## \*\*\*Oblivious RAM Revisited

Benny Pinkas, Tzacky Reinman

Summary: hide access patterns using randomization.

If an encrypted search scheme provides robust data confidentiality and query confidentiality, implicit information useful for statistical inference may still be leaked. For example, if an encrypted query consisting of some random sequence of characters is frequently followed by another action, like checking stock prices, this constitutes a strong correlation which may be used to infer properties of the encrypted document and query. To combat this, the authors propose the use of oblivious RAM constructs to conceal the data access patterns (on the server) by the user. It can be thought of in the following way: to prevent meaningful statistical inferences from being made about a user’s activities, whenever an action—a read or write—is performed, randomly include other randomly chosen reads and writes to obscure what the user was actually interested in.Unfortunately, the overhead cost of this may not yet be practical in most commercial systems, but it is one of the only ways to prevent this subtler form of information leak.

## \*\*\*Veriﬁable Symmetric Searchable Encryption for Multiple Groups of Users

Most encrypted search implementations assume only one person would be able to perform the searches, or, if multiple people, then they all must share the same secret, and that secret would allow them to query the secure index. But what if we want to be able to be able to revoke the ability for a user to query the secure index? Take this example:

* User 1 makes secure index for document using secret
* User 2 is trusted to query secure index by sharing with him the secret
* User 1 no longer wants user 2 to be able to query the secure index

How can this be accomplished? A solution is to simply use two secrets, and only give User 2 one of the secrets. Example:

* User 1 makes secure index for document using secret1 and secret2
* User 2 is trusted to query secure index, partly, by sharing with him secret1
* User 1 gives a server secret 2
* User 2 may submit encrypted queries to server on secure index
* Server cannot determine contents of encrypted query, nor contents of secure index, except which documents are ranked as relevant to the encrypted query
* User 2 still needs the server because both secret1 and secret2 must be used to query secure index
* User 1 no longer wants user 2 to be able to query the secure index. He informs server not to honor his query requests
* Even if User 2 has a local copy of secure index, he cannot query it since he does not know secret2 and he can’t ask the server to query it on his behalf since the server has been instructed to de-authorize him.

In pseudo code, a term is hashed into the secure index like so (or something more elaborate):

hash2(hash1(termplain text | secret1), secret2)

Thus, to check the secure index for termplain text, both secret1 and secret2 must be known.

This is the essential idea behind multi-user systems. Note that these models generally assume the server (who is trusted with one of the required secrets) and the partially trusted users do not collude. If server agrees to continue servicing requests of deauthorized users or if server gives the users secret2, then they will be able to continue querying the secure index.

## \*\*\*Conﬁdentiality-Preserving Rank-Ordered Search

A. Swaminathan, Y. Mao, G.-M. Su, H. Gou, A. Varna, S. He, M. Wu, and D. Oard

Relevance: Ranked search results, i.e., context-aware searching.

Most encrypted searching research focuses on Boolean search, where a document is relevant to a query if and only if all of the terms in the query match. As indicated elsewhere, this has a number of problems with respect to precision and recall. The results in this paper represent an important advance over prior encrypted searching schemes in that it ranks documents (out of a set of documents) according to estimated relevance to a query. In modern IR systems, scoring the relevancy of a document to a query is, arguably, its most important task. If standard relevancy scoring techniques in IR can be brought to bear on encrypted searching, the utility of encrypted search will be significantly improved.

Two notable issues with their results: (1) In the case where the cloud storage provider is untrusted, relevancy scoring is conducted by the client’s computer. This is difficult to avoid if the desired outcome is minimal information leakage (we would like to even hide the frequency of encrypted queries, and especially terms of the query, since if any information is leaked with regard to frequency, then statistical knowledge of likely distributions of words—e.g., Zipf’s law—may be used to infer likely mappings between encrypted query terms and words. (2) Their work does not explore any proximity measures, which are of utmost importance with respect to returning relevant results.

