself could apply all sorts of transformations to the query terms, e.g., apply stemming for morphological word variation

tolerance, apply 1-edit distance patterns for typographical error tolerance, and so on…sort of like how I originally planned on doing this before I decided hidden queries were a necessity. Regardless, since hidden queries are needed, the best we can do is, and I could be wrong, encapsulate a subset of that functionality by embedding the necessary variations and metadata in the query itself. I'm sorry if that was confusing.)

For experimental designs, can you come up with your experimental designs? By saying "experimental design", I mean the definitions of:

(1) What are the metrics we will measure?

Note I will work on this a lot today (I worked on it quite a bit yesterday, but only have this document to show for it so far). However, here are some metrics related to the effectiveness of information retrieval (IR):

- Precision: *{ relevant documents } AND { retrieved documents} / { retrieved documents }*
    - Being forced to sift through irrelevant results is a waste of time and energy
- Recall: *{ relevant documents } AND { retrieved documents } / { relevant documents}*
    - Missing relevant documents in returned results may be very costly, especially in the "vertical search" market there may only be a small set of documents that are actually relevant
- Combing these two measures into a Mean average precision makes more sense in our case since our relevancy score will not be just "relevant" or "not relevant", but instead it will provide a "degree of relevancy", and we will return the top N results.
    - Average precision considers the order in which documents were presented (based on their rank score).
    - Mean average precision is just the mean of the average precisions over a set of queries.
- Time lag: *on average, how long does it take for the system to process the test queries. Lag is bad for two reasons: (1) clients want results quickly – anything more than 2 seconds is often considered intolerable, and (2) larger lag ~ higher cost to process query, so the CSP will charge more per query on average.*

I think **Mean average precision** and **time lag** are two important IR measures. Parameters like **block size**, **false positive rate**, relevance functions used, etc. will effect these measures.

The next set of measures are related to a factor unique to encrypted search: how much information is leaked by the system? There are three types of information leakage that I have identified:

(1) Information leaked to clients authorized (know the secrets) to query the collection of secure indexes. They may be authorized to conduct searches, but they may not be authorized to view the contents of any encrypted document. So, it may still be important to avoid exposing too much information to authorized clients.

Larger blocks => less information is leaked. Higher false positive rate => less information is leaked. Returning irrelevant results => less information is leaked. In all cases, less information leakage implies less effective IR measures. In fact, information leakage is minimized when the set of retrieved documents for a query is random. This is great for no information leakage, but

terrible for IR measures. *It may be best to ignore this type of information leakage in our study for now, unless we want to devise some cost function that takes both of these factors into consideration.*

(2) Information leaked by the structure or contents of the secure index

We will do our best to mitigate this type of leakage, e.g., (1) seeding the secure index with random junk to make frequency analysis harder/impossible, (2) the same term in different documents (and different blocks in the same document, if deemed necessary) will have different cryptographic hashes in the Perfect filter to make correlation analysis ineffective when *only considering the secure indexes*.

If think our current system design does about as well as is theoretically possible while still allowing relevancy measures to be very useful. The only thing that can be inferred from a secure index is the approximate number of "real" terms per *Pf* block. But do note this is complicated by the fact that every Pf will have a randomly chosen number of junk terms added to it. They can still fit a statistical model to this, e.g., assume the number of terms is some constant C + uniform(lower, upper), where uniform(lower, upper) determines how many junk members are added to the filter. An effective way to infer C is to take as input all of the *Pf's,* and find values for lower and upper for the uniform distribution that maximizes the probability of seeing that distribution of *Pf* index sizes. At that point, they have an estimate of C, i.e., C = sum{numIndex(Pf_i) for i = 1 to N} / N – (upper + lower) / 2. Of course, the model can be wrong; maybe it is not the uniform distribution, but some other distribution. Or maybe C is itself a random function.

Note that proximity relevancy measures can still work even if number of terms per block is unknown as long as we use approximately the same number of real 1-grams per block. If so, then block index can directly serve as an approximate position and thus we can use them in our distance functions for proximity.

I believe some of the papers discussed this type of information leakage in terms of a theoretical *game*. I'll have to look at that again, although none, as far as I can tell, talked much about the issues I bring up above. For now, I think it may suffice to talk about this kind of information leak in terms of what we've done to make it impossible to determine much, if anything, from the structure and cryptographic content of the secure index itself.

(3) Information leaked by an adversary that considers the client's history (e.g., query history and history of retrieved documents). This isn't discussed much at all in any of the papers. Some talk a bit about oblivious RAM, but then point out it's at present too costly.

So, one kind of leak is simply seeing a user's history of retrieved documents. From this, a histogram over the documents can be constructed. Now, histograms between users can be compared, e.g., users can be grouped together. Other information, like "when the query was submitted" can further refine the groups. This does constitute an information leak.

There are, in fact, some things that can be done about this. One idea is introduce randomness into the documents retrieved for a given query. For instance, when a user constructs a query, add a couple of random words as search terms. This will decrease the relevancy score, but this may or may not be a problem. (E.g., local searches on the reduced collection of documents.)

Now, also, if the same query always has the same hidden representation, this is also problematic. Again, most papers didn't address this. Why is it a problem? An adversary, like a CSP, can do two things: (1) More refined histograms, now factoring in query terms also. (2) Keep track of how many times a given hidden query term has been seen, then map these to retrieved documents. Once each hidden query term has been mapped like this, they not only know the frequency of each hidden query (from a single user or a collection of users), they also know what the hidden query terms map to. This is really just a substitution cipher (although not all of the terms are seen, only those terms a user has decided to search for), which can be broken, or at the very least leaks a lot of information.

