How are you doing? I appreciate you agreeing to be on my thesis committee.

I am making solid progress on my thesis. I recently had an insight that has led me to want to explore a new variation.

First, a bit of a refresher for you. I know you're very busy and have probably not had much time to consider my ideas. A lot has actually changed.

I'm exploring secure index structures and trying to find ways to improve the relevancy of searches on them without compromising the confidentiality/privacy. This means:

(a) Queries are hidden -- what is transmitted are one-way hashes of a user's search terms, where a term can be anything, but here's how I have been doing it:
    a. If a term is a single word, then add the cryptographic hash of it to the hidden query, e.g., "ercal" -> SHA256("ercal" + "a secret").
    b. If a term is a phrase, then add the cryptographic hashes of its bigrams, e.g., "doctor gunes ercal" -> {SHA256("doctor gunes" + "a secret"), SHA256("gunes ercal" + "a secret")}. In this way, phrases of an arbitrary length may be found, although this approach an by itself result in false positives. Methods have been devised to reduce this rate, e.g., if proximity information is provided we can do better, e.g., the bigrams above exist in separate, non-adjacent blocks, then this is not a phrase match.
    c. To prevent patterns in hidden queries from emerging, e.g., a histogram of hidden queries may reveal a lot of information about users and plausible mappings from hidden terms in the hidden queries map to real terms (a substitution cipher). To counter this:
        i. 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.

        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 N^{K-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

(b) The secure indexes leak very little information about the contents of the document it represents – even information like "how many words does this secure index contain" can be hidden, although to facilitate proximity scoring some information about that can optionally be leaked. The secure index is not as secure as an encrypted document (assuming large block encryption) – the secure index must be searchable, after all, but every searchable term is a product of at least one cryptographic hash, and what's inserted into the index is not the hash of the terms, but a transformation of the hash, e.g., in the perfect hash based secure index, a hash of N bits is transformed into a hash of M bits and those M bits are inserted. Typically, M is 10 bits or so, but it can be anything desired. The larger M is, the less likely a false positive will occur, where false positive rate is equal to $\frac{1}{2^M}$. If M is 10, then false positive rate is ~ 0.001. Actual tests by me have born this out in practice. And, being only 10 bits, every string in the universe maps to 1024 possibilities – so even if you could somehow find a way to efficiently find inputs that map to a given cryptographic hash (I use a variation of SHA256), the index only contains 10 bits for that entry – so there is 0.001 probability that any given input will map to whatever is "truly" mapped there.

What's more, I do not insert the result of this into the secure index. Rather, for each term t, I insert cryptographic_hash(cryptographic_hash(t + "a secret") + "a document reference"). The document reference is known – it's only used as a "salt" to prevent any patterns from emerging for the same terms in different secure indexes, and to make finding collisions more difficult (but it's not that hard in this case since every string must map to 1 out of the 2^M bit patterns). This makes searching a collection of documents less efficient – we cannot use a single, global index, but rather each document has its own separate index – but it is desirable to avoid leaking too much information.

Finally, I "poison" the secure index with random noise (add extra index locations) where there is no way to determine if a particular index is for a "real" term or a random noise term. This does not effect the false positive rate at all; the more you poison it (it's a parameter), the less information leakage, but the larger the secure index file/memory space requirements.

Ok, now that that's out of the way, on to more recent updates.

I have devised a few different variations of my perfect-hash-based secure index.

PsiBlock, PsiFreq, and PsiPost. PsiBlock uses the perfect hash to map to an index. This index is used to both find out if the term is there (1/2^M false probability rate), but also to determine the places it approximately appears in the document (each word is assigned to a block). The level of approximation is a parameter, e.g., you can have blocks of length 1 (which is thus an exact position), up to blocks of length R, where R can be any positive integer. Typically, I make the blocks be 250 words, which is expected to be 1 page (average page has 250 words). This leaks some information in addition to term presence, but only (if desired) approximate positions.

If the document has 1000 words, then this will create four bits per searchable term. I use a bit vector (a single one for all of the terms, but each term will index into different parts of this bit vector – so, it's pretty space-efficient) to represent this information. If the document is really long, say 100000 words, then this would create 400 blocks, thus each term will 400 bits for block information, which means the PsiBlock secure index may be quite large. Note that though there are 100000 words, many of them will not be unique.