## \*\*\*Efficient Text Proximity Search

*Relevance: Proximity-sensitive scoring demonstrably (on TREC test sets) improves the relevancy of the top k results over typical state-of-the-art approaches which do not include proximity scores for query terms, e.g., Okapi BM25 (tf-idf variant). This work shows that it is possible to efficiently use an index (they use an inverted index, but a bloom filter could be used for providing quick and dirty approximations) to efficiently compute rank scores when proximity scores are included.*

In “Efficient Text Proximity Search”, Ralf, Andreas, Seungwon, Martin, and Gerhard consider the importance of proximity of terms in keyword searches. For instance, the query “surface area of rectangle” consists of three meaningful terms: “surface”, “area”, “rectangle”. Relevancy scores which do not take into account proximity will favorably rank any document which contains these terms. Clearly, a document which reads “… the **surface** was filthy, so he resolved to sweep the entire **area.** As he was looking at the **rectangle** his daughter drew …” should not be rated as highly as the document, “... to calculate **surface area** of a **rectangle** …” They consider previously proposed proximity scores and observe a recurring problem: they’re slow since (generally) they don’t use pre-computed proximity indexes (offline). They demonstrate a fast solution that uses pre-computed information about the distance between pairs of terms in a document. Their approach trades speed for space: their proximity information grows the index significantly. If space is a concern, and for large document stores it probably is, this may not be a reasonable option.

Extra details: They analyze Buttcher’s Scoring Model, which is a linear combination of a content-based score (i.e., weighted keywords in a bag of words model, like BM25, which is a vector space model informed by a tf-idf variant weighting scheme) and a proximity-based score. The proximity score, in this model, is derive from Buttcher’s Scoring Model. This model, intuitively, gives higher scores to documents which have more, less distant occurrences of adjacent terms. This seems like a natural enough proximity score, although there are alternative models, e.g., finding a minimum interval (minimum position and maximum position in the document) over the subset of query terms present in the document.

They point out that relevancy scores in which proximity measures are provided perform significantly better on TREC test data sets than relevancy score that do not include proximity information. Note that by proximity score, I do not count models which implicitly include a naïve proximity model as a consequence of partitioning a document into blocks and doing a search over each block independently; most constructions do this to prevent a document from scoring high in relevancy by having all the query’s terms but with large distances between them.

## \*\*\*Concept Search: Semantics Enabled Information Retrieval

Relevance: Relevance, context-aware search.

There are two major problems with syntactic (no semantic information used): (1) Different words (or phrases) may be used to express similar meanings depending on the context. This is called synonymy. (2) The same word (or phrase) may be used to express different meanings depending on the context. This is called polysemy. Both of these problems harm the relevancy of results.

Semantic search asks, “What is the meaning of the text, and does the query match that meaning?” This may involve a lot of techniques found syntactic searching, but in addition, it uses natural language processing to perform word-sense disambiguation, part of speech tagging, or named entity recognition. When combining this with ontological knowledge and logical or probabilistic inference, the IR system may begin to process queries in a way that resembles people's ability to understand text. For instance, if a user says, "find me documents that have carnivores hunting prey", he is interested in concept that is a specific example of that, e.g., "a dog chasing cats" or "lions hunting antelopes".

Using part of speech tagging, it can be determined that "carnivore" is the subject, "hunting" is the verb, and "prey" is the object. Using word-sense disambiguation, the word senses can be determined fairly accurately, e.g., "carnivore" maps to "carnivore-1" (word sense 1). Using an ontology (like Wordnet), it can be determined that "carnivore-1" is a concept which includes (more specific concepts) like "dog-1", "lion-2", etc. Then, we may assume we can expand "carnivore-1", to {"carnivore-1", "dog-1", "lion-2", ...}, "hunting-3" to {"hunting-3", "chasing-1", "preying-4", ...}, and "prey-4" to {"prey-4", "feline-2", "antelope-1", "cat-1", ...}. This is not an easy problem, but experiments have demonstrated considerable improvements in recall and precision of returned results with even modest implementations of this sort of construction.

