Thank you. I made huge progress.

I have basically verified -- I hope -- all of the secure indexes. They seem to give the expected output on the simple test cases I devised. Now, I am in the process of making an experiment framework.

I will measure more things than originally planned. Also, due to the modular nature of my solution, I was able to incorporate more secure index types. There are 6 in total. My favorite is the Psip – perfect hash that indexes into a scrambled/randomized posting list. First, I'll explain each of the secure indexes, then afterwards I'll go over my intended experiments.

(1) **Psib** - perfect hash block-based secure index; location information is at the granularity of block size inputs. It provides approximate location information of terms, and approximate frequency information.

The degenerate case is when block size = 1 (1 word), in which case, while inefficient, its output is the same as the Inverted index. More or less -- there is still the possibility of false positives, which is a controllable parameter. In fact, this applies to all of the secure indexes; they all have a large set of parameters, which I encapsulate in a secure index builder class, one such class for each secure index type.

(2) **Bsib** - bloom filter block based secure index. It's the same as Psib, by definitional, but outputs will vary. Bsib is smaller, in practice, than Psib -- a part of that is due to how I represent blocks in the Psib, and a part of that is due to the fact that theoretical efficiency of the perfect hash has not yet been achieved in practice.

Note that in both Psib and Bsib, we only know a term (a term is anything that can be searched, in our case we can search for arbitrarily sized n-grams – although the block based secure indexes, for speed reasons, can only truly match a term that is less than or equal to the size of the actual words per block, but since blocks are much larger than terms in practice, this may not be a problem, e.g., blocks of 250 words and terms may on average be exact phrases less than 5 words) falls somewhere within a block. Thus, on each query that asks for the locations of a term, if its found, I return a randomized position that falls within the ranges represented by the blocks that test positive for the term.

In this way, the same query will result in different output each time – initially, I used a worst-case estimate, e.g., if the block is size 250, and the terms are both size 10 (10 words in each phrase), then I make the worst-case estimate that they are 250 – 20 words a part. However, this seemed to be too pessimistic. An average will work better, but in the end I just decided to randomize it. I did this for two reasons: (1) the long-run average of this randomized approach will be approximately the same as the deterministic average, and (2) I wanted to use a consistent, single interface, called "getLocations", such that I can send the output of getLocations to a single function that processes it to do calculations like min pair-wise distance, etc. Less work, less code, fewer bugs. I do have faster implementations for min pairwise distance calculations for Bsib and Psib that don't use the getLocations interface, but I stopped using the faster variations (not sure why – it just seemed appropriate to focus on the general case, so that the architecture itself supports easily adding secure indexes without a user having to implement anything but a few basic interfaces for it and the rest will be taken care of by the secure index db).

Note: The secure index db works for ANY object that implements the SecureIndex interface, e.g., in a secure index database, there could be any number (I have made 6) ***different*** *secure indexes--the secure index db doesn't even need to know what kind of secure index any of them are–and new kinds of secure indexes can be added at will as long as you implement the secure index interface. I have recognized several other interesting possibilities for secure indexes, most of them would be simple combinations of EXISTING approaches, e.g., if a secure index type provides no (or weak) proximity information, but strong frequency information, combine it with an index that has strong proximity but weak (or no) frequency information.*

(3) **Psif** - perfect hash frequency secure index. It ignores location information -- it accurately and efficiently represents frequency of words (or, rather, searchable terms). Thus, it does not implement the "locations" interface, thus it cannot participate in any sort of proximity measures.

Btw: All of the proximity-enabled secure indexes implement a "getLocations" interface, which gets the locations of the TERMS of a query (where a term can be a single unigram, all the way up to an n-gram of arbitrary size). This turned out to be more complicated than expected -- the easy case is when a term is a unigram or a bigram, in which case the secure indexes store those directly and calculations are fast and straightforward on those. Otherwise, a term is an n-gram, n>2, and we need to use fancier algorithms that take advantage of whatever information is given in the secure index (e.g., bigrams can be used to test for the presence of phrases in a block, and other approaches are used elsewhere). Thus, we can do a search for, for example: query = keyword1 keyword2 "this is an exact phrase 1" "this is an exact phrase 2" keyword 3.