One solution to this: for a given query term, hash it with the document's id (and do the same during document construction – which is already done but for different reasons). Now, send that hidden query term to the CSP. The same query term looks different for each document.

The problem with this is twofold. (1) If there are thousands or more of documents, and you (almost assuredly) want to the query some sizable subset of this collection, then a separate (or a combined) query must be constructed for each document. This is simply not feasible if the document collection is even slightly large, e.g., 100 or more. It's slow and bandwidth hungry. (2) Information leakage still occurs. First, the same query term on the same document always looks the same. So, we're back to the original case, with the exception that we have less data to do statistical inference. Second, since the CSP knows that the next M queries over some small interval of time from the same user likely represents the same query (indeed, for efficiency, the user may package all of the separate document queries together), this can be exploited.

However, there is another way, and it borrows from the idea of Oblivious RAM. We can address this to whatever extent is desired by trading space complexity for information leakage resistance. I've discussed the idea in the proposal, but here it is again.

When constructing the secure index for a document, for each searchable term in the document, hash it with a secret and then add that to the *Pf* (note that the Pf will apply its own perfect hash function and then cryptographic hash function to each term, also). This is what we've always done. In addition, to make it so that each hidden query term has N variations, add this term N-1 more times, but each time hashing it with some variation, e.g.:

term -> { hash(term | secret), hash(term | secret | secret), hash(term | secret | secret | secret), ... }. Note that each "secret" can be the same, or they can be different; it doesn't matter. So now, each term has N representations in the secure index, which means that it blows up the index by a factor of N. But the larger the N, the less information is leaked. How? Well, the idea is

for the client to randomly choose some valid concatenation for each query term, e.g., query = "hello world" => hash("hello world | secret | secret | secret | secret"). Now it's true that with effort this leaks information too, but not nearly as much as the previous solution (nor is it nearly as costly). (1) The same hidden query term for the same document will vary in N different ways, as opposed to the previous solution which didn't vary for a given document. (2) The time between the same query term will be spread out much further in time. This makes it very difficult to do any sort of correlation analysis. When we combine this with random query seeding, then it becomes nigh impossible to do any correlation analysis.

In the end, we do want to return relevant documents for a query, so some information is going to be leaked. But, I think with our construction, you're not likely going to discover anything revealing. This will be probably the strongest link in the chain; weaker links include people revealing a secret (or partial secret), talking over insecure channels, etc.

(2) What are the parameters we will input?

- Block size
- False positive rate
- How *relevancy* is measured
    - This is important. For instance, if we measure proximity with a function *f* that takes distance as an input, should *f* be fed the square of the distance or the absolute value of the distance? And, of course, the function template, *f*, can also be a parameter, e.g., is term proximity proportional to the reciprocal of distance or is it related to exp(distance)?
-

(3) What values will we change to study their effects to the metrics we defined for (1)?

It will be the best, if we can name each experiment with well-defined input parameters. For example, if the primary input parameter that will be changed is "block size" in the number of distinct words, we name the experiment as "Block Size Experiment" and enumerate the block sizes that will be used for the experiment.

I usually developed multiple "experiments", one for each input parameter that will be studied. Those multiple experiments were derived as "the base experiment" as the core. The core experiment will be the "base case" for all experiments. For example, if block size is, say 50 words for the base experiment, "block size experiments" changed the block size to 100, 200, 400, 600, 800 words and so on. I applied the same strategy to other experiments.
One of the questions for this strategy is "how to define the base experiment"? I usually tried some arbitrary numbers to see if the set of numbers can be a good starting point for any interesting outcomes. For example, if the block size is set to "10 words" in the base experiment, and by increasing the size by 10 words for each experiment (e.g., 10, 20, 30, 40, 50 60 and so on) did not make any interesting effect to any metrics I should observe, I stared over the same for 50, 100, 150, 200, and so on). I repeat this for other input parameters until I figure out where is the best place to start to show interesting outcomes (by the way, I hope you keep the outcomes from doing this, since some of them may be used as the outputs from your real experiments).

Please feel free to let me know if there is anything I confused you.

I've been working on this (as you can see from the length of my email ☺). I'll also work on this today. Late tomorrow night, I'll send you whatever else I came up with and we can work on integrating our results. If desired, we can meet Friday to go over things? It would be problematic meeting tomorrow or Thursday, but if that is best for you I can certainly make arrangements.

And, as an aside, block size is definitely one of the more interesting parameters to fiddle with. There are big efficiency gains to making block sizes as large as possible, like smaller secure index size and fewer Pf's to query (fewer queries per document since fewer blocks), but if they block size is too large then proximity suffers. It's an interesting trade-off problem. Almost all of this stuff has to do with trade-offs, so it's properly an optimization problem given some objective function (e.g., linear combination of cost functions).

So, I'll be thinking about formalizing all of this in terms of an optimization problem where we want to minimize some linear combination of cost functions when applying a set of queries to a set of documents, in which each query has some known ranking on the documents. I think we can get these data sets from TREC. If this sounds like too much, let me know. I don't mean to overcomplicate things; I know where on a strict deadline!