## \*\*\*Secure Indexes

*Relevance: Offline Bloom filter. Encryption.*

In “Secure Indexes” by Eu-Jin Goh, a Bloom filter is used to create secure indexes of documents. In general, an index is a data structure which stores documents in a way that facilitates rapid, efficient retrieval and ranking operations. A secure index is motivated by the same concerns over speed and efficiency, but must also service confidentiality needs.

Goh points out that a traditional hash table is unsuitable as a secure index since it leaks too much information. A secure index should reveal no information about the contents of the document unless a secret (called a trapdoor) is known which allows the secret-holder to query the index. The only information revealed by this query operation is whether a given document has the specified, possibly encrypted, keyword. If the query is encrypted, then even if an attacker—like an untrusted cloud storage provider— captures an encrypted query (the trapdoor), it cannot determine what the query represents, affording both data confidentiality and query confidentiality.

To guard against information leaks possible through correlation analysis, the same terms in separate documents ought to map to different index positions in the Bloom filter. The author recommends appending a document id to the plaintext terms of the document during the construction of the secure index, and likewise during the construction of encrypted queries, to accomplish this end.

*Notes: The author points out that the index does not conceal information that the cipher text itself publically reveals. For instance, the cipher of the encrypted document has a publically exposed file size which may, in some cases, be used to infer statistical properties about the document. However, it should be pointed out that secure indexes may expose other statistical properties. For example, a bloom filter of size N, assuming an approximate range for the probability of false positives (it should not be too high for it to remain useful as a way to find documents relevant to a given query, nor too low for it to be an appropriate data structure to use), may be used to infer the number of unique tokens (like words) in the document. Admittedly, since it may not be generally known what a token represents (e.g., does it include multiple words or variations of keywords?), this sort of cryptanalysis may be of limited value. Still, this does constitute a very weak form of information leakage separate from the leakage afforded by the cipher text itself.*

## \*\*\*Spectral Bloom Filters

*Relevance: Maintain multiplicities (approximate) for keyword weighting relevancy scores.*

Saar Cohen, Yossi Matias

A classical Bloom filter represents a set, S, approximately as S’, where |S’| = |S| + fp\*|U-S|, where fp is the false positive rate and U is the universe of possible members of S. It’s very space efficient; it’s a factor 1.44 of being optimal for a given false probability rate. So, this is a good choice if you can tolerate some false positives. However, what if instead you are interested in a multi-set, e.g., to count multiplicities of words (word frequencies)?

Spectral Bloom Filters efficiently represent multisets with only a small increase in space complexity over classical Bloom filters. It works quite differently than the Space-Code Bloom Filter construction (which also represents a multi-set) in that it associates each position, previously a bit in the classical Bloom filter, with an array of bits, e.g., 3 bits per position allows 8 discrete values. Thus, when inserting, simply increment the value at each of those respective positions. The primary differences come from how the multiplicity of an input is estimated—take the minimum of all the counters that the input maps to—and in how updates are handled, e.g., if x’s multiplicity is incremented, then a further optimization is to only increment the values at the k hashed positions if all the positions are equal. Of course, with this optimization, it is no longer a Bloom filter which can accomplish deletions. However, these optimizations do significantly improve the multiplicity estimates.

## \*\*\*Building A Better Bloom Filter

*Relevance: Quick and easy obfuscation of a Bloom index (index-based transformation). Construct k hash functions from two simple “basis” hash functions in a way that is comparable (distributes different items, not matter their similarity, to independent indices) to expensive cryptographic hash functions. Since hash functions can be constructed at will with these properties, this is another way to make sure that similar documents (using the same secret key) will have different distributions of 0’s and 1’s to guard against information leaks possible through correlation analysis.*

In “Building A Better Bloom Filter”, Michael Mitzenmacher and Adam Kirsch show that only two “basis” hash functions are needed, in principle, to construct the k hash functions that minimize the false positive rate.

This has interesting theoretical implications. First, the randomness of the hash functions may not be as important as once suspected; this insight follows from the use of using only two well-behaved hash functions to construct the k hash functions. This means that only two hash functions per query need to be evaluated, which has the added benefit of being computationally simpler.