It will find the (approximate) locations of those 5 terms, and then apply the distance measure on it to get a proximity score for it. As I said previously, I use a min-pairwise distance measure, which seemed to do the best in practice.

Recall: we can combine the distance measure with the term weight measure to get a total score that is sensitive to both query term proximity and query term weights.

(4) **Psip** - This is an interesting approach. A perfect hash containing a scrambled postings list (positions) for each bigram/unigram. This represents location information in an unusual way – by taking a word at position k in the real document, and making it at position k + R, where R is a discrete random variable. This random variable R can be any discrete distribution – it doesn't matter. For instance, you could make R have the following pdf f:

r	-5	0	5
f(r)	0.25	0.5	0.25

In which case, each word will be shifted 5 units left with probability 0.25, remains in the same position with probability 0.5, and shifted right 5 units with probability 0.25. I have chosen, for the test, to simply use a uniform distribution that ranges from –r to r, where r is usually something like 10.

So, this tells users approximately where a word is, but order information is unknown. It has some similarities to block based secure indexes, but it is more general and moreover, the block-based secure indexes leak the following information: they know that words in block j come BEFORE words in block j+1.

So, the psip may leak less less information. Moreover, the approximate location is CENTERED around the actual location, as opposed to block-based secure indexes, in which a term falls within certain blocks and it doesn't matter where in that block. This information is lost – for better or worse, depending on your need for security.

For instance, if two terms are separated by 2r+1 units (word positions between them, i.e., the difference between the last word position of the first term and the first word position of the second term), and we are using a radius of r in our psip (a word can be r units left or right of its real position), then at one extreme they will show in the psip as being 4r+1 units apart, and at the other extreme they will show as being 1 unit apart. If the two terms are actually separated by only 1 unit, then at one extreme they will show as being 0 words apart in the psip, and at the other extreme as being 2r+1 units apart. In the block-based secure index, with block sizes of size 2r (note: 2r block sizes must be used to be comparable to psip, since psip is +- r, which means a term falls within a 2r range centered around its real position), the same situation: at one extreme they may be 4r units apart, and at the other extreme 1 unit apart. So, for the same level of granularity r, psip does twice as well as far as proximity information, AND ordering information is lost. Less leakage of information, more accurate. It is, however, slower – but not necessarily larger (I haven't chosen the best representation of the psip, but it's still reasonably small).

To find terms in the psip, e.g., a phrase of k words, I have to do some heftier calculations, e.g., finding each multi-word term itself involves finding intervals of all the bigrams in it for which their maximum difference is less than or equal to the maximum distance possible for the real term to be spread out given a radius scrambling of r. Tests will reveal how slow or fast it is. And, of course, this just gets us the locations of terms – after that, we must use the locations to calculate, say, the min pair distances between DIFFERENT terms in the query. A lot of calculations may be needed for queries with large exact phrases in it. The simple case of queries with no exact phrases, e.g., just a set of keywords, is MUCH MUCH less complicated, both algorithmically and time-wise, although I haven't always attempted to optimize this simple case so while it will be faster, it won't be as fast as it could be.

Note that if we make offset radius 0, this will for the most part have the same output as an inverted index. However, it defeats the purpose of a secure index to have such refined knowledge, although there still isn't very much known about the secure index, e.g., we know that patterns of m bits (m is 7 to 12 typically) have certain positions. It is trivial to make the listings be as approximate as desired, e.g., throw out some occurrences of a term if it appears multiple times within some range of word positions—I have implemented and commented out the most simple variation of this.

