Encrypted Search: Enabling Standard Information Retrieval Techniques for Several New Secure Index Types While Preserving Confidentiality Against an Adversary With Access to Query Histories and Secure Index Contents

By Alexander R. Towell, Bachelor of Science

A Thesis Submitted in Partial
Fulfillment of the Requirements
for the Degree of
Master of Science
in the field of Computer Science

Advisory Committee:

Hiroshi Fujinoki, Chair

Gunes Ercal

Tim Jacks

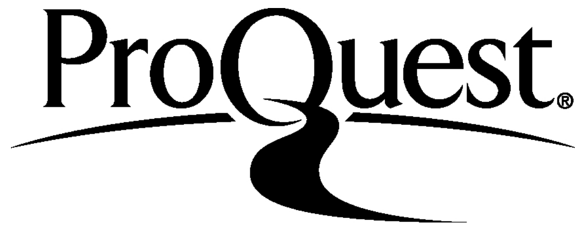
Graduate School
Southern Illinois University Edwardsville
September, 2015

ProQuest Number: 1601582

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.



ProQuest 1601582

Published by ProQuest LLC (2015). Copyright of the Dissertation is held by the Author.

All rights reserved.
This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

ABSTRACT

ENCRYPTED SEARCH: ENABLING STANDARD INFORMATION RETRIEVAL TECHNIQUES FOR SEVERAL NEW SECURE INDEX TYPES WHILE PRESERVING CONFIDENTIALITY AGAINST AN ADVERSARY WITH ACCESS TO QUERY HISTORIES AND SECURE INDEX CONTENTS

by

ALEXANDER R. TOWELL

Chairperson: Professor Hiroshi Fujinoki

Encrypted Search is a way for a client to store searchable documents on untrusted systems such that the untrusted system can *obliviously* search the documents on the client's behalf, i.e., the untrusted system does not know what the client is searching for nor what the documents contain. Several new secure index types are designed, analyzed, and implemented. We analyze them with respect to several performance measures: confidentiality, time complexity, space complexity, and search retrieval accuracy. In order to support rank-ordered search, the secure indexes store frequency and proximity information. We investigate the risk this additional information poses to confidentiality and explore ways to mitigate said risk. Separately, we also simulate an adversary who has access to a history of encrypted queries and design techniques that mitigate the risk posed by this adversary.

KEYWORDS: (Encrypted Search, Information Leaks, Perfect Hash Filter, Query Obfuscation, Secure Indexes)

ACKNOWLEDGEMENTS

I would like to recognize my better half, Kimberly Wirts, for her patience and encouragement.

I would like to thank my parents, Bill and Sharon Towell, for their constant support throughout the entire process.

I would like to express my gratitude to my thesis advisor, Prof. Hiroshi Fujinoki, for introducing me to my thesis topic and lending me his time and support.

I would like to thank the rest of my thesis committee, Prof. Gunes Ercal and Prof. Tim Jacks, for their insights and challenging questions.

TABLE OF CONTENTS

ABSTRACT ... ii

ACKNOWLEDGEMENTS ... iii

LIST OF FIGURES ... vi

LIST OF TABLES ... ix

Chapter

I. INTRODUCTION ... 1

II. REVIEW OF LITERATURE ... 3

- Confidentiality ... 3
- Information Leaks ... 6
- Online and Offline Searching ... 7
- Mapping Queries to Documents ... 9

III. RESEARCH OBJECTIVES ... 15

- Secure Indexes ... 15
- Relevancy Metrics ... 30
- Information Leaks ... 34

IV. EXPERIMENTS ... 41

- Inputs ... 41
- Outputs ... 42
- Platforms ... 43
- Global Parameters ... 43
- Adversary Simulation Results ... 44
- Secure Index Results ... 50

V. EXTENSIONS ... 84

- Set-Theoretic Queries ... 84
- Fuzzy Set