## \*\*\*Compressed Bloom Filters

*Relevance: Compression used as a weak for of data confidentiality, e.g., it makes it slightly harder to do analysis on the sequence of 1’s and 0’s of the Bloom filter to infer statistical properties about the document it represents.*

In “Compressed Bloom Filters” by Michael Mitzenmacher, an interesting question is addressed. Normally, one tries to find the optimal k = number of hash functions (in the sense of minimizing probability of false positives) for a bloom filter with a given size m (m bit array) and n members. Or, alternatively, for a given false positive probability, one tries to find the optimal size m. However, what if the metric of primary interest is the compressed size (e.g., transmission size) of the Bloom filter?

If a compressed size z is desired (assuming an optimal compressor according to Shannon’s source coding theorem), how do we find the optimal values of m and k such that the probability of a false positive is minimized? When the false positive probability is minimized for ab uncompressed Bloom filter for a given m and n, then k = ln2 m / n. This means that a particular bit in the Bloom filter will be 0 with probability ½. However, this also means that the sequence of 0’s and 1’s will have maximum entropy, and thus it is losslessly incompressible. Instead, if we choose a k such that finding a 0 is not ½, we can compress each bit, on average, to H(p) = -p log(p) – (1-p) log(1-p).

Analysis shows that the worst-possible value for k when optimizing on compressed size is, indeed, k = (ln2) m/n, which is amusingly the best possible value for the uncompressed Bloom filter. It is pointed out that the compression size improves as k goes to 0 or k goes to infinity. This, of course, is not tenable, but it is possible to make k < ln2(m/n) and expect better performance in the sense that the compressed bloom filter with a given false positive rate will be smaller than an uncompressed bloom filter with the same false positive rate.

*Additional relevance: These results have some additional relevance to using Bloom filters for encrypted searching in that transmission size (file size) may be more important than the uncompressed size, especially as the number of variations per word in the document that are inserted increases, e.g., 1-edit errors and limited wildcard searching, 1-word matches, 2-word exact phrases, …, n-word\*\*\* exact phrases.*

*\*\*\* Note: using more than 2-grams for exact phrase searching is not strictly necessary, since we can decompose a large phrase search into a chain of 2-word phrase searches—although this has a false positive rate independent of the bloom filter’s false positive rate), etc.*

## Weighted Bloom Filter

*Relevance: Unclear how to incorporate it, but after having read and digested it, I noted some similarities it had to some of my ideas with respect to how to optimally train a Bloom filter to do better than the classical naïve optimal values for m and k for a given n members and a given fp false probability.*

In “Weighted Bloom Filter” by Jehoshua Bruck, Jie Gao, and Anxio Jiang, they discuss the effect that non-uniform query frequencies will have on the false positive rate. They don’t put it this was exactly, but they are making the point that to reduce the probability of a false positive, one should minimize:

P[false positive] = P[false positive | query] \* P[query]

The more likely a nonmember query is, P[query], the greater effect it will have on the probability of a false positive. They suggest using query frequency and member likelihood data to tune the Bloom filter such that more frequent / more likely queries / members use more hash functions, and less frequent / less likely queries / members use fewer hash functions. In this way, infrequent queries will only check a few positions, which means there are fewer chances for a 0 to occur, thus increasing the probability of a false positive on them. However, in turn, since infrequent query members have set fewer bits, this makes it so that there are more 0’s in the sequence of bits. This means that more frequent nonmember queries will have a higher chance of seeing a 0. Thus, they weight things more appropriately in correspondence to the effective they will have on the false positive rate. They call this the weighted bloom filter.