(5) **Psim** – perfect hash minimum pairwise distance. So, this one is interesting also. First, it DOES NOT reveal location information at all. All if reveals is the minimum distance between two pairs of unigrams. So, notice I didn't say "pairs of terms". This is a limitation of Psim. It doesn't tell you about distances between general terms, like the exact term "this is phrase 1" and the exact term "this is phrase 2". So, immediately we see that Psim cannot service exact phrase searches, although since it uses Psif for frequency information, it can service frequency requests for arbitrary terms.

Since the best distance measure seemed to be the minimum pairwise distance, I went ahead and directly encoded this information in the Psim. So, it leaks for little information, but provides exact information on pairwise distances up to distances of k. If two terms have a minimum distance that is larger than k, then we just assume the worst case, e.g., they are maximally far apart. Parameter k is

designed to restrict the number of possible pairings – if we went with no restriction, then a document of size n would have O(n^2) pairs, e.g., a document with a 1000 UNIQUE words would have a million min pairwise distance pairings, which is excessive. By using k, it is O(kn) instead, e.g., if k = 10, then it is O(10n). Still quite large compared to the other secure indexes, e.g., the other secure indexes are just O(2n) (it is 2n, instead of n, because it stores both unigrams and bigrams—bigrams are for exact phrase matching).

With sufficiently small k, Psim may be workable. We'll see hopefully in our tests. This is the least tested version, but I don't think I have time to try to validate it anymore, I need to move on to finishing the testing framework.

Note: It occurred to me later in my research that for some of the secure indexes, I do NOT need to store bigrams, e.g., psip just needs unigrams. I would address this (make it more efficient), but there's not enough time. Also, the other secure indexes need the bigrams, and when I send a hidden query, the bigrams are sent for a phrase already, so to make all of the hidden queries the same no matter which secure index types are stored in the secure index db, may as well make them all store bigrams. Note that bigrams even in the psip make still be desireable for speed reasons, e.g., quickly test presence of exact phrases without having to do all the later calculations that psip does.

(6) **Bsif** – bloom filter block based index + perfect hash frequency index, i.e., Bsib + Psif. I just composed them together, so it wasn't much work. All of the secure indexes can be composed in a similar way, e.g., Psib + Psif, to compensate for their strengths and weaknesses. I choose this combination because Bsib is more space efficient than Psif (but not as fast), but since blocks lose a lot of frequency information (cannot represent frequency very efficiently), I combined it with an index that can.

Experiments:

The outputs from the various experiments will be compared with the canonical output of a plaintext Inverted Index which provides completely accurate (no approximations/randomizations/information loss) results. This canonical output will be the presumed "ground truth", so the question is, how accurately does a secure index match this canonical output? (Implementing an inverted index to support the kinds of queries and relevancy measures I wanted ended up being a lot more challenging than I expected. In hindsight, I should have went with a simple sequential scan-type interface, but that could be very slow for some of the more complicated queries I support.)

Also, secure index construction for each type has many parameters. Here is a total list:

(1) Stop word list (this is a parameter that can be used per secure index). This is complete and working for all secure indexes (a simple preprocessing step for a secure index construction).
(2) Stemming – true or false. Either always false or always true. Stemming is complete and working for all secure indexes (a simple preprocessing step for a secure index construction).
(3) Other forms of locality sensitive hashing, e.g., inserting Soundex representations of words. This is not implemented, but it is a simple matter to plug it into the preprocessing step.
(4) Load factor – of the "index" positions, how many are for real members, and how many are for fake members. Fake members are used to obfuscate the index, and doing so without effecting

(theoretically and in practice) the expected results, although it may change things slightly by chance. This is basically a "poisoning" step, and it is working completely for the perfect hash-based indexes, and I have worked out how to do it with the bloom filter based index – it is trivial to implement (code-wise), but I do not want to add to the number of experiments we must analyze. Theoretically, as mentioned, it should have no effect on results that changes its expected (statistical) behavior across many different queries.