I also have a parameter to specify a max number of blocks – usually, it's around ~100—to prevent unreasonably large bit vectors from being required. This approach generally results in secure indexes that are approximately the size of the original unecrypted document. If file compression is used, since the bit vector is often sparse, it tends to be smaller than the original document. If the max number of blocks is set low, like 10, then very small sizes results.

I use this information in the PsiBlock to do relevancy searches: I weight the terms according to standard information retrieval practice, e.g., use bm25 (a tf-idf term weighting variation) – which uses frequency information (over entire collection of secure indexes) to score relevancy. I also added to this score a proximity score that is based on the aggregate sum of the minimum pair-wise distances in a secure index of all the query's search terms (where a single term can be a unigram up to an n-gram, n arbitrarily large). Min pair-wise distance in tests has proven to be better than the alternatives, but it only slightly improved things in the papers I read about it. Still, even a slight improvement is important when it comes to searching (Google pays huge sums of money for every tenth of a percent).

PsiFreq: This secure index (also based on a perfect hash) does not represent locations at all, just term frequencies. This uses the minimum number of bits needed to represent the most frequent term in the document. This results in very small file sizes, but because it can represent the actual term frequencies exactly (if desired – or an approximation of info leakage is a concern), the bm25 score tends to be more relevant than PsiBlock, which only represents frequency of terms at the block level (e.g., a term appears in block 3 and block 6, so its frequency is at least 2, but it may have appeared multiple times in block 3 or 6).

I'm developing an experimental framework to evaluate their relative performances.

PsiPost: Use a perfect hash, as in the prior two. Like in PsiBlock, it represents proximity information. However, it uses not a bit vector, but a vector of locations. For instance, the bits 00101 for term t represents the locations t is at, in this case block 3 and block 5. So, in PsiPost, I just store [3, 5] instead. I started to find a space-efficient encoding for this structure, but decided otherwise.

A similar algorithm to PsiBlock is used to rank (with respect to queries) PsiPost secure indexes. I haven't finished it yet, but it looks like PsiPost will be faster (algorithmically) when calculating minimum pair-

wise distance for reasons understood analytically. In fact, PsiPost and PsiBlock can use the same file format representation, if desired, and it's just a matter of using different data structures (in memory). What's the best file representation? Probably just a compressed file, but ideally I want to be able to load files quickly into the secure index database, so an efficient file format for that is still something I have pursued…although maybe I shouldn't have.

BsiBlock: This is a bloom filter representation. I am just now finishing it. It's a lot like PsiBlock, except where PsiBlock evaluates just one hash function, BsiBlock must represent multiple hash functions per block, where each block is represented by a different bloom filter. I am curious to see how well it does compared to others.

BsiFreq: This could be a counting bloom filter, but I don't think I'll bother with it unless I have time. It would be similar to PsiFreq.

Last, and most recently, **PsiMin**. This one has a lot of promise, and I only began thinking about it while implementing algorithms for my min pair-wise distance algorithms recently. The idea is to do the following:

Starting with the PsiFreq to store term presence and term frequency information. Then, add *another* perfect hash to index into a pairs of terms – not all of the pairs, but pairs of terms for which minimum distance is less than some threshold k. Probably, I will store the pair typographically to minimize number of pairs needed to be stored (and for proximity, order of terms in a document doesn't matter anyway), so that I store ($term_i$, $term_j$), where $term_i < term_j$ typographically. There will be O(k*(number of words)) pairwise terms of this. Each entry in the pairwise entries will require log(k) bits. If terms are present but are farther than k, I will assume a worst-case distance in the algorithms.

So, for large k, this is probably not reasonably, but the good news is, for proximity scoring, nearby words are important, but words that are farther than some threshold aren't. So, whether it is 3 pages away or 50, it may not matter that much, but it does matter if they are 4 words as opposed to 10 words. This is one of the reasons why I decided to pursue this, since block granularity in PsiBlock is not very effective for small blocks, e.g., 20… it may work better for PsiPost, we'll see, but PsiMin seems really promising.

For a query, we'll have to represent it differently. Basically, for term frequency information, I store the hash of the terms in the query, and for pairwise distances I also store the pairwise hashes. In this way, no information is leaked.

Anyway, I thought you'd like an update on my progress. I would be happy to discuss this with you any time.

Also, have you had any luck finding someone in the math department to sit on my thesis committee?

Regards,

Alex