*Note: This has some similarity to some of the ideas I was considering, except that I am learning hash function parameters from the frequency data, whereas they are learning how many hash functions to use for different kinds of members/queries. Since my approach has more degrees of freedom, there is far more opportunity for optimization, but that comes at the cost of increased Bloom filter construction time.*

## \*\*\*Practical Techniques for Searches on Encrypted Data

*Relevance: online, n-gram (Boolean keyword search: no ordered documents by some relevancy measure)*

In [9], one of the earlier papers presented on encrypted searching, the author points out that since the advent of cloud storage, many individuals and organizations would like exploit these services but do not trust cloud storage providers to secure their privacy. The naive solution is for the client to encrypt the data and upload the encrypted content to the CSP. However, if the client is on a device with limited resources then this may not be reasonable solution, especially as the volume of sensitive information expands. Rather, they need some way to allow the CSP to obliviously search through the confidential data without needing to know the contents of the data or query.

They propose a solution to this problem, with the limitation that only simple Boolean keyword searching is possible. On large document stores, a Boolean keyword search will fail to return many relevant documents (e.g., missing a term due to a synonymy or alternate word spellings) and return too many irrelevant documents (e.g., a document contains all of the query keywords, but due to lack of proximity bare no relation to the intended meaning).

In addition, their solution is slow (linear scan through the documents—online) and may return false positives; they point out that the user will be able to eliminate all false positives after downloading and decrypting the document, and conclude that it therefore is not a problem so long as it does not overwhelm the communications channels. They fail to recognize the scarcer resource: human attention. People do not wish to sift through irrelevant results.

## \*\*\*Public-key encryption with keyword search

*Relevance: online, n-gram (Boolean keyword search: no ordered documents by some relevancy measure)*

In “Public-key encryption with keyword search”, D. Boneh, G. D. Crescenzo, R. Ostrovsky, and G. Persiano explore searchable encryption using public keys such that, for instance, someone may provide Alice with a encrypted document that is searchable using Alice’s private key. This solution has many of the same problems as the solution presented in “Practical Techniques for Searches on Encrypted Data”, i.e., it’s slow—does a linear scan to find a query’s term in every document—which is not practical for reasonably large data stores—rather than representing the document with an efficient retrieval index.

## \*\*\*Secure Conjunctive Keyword Search over Encrypted Data

In [1], they propose a system which permits secure conjunctive queries for certain keywords (trapdoors) on a given set of fields, like the “From” field in an email. By secure, they mean that given access to a set of indexes for encrypted documents and a freely chosen set of trapdoors, adversaries—like an untrusted cloud storage provider—must not be able to learn anything about the encrypted documents except whether it matches those specific trapdoors.

Their work demonstrates a slight important over the single keyword searching discussed in [9], but their solution is still rather limited. They still only perform exact string matching (instead of approximate string matching), their solution inflexibly requires the document creator to tag specific keyword fields for searchability—in their case, these field names are leaked—and finally they do not even consider adversaries which consider historical data (previous queries and results) when making inferences. When this is taken into consideration, users are vulnerable to a subtler form of information leak.

## \*\*\*Fuzzy Keyword Search Over Encrypted Data In Cloud Computing

In “Fuzzy Keyword Search Over Encrypted Data In Cloud Computing” they propose a mechanism to address the limitation in which only exact matches on keywords are allowed. For instance, their solution can handle approximate matching on typographical errors. To accomplish this, when constructing the secure index, for each term in the document, add all k-edit error patterns, where an error is an insertion, deletion, or substitution of a character. For example, to allow up to a 1-edit error on the keyword “age”, expand it to {age, \*age, a\*ge, ag\*e, age\*, \*ge, a\*e, ag\*}, where the \* represents any character.

This represents an improvement over previous methods with respect to enabling approximate matching. This solution somewhat resembles well-established methods in IR (e.g., permuterm indexes) but applied in the context of encrypted searching. There remain many limitations with their solution, however. They do not consider the importance of term proximity and their solution inflates the size of the index significantly. Thus, more probabilistic solutions or just-in-time solutions may be preferable, e.g., simple spell correction.