(5) For the perfect hash indexes, several different algorithms exist by the library I am using, and my indexes are completely agnostic. We will be using only one of the algorithms, the minimum perfect hash. This one also ignores the load factor parameter, which we will not be using anyway, but if we did want to vary load factor we would have to use a different algorithm, e.g., CHD_PH instead of CHD (two different names the author of the perfect hash library gave them, CHD_PH is a non-minimum perfect hash, CHD is a minimum perfect hash.)

(6) Maximum frequency. This is a parameter for PsiFreq to make it so that if the frequency of a term is above a threshold, it will make it maxFreq instead. It is doubtful this would have much effect – any very common freq will, after all, not impact term weighting much since these are essentially nonmeaning-ful stop words, e.g., linguistic glue. Thus, if you make the max freq be, say, 512, then you can fit every freq, from 1 to 512, in 9 bits of code. True, other representations, like Huffman codes, would give much better compression, but this is only useful for disk storage— once it is in memory, I elected to use a format that allows me to randomly index into a large bit array, in which case every frequency represented in the bit array must use up the same number of bits. Compression algorithms, like an arithmetic encoder, can be used to minimize space on the disk. In our research, the file format is similar to object memory footprint, so we can estimate memory consumption by file size. (I actually use a Variable Integer file encoding format for many items that do not need random access, which reduces the number of bits needed to represent ; however, this same encoding can in principle be used in the data structure – I didn't do that because it was an unnecessary detail.)

(7) False positive rate will be varied. However, the false positive rates on compositions of multiple secure indexes can each vary their false positive rate, sometimes without affecting the theoretical performance much at all…e.g., PsiMin could allow a high false positive rate on min pairs, but a low false positive rate on whether each word in that min pair is present in the document. I will just use the same false positive rate for all of this, even though that is not optimal.

Additionally, for the composed secure indexes, for something to be a false positive, it must be a false positive in both secure indexes. This means each secure index that composes it can use fewer bits per hash entry for the same level of false positive rate. Their total bits (hash bits for secure index 1 + hash bits for secure index 2) must be the same, though, so no loss space wise. However, to put this to use, both secure indexes must be queried, making operations on it more expensive. There are a lot of trade-offs here. The math is worked out (and in some cases, I implemented it and then commented it out), but I won't be exploring it in this research. Future work on "composable secure indexes"?

Boolean Experiment (Like Experiment #1)

Boolean search – no relevancy scoring. (Could also use ANY or ALL, but let's just use ALL – the secure index interface supports both though.)

Types of Boolean search experiments:

(1) One keyword search, e.g., query = 'hello'
(2) N keyword searches, e.g., query = 'hello ercal fujinoki alex …'
(3) A single phrase search, e.g., query = '"this is an exact phrase"'
(4) N phrase searches, e.g., query = '"this is exact phrase one" "this is exact phrase two" …'

Inputs:

(a) How many secrets (each secure index allows searching on its unigrams and bigrams with up to K secrets to prevent info leakage.) Possible values:
    a. 1 secret
    b. 5 secrets. For all but PsiMin, k secrets instead of 1 will not inflate the secure index much, since it is just a constant factor increase in number of members in those cases.
        i. The PsiMin will not scale well to multiple secrets since we should also do a pairwise matching on all secrets of every min pair term, e.g., each pair of terms present in the secure index, if there are k secrets, has k(k+1)/2 secret pairings. There is a way around this, e.g., only use the k secrets on unigrams and bigrams, not unigram min-dist pairs, but I didn't consider it important enough to investigate.
(b) Block size, if a block-based index. (Words/block)
    a. Parameter values to explore: 250, 1000
        i. 1 to infinite block sizes are supported, but we'll only use 250 and 1000.
        ii. False negatives on term phrases in queries are possible in current implementation of block indexes since I do not check if a bigrams in a term exist in a chain of blocks – this can be done, but for speed reasons and simplicity I do not do this. If block sizes are large, however, the probability of this occurring is small, i.e., ~1/(block_size – number_of_words_in_term)
    b. There is another parameter, maximum number of blocks, that can be used to prevent indexes with thousands of blocks from being made – however, in our test framework, to precisely control block size, we will make maximum number of blocks be infinity.
(c) Expected false positive rate on unigrams/bigrams.
    a. Parameter values to explore: 0.01, 0.001, but any desirable rate is possible.
(d) If using a PsiPost, offset radius.
    a. Parameter values to explore: 10, 50
(e) Randomly generated query set that provides each of the kinds of queries above.
    a. Query sets with one keyword
    b. Query sets with N keywords
    c. Query sets with **one** phrase consisting of:
        i. One or two terms (directly in the secure indexes: unigrams and bigrams)
        ii. Three to ten term phrases, uniformly distributed.
(f) Various corpora to submit queries against.