## \*\*\*Phrase Search over Encrypted Data with Symmetric Encryption Scheme

Most searchable encryption schemes only allow matches on keywords, but in “Phrase Search over Encrypted Data with Symmetric Encryption Scheme” by Yinqi Tang, Dawu Gu, Ning Ding, and Hiaining Lu, a method for secure exact phrase matching on multiple words is elaborated on. Phrase searches consist of approximately 10% of web search queries, so this is an important innovation. Unfortunately, they require clients maintain a local dictionary on their computers to facilitate the capability. As long as such data must be maintained locally to perform searches, one may reasonable argue that local searchable indexes should be maintained instead. Local indexes, freed from many of the security concerns, would permit any sort of search operation without the need to communicate with server until a specific document is desired (thus less information leaks in the form of access patterns); the downside with this approach and their approach, of course, is that, local data structures must be maintained (and kept consistent with the document store). Additionally, in their searchable encryption construction, no context-aware (ranked relevancies) searching is afforded; a document is either deemed relevant or not relevant.

## \*\*\*Adding Compression to Block Addressing Inverted Indexes

In “Adding Compression to Block Addressing Inverted Indexes” by Gonzalo Navarro, Edleno Silva de Moura, Marden Neubert, Nivio Ziviani, and Ricardo Baeza-Yates, inverted indexes and their uses is discussed in considerable depth. The inverted index is the single-most popular (offline) index in use in IR due to its efficiency, speed, and versatility. The basic idea is to map each term to a set of positions, whether those positions are exact or approximate (e.g., blocks within a document or whole documents). The list of positions that a term maps to is called its postings list, and since the terms are sorted lexographically, any term’s postings list can be found be found in O(logN), where N is the number of unique terms in a document collection.

In this paper, they consider several different variations of the inverted index. One of the more interesting discussions concerns the use of compression—using Huffman coding where the symbol table consists of words rather than, say, letters. Using this encoding, when a search for a particular word is conducted, they are able to use the shortened codeword instead, which means that not only do we benefit from compression, but also reduced IO overhead and CPU overhead when performing string comparisons.

Another possible use of Huffman encoding is to put it to use as a very weak form of obfuscation; by keeping the mapping of symbols to encodings (bit strings) private, a very weak form of encryption is provided. However, this is not recommended for any IR system which requires data confidentiality, as it would be trivial for an attacker to reverse engineer the mapping using basic cryptographic analysis.

## \*\*\*Space-Code Bloom Filter for Efficient Per-Flow Traffic Measurement

Abhishek Kumar, Jun Xu, Jia Wang, Oliver Spatschek, Li Li

A classical Bloom filter only answers membership queries; it represents a set S approximately. However, what if you need to count the multiplicities of the members, e.g., what if a word in a text is seen multiple times and you wish to track the frequency? Then a Bloom filter is not appropriate. In this paper, the authors propose what is called a Space-Code Bloom filter. It combines multiple Bloom filters, such that when an item is added, one of those filters is randomly chosen. To answer the multiplicity question, count the number of matching Bloom filters, and then use some estimator, e.g., MLE, to answer the question, what is the most likely multiplicity that would cause that many matching Bloom filters?

## \*\*\*Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality

Piotr Indyk, Rajeev Motwani

To avoid the curse of dimensionality—or in other cases avoid having to explore a space that combinatorially explodes—this paper suggests using locality-sensitive hash functions for dimensionality reduction; that is, use hash functions in which the probability of a collision is high for “close” elements and low otherwise. (Distance preserving.) Thus, LSH functions are not at all like cryptographic hash functions; crypto hashes, like most hash functions, are designed to minimize the probability of collisions. Some examples of LSH functions:

* hamming distance functions, e.g., convert inputs A and B into bit strings, then the hamming distance is the number of bits different. Randomly sample k bits (out of n, k < n) from each string, and apply the hamming distance function to the reductions. The more likely two strings A and B are, the more likely they’ll map to the same hash value.
* Other examples include Phonetic algorithms, like Soundex, which map similar-sounding words to the same hash.

## \*\*\*Locality-Sensitive Bloom Filter for Approximate Membership Query

Yu Hua, Bin Xiao, Bharadwaj Veeravalli, Dan Feng

Bloom filters generally assume cryptographic hash functions, or at least hash functions which uniformly distribute over the bit positions. However, what if we relax this requirement? Then, we can choose hash functions which are more likely to test positively for non-members that look like members by using locality-sensitive hash functions.

NOTE: See my discussion on *Extremely Fast Text Feature Extraction for Classification and Indexing* for thoughts on the applicability of this in the context of encrypted searching.

## \*\*\*Extremely Fast Text Feature Extraction for Classification and Indexing

*Relevance: Offline. Obfuscation.*

George Forman, Evan Kirshenbaum

“Mix” up how an item hashes by not explicitly changing the hash function, but by altering a map which transforms an input x to an input x’, at which point x’ is hashed. Note that cryptographic hash functions accomplish this, as well, but they are computationally expensive.

Using this method, existing fast and efficient hash functions may be used with up to permute(N, k) variations, where N is the number of integers that a symbol can be transformed into and k is the number of symbols in the language, e.g., alphabet.

If the hash function is kept private, and only transformed inputs, x -> x’, are used as queries, this is similar to the method of using Huffman codes. It also has the same weaknesses.

*Additional relationships with research (that I am now no longer as keen on exploring anyway): Its discussion on representing strings as a hash vector had some interesting things to say, namely, that this representation serves equally as well as the bag of words (strings) representation when learning a classifier on it. This is an efficient, versatile way to represent hash functions: only one hash function template is needed, and by varying the symbol mapping we can have as many as we need. Then, learning hash functions in the context of minimizing false positives on probable search queries becomes relatively easy, e.g., a simple hill-climbing algorithm.*

*In the context of our research, this could still be used to learn hash functions (non-cryptographic) that work well to create hash functions that minimize false positives more effectively than if we assume that the inputs are uniformly distributed over the entire domain. However, such hash functions, being non-cryptographic, will leak more information (we could concatenate all terms in a query with a secret known only to the user(s), like in other approaches, but to minimize information leakage inputs should be uniformly distributed over the domain).*

*If we learn a hash function that minimizes false positives on cryptographic hashes of the members, this drastically reduce the effectiveness of being able to learn hashes which minimize false positives due to the lack of regularity in the inputs. In fact, theoretically it is no longer possible since all of the inputs have no resemblance to each other (due to the cryptographic hash); the most succinct description can do little better than a table lookup.*

*The alternative is to learn, in essence, locality-sensitive hash functions, where non-member inputs should be less likely to hash to a member input the more probable said non-member inputs are. There are more details I have considered which goes a long way towards resolving these issues, but I don’t want to waste much more time on it since it’s no longer a research direction. (And, moreover, it doesn’t seem as promising in light of the importance of confidentiality guarantees.)*

## \*\*\*Distance-Sensitive Bloom Filters

How do we get Bloom filters to answer approximate queries without, for instance, inserting many different variations of a given member, which causes the number of members to blow up.

Essentially, “Is x’ close to an element x of S?” where closeness is defined in terms of some measure, like the Hamming distance. This only works, however, for Bloom filters that use the same hash functions and are of the same size. This remains possible, however, in the sense that we can, for instance, construct a Bloom filter from an IR query using the same parameters as the Bloom filter(s) we wish to compare to; in this case, we can use a distance metric, like relative Hamming distance, to rank the secure indexes. However, in general, these constructions are problematic and had less than impressive experimental results.

## \*\*\*Network Applications of Bloom Filters: A Survey

Andrei Broder and Michael Mitzenmacher

Interesting read. Re-exposed me to the perfect hash, i.e., for a set with n members, a perfect hash can be constructed in which each member maps to a different integer in the range [0, 1, …, n-1]. If we combine this with the use of a cryptographic hash then we can approach the theoretical limits of bits per member given a certain false positive probability

* membership\_vector[perfect\_hash(x)] = crypto\_hash(x)
* To check if an x’ is a member, see if membership\_vector[perfect\_hash(x’)] equals crypto\_hash(x’). If x’ is really not a member, then it will test positively as a member with probability 1 / 2{number\_of\_bits\_output\_by\_crypto}.