Outputs:

(a) Size of secure indexes (file size – which is also proportional to memory size also since the files are in general a serialization of the contents, e.g., a bit array has same size in memory as in the file).
(b) Average query time (what was the lag time between issuing a query and getting results)
(c) Recall and precision
    a. Recall: {relevant documents} AND {returned documents} / {relevant documents}
    b. Precision: {relevant documents} AND {returned documents} / {returned documents}

    Note: it is easy to get total recall by returning EVERY document reference, but this hurts precision. Trade-off, in general. Elsewhere, we will use Mean average precision instead, which only works for results which are ordered by relevancy.

Frequency Experiment (New Experiment)

Previous boolean experiment just used the "containsAll" interface on queries. This just uses the "getFrequency" interface on a query **TERM**. A query has no frequency, but the terms in a query do.

That is, how frequently (approximately or exactly) does a term occur in the secure indexes?

Types of frequency experiments:

(1) Keyword (unigram) and bigram phrases – both are directly stored in secure indexes
(2) Complex phrase terms – each secure index will use an algorithm to determine (with possibility of error that is independent of the possibility for error on the unigrams/bigrams) whether the term exists, so this will be an important experiment. Most use a biword model as a basis, then apply additional heuristics to see if these biwords exist within some expected range or other heuristics. PsiFreq is the only one that just uses a biword model, i.e., do all of the bigrams of a phrase exist in it? *Note that the bigrams of a phrase of k words will be represented in the hidden query as an array of k-1 hidden terms.*

Inputs:

(a) Randomly generated query set that provides each of the kinds of queries above.
    a. Query sets with one keyword or a single bigram.
    b. Query sets with a single three-to-K-word phrase
        i. K = 3 to 5, uniformly distributed.
        ii. K = 6 to 10, uniformly distributed.
(b) Various corpora to submit queries against.
(c) Block size, if a block-based index. (Words/block)
    a. Parameter values to explore: 250, 1000
        i. 1 to infinite block sizes are supported, but we'll only use 250 and 1000.
        ii. False negatives on term phrases in queries are possible in current implementation of block indexes since I do not check if a bigrams in a term exist in a chain of blocks – this can be done, but for speed reasons and simplicity I do not do this. If block sizes are large, however, the probability of this occurring is small, i.e., ~1/(block_size – number_of_words_in_term)

b. There is another parameter, maximum number of blocks, that can be used to prevent indexes with thousands of blocks from being made – however, in our test framework, to precisely control block size, we will make maximum number of blocks be infinity.

(d) Expected false positive rate on unigrams/bigrams.
    i. Parameter values to explore: 0.01, 0.001, but any desirable rate is possible.

(e) If using a PsiPost, offset radius.
    i. Parameter values to explore: 10, 50

Random

NOTE: Do not need to vary secrets – done that for the first experiment already, and I don't think it will reveal much doing it again.

Outputs:

(a) Average query time

(b) Average sum of errors for each term across the entire corpus

$$\frac{\sum_{doc \in corpus} \sum_{term \in term\_set} |real\_freq(term, doc) - approximate\_freq(term, doc)|}{|corpus||term_{set}|}$$

Note: Do not need to measure size of secure indexes – done that for the first experiment, and it won't change for this experiment. This is just another way to query the secure indexes.