However, I should note that while, in theory, this combination can achieve the theoretical limit (bits per member given a certain false positive probability), most implementations in practice don’t achieve this.

Also, it’s trivial to extend the implementation given above to a multiset that is linear in the number of members, n, e.g., if you want to be able to count up to K, then you will need log(K) bits per member, so n\*log(K).

## \*\*\*Efficient Fuzzy Search in Large Text Collections

Hannah Bast and Marjan Celikik

More treatment on fuzzy searching, e.g., edit distance to be tolerant to typographical errors, etc.

## Bloofi: A Hierarchical Bloom Filter Index with Applications to Distributed Data Provenance

Adina Crainiceanu

Summary: efficiently query a set of Bloom filters.

If the bloom filters use the same hash functions and same array sizes, then simple unions can be accomplished by simply ORing the bit vectors. A tree index, then, is where you start with the Bloom filters, then union adjacent ones together to make parent nodes. Do this again for the parent nodes until you have a single root node. The root node can quickly answer the query membership function, “Does any Bloom filter contain x?” If not, then no further processing required. If so, then while this may be a false positive, find do a depth first search on the tree: if a node answers no, then you can skip traversing that node. You do this until you have a set of leaf nodes which all answer yes.

In the worst case, every leaf node will answer yes, and so you must traverse the entire tree, performing O(n) query membership tests, where n is the number of leaf nodes. The best case is where only one of the leaf nodes will answer yes, in which case you only need perform O(log(n)) tests (in a binary tree, each interior node checks 2 children, and there are log(n) levels).

They also explore other tweaks, e.g., by observing that as you approach the root of the tree, the likelihood of false positives increases; in the worst-case, you have Bloom filters which are all set to 1’s, so answer ‘yes’ to everything. If a Bloom filter has all 1’s, then essentially create a short-cut from the root to that node’s children, because any parent of that node will also be all 1’s.

Notes:

I had an idea similar to the idea proposed in this paper, motivated by the following concern: even if each individual query with respect to a single document is serviced quickly, what if our document store consists of a large number of documents?

Since a large part of our work is evaluating the effectiveness of term proximity using multiple secure Bloom filter indexes per document (blocks), there is immediately more work for us to do.

Let’s assume the following:

1. Keyword weighting will be used: term frequency and inverse document frequency
   1. term frequency: a query term’s frequency for a document is either the number of blocks for which the term tested positively for membership (out of the |B| Bloom filters), or if using a construction which counts multiplicities (like Spectral Bloom filter), then we can approximate the frequency per block, also, and sum those up over all the blocks.
   2. inverse document frequency: for each term, out of how many documents in the document store did the term appear test positively? (i.e., if a term tests positively in at least one of the Bloom filter blocks for a document, then that counts as 1.)
      1. Immediately, this makes the most straightforward implementation of our relevancy scoring algorithm use two passes over all the documents.
      2. The temptation is to create a global index to keep track of this, but the problem with that is:
         1. This is only possible if we relax query confidentiality; this relaxation will permit term frequency analysis to be performed, i.e., the term frequencies in the global index can be matched with other known term frequency distributions to infer plausible matchings.
            1. Once such a matching is constructed, this serves as a prior. If this matching is not possible, then the prior is just the uniform distribution. As more information comes in, the posterior encrypted term -> term probability map may be updated, e.g., whenever Alice queries the document store using encrypted query terms X Y Z, she looks up stock quotes afterwards. The probability P[performed query X Y Z | looked up stock information] can be estimated from data; by Bayes rule, this is estimated to be P[looked up stock information | performed query X Y Z] P[performed query X Y Z] / P[looked up stock information]. P[performed query X Y Z] is the prior, and we want to use an informed prior to update the posterior, P[performed query XYZ | looked up stock information]. The better the prior, the better our posterior. So, by leaking such information on the prior, we make it a lot easier for an attacker to guess, with some probability, the