Term-Weighting Experiment (Like Experiment #2)

This is a standard relevancy measure in information retrieval (IR). We will not be using it with standard approaches, like the vector-space model, however, since our secure indexes do not reveal what is inside the documents – it only reveals this information WHEN an authorized user sends the a hidden query to the server so that the server can OBLIVIOUSLY search the secure indexes. The server doesn't know what the search represented, and it can't read the contents of the secure indexes, all it can do is say "this hidden term was approximately in these locations in these documents with approximately this frequency".

So, I use a variation of term weighting where I only consider the terms that are in the query, not all the terms in all of the documents. An argument can be made that I should use a canonical inverted index that implements something like a vector space model for my canonical output, but (a) I had this realization later in my programming efforts, and (b) some IR systems do use the approach I'm using. Regardless, if a secure index type A does better on secure index type B on the currently specified canonical output, then it will – I strongly believe – tend to also do better on a variation where its output is compared to canonical output using a vector space model. I leave this to future work.

I have implemented term weighting measure BM25, which considers how many documents in the corpus contains a given term and how many times a given term occurs in a given document. The idea is twofold: if a term is rare, i.e., only occurs in a few documents in the collection, then this is a very important, discriminating term. Value it highly. Likewise, if a term occurs in a particular document a lot, then that is also noteworthy. BM25 is just a heuristic than considers both of these factors.

Types of BM25 experiments:

(1) One keyword search, e.g., query = 'hello'
(2) N keyword searches, e.g., query = 'hello ercal fujinoki alex …'
(3) A single phrase search, e.g., query = '"this is an exact phrase"'
(4) N phrase searches, e.g., query = '"this is exact phrase one" "this is exact phrase two" …'

Inputs:

(a) Randomly generated query set that provides each of the kinds of queries above.
    c. Query sets with one keyword or a single bigram.
    d. Query sets with a single three-to-K-word phrase
        i. K = 3 to 5, uniformly distributed.
        ii. K = 6 to 10, uniformly distributed.
(b) Various corpora to submit queries against.
(c) Block size, if a block-based index. (Words/block)
    a. Parameter values to explore: 250, 1000
(d) Expected false positive rate on unigrams/bigrams.
    a. Parameter values to explore: 0.01, 0.001, but any desirable rate is possible.
(e) If using a PsiPost, offset radius.
    a. Parameter values to explore: 10, 50

Outputs:

(a) Average query time
(b) Mean average precision.

Note: I do not think there is much value in exploring the file size output again, although the information will be there to collect I imagine.

Proximity Scoring (Like Experiment #3)

Having read a lot of the literature, I decided to go with a relevancy score that accepts the minimum pair-wise distance summation of all the terms present in the document. It performed well in previous studies compared to other measures, like "minimum coverage" (of all terms present in the document, what is the minimum word position range required to cover them all). I have implemented a minimum coverage algorithm (for determining a term in PsiPost, actually – but it can be readily used for this too), but I will only investigate min-pair-wise in this research, since it seems to perform better anyway.

Having chosen the distance function, we must now apply a function to this distance such that large values score less than small values, but that large values have an asymptotic limit, e.g, a pair of words that have a min distance of a million is no better than a pair of words that have minimum distance of a billion. Proximity, in both cases, is "equally bad." Our algorithm of choice, then, is:

$$\log(alpha + e^{-beta \times d/|query\ terms\ matching|})$$

Where d is sum of distances, and |query terms matching| is the number of terms present that matched.

Same kind of experiments as in **Term-Weighting Experiment**.

Proximity + BM25

Use a linear combination of term weighting and proximity scoring. Term weighting has been rigorously proven over the years, proximity weighting less so. The idea is we will weigh BM25 more than proximity, that is, we will try to just slightly tweak/adjust the BM25 score with a bit of proximity relevancy, but not so much that BM25 is overridden.

Some kinds of experiments as in **Term-Weighting Experiment**. There is, however, one addition:

Query obfuscation. It's a lot like using multiple secrets per searchable term to make histogram analysis of user query patterns from emerging. However, unlike secrets, which are expected not to have an effect on the "long run average" over many queries, obfuscations may or may not – I don't know yet.

An obfuscation is just adding "noise terms" to the hidden query. Nosie terms can be uniformly sampled, or they can be sampled from a non-uniform distribution to give the appearance that certain noise terms are very important…but in fact aren't. They're just there to obfuscate things.

New inputs: 1 obfuscation, or 2 to 5. (Any arbitrary amount can be used, but this seems enough.)

Final thoughts:

Note that all of these extra secure indexes and generalization of the architecture (where you can just plug in new secure index types and only need to implement several interfaces for it to work WITH the other secure index types in the same secure index database) slowed me down considerably, e.g., I had to run down a lot of bugs during the validation stage for the new secure index types—like making simple test cases that I could analyze myself and compare expected output to actual output. This took a long time. I do think my work is interesting and thorough, but this required a lot more work on my part. As long as I am prepared to put in this work (I have slept only a few hours a day for the last couple of week), so be it. I hope you are not too upset with the amount of work I chose to take on – I just wanted my research to make a positive contribution, and to do something you and I both will be proud of. I feel this may be my last chance to submit a computer science paper, and I want to give it my best. I also, of course, desperately want to get my master's degree behind me! ☺

Honestly, I have never worked harder in my life. This semester has been crushing, but it's almost over!

Some additional future work items that you may be interested in, or that you may want me to expand upon in writing or oral communication:

(a) Using a directed acyclic graph of semi-trusted/untrusted intermediate nodes that "decorate" hidden queries with more partial secrets (all of the secrets must be known to query indexes); without them, a semi-trusted user cannot query the secure indexes, even if he or she has a local copy.
(b) One of the few things that does leak when interacting with a secure index database: which documents your queries are relevant to (e.g., histogram analysis). This too can be obfuscated.
    a. For each query a user is interested in, submit up to K queries in a way that will more uniformly sample from the secure index db. This requires some front-end work on the

client to keep track of which queries resulted in which documents, and choosing queries that will give some reasonable distribution. (Or just submit some random queries, e.g., random keywords sampled from a Zipf distribution.)

K result sets will be returned. However, the user knows which of these result sets are for the actual query of interest. So, what's returned to the user looks like a uniform sampling of the references over all of the queries, but what is actually of interest to the user is the result set of one particular query mixed in with the other queries.

b. 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.

(c) I had made a start on devising a sophisticated tree representation for hidden queries to facilitate *hidden query expansion* (or refinement) done on the client's computer before submitting to the server, but I do not have time to implement these ideas.

However, the more stuff you put into the hidden query tree the more information is potentially leaked. This tree is used to represent propositional logic, e.g., "{this term AND this term OR {this term OR this term OR {this term AND this term} OR …. }}". For instance, the clause "{word1 OR word2 OR … OR wordk} may represent synonyms of some word of interest, or a list of misspellings, and so forth. You can, of course, submit the synonyms directly in the secure index at only a slight cost if it is done selectively, e.g.:

a. Preprocess a document. For each word in the document, do word-sense disambiguation, part of speech tagging, etc.
b. Once this is done, for each word, also insert some of the common synonyms (Wordnet) for it.
c. For each word, also insert some of the more specific concepts referred to by the word, e.g., "mammal" -> "cat, dog, …". Clearly, if too much of this is done, the secure index will inflate.

So, instead of doing this in the secure index, it can also be done on the client's end—this is called "hidden" query expansion. Query expansion is itself a common topic, but doing this in the content of keeping the contents of the hidden query confidential is a new twist on this. Maybe the best approach is to do a bit of both: hidden query expansion and some extra data in the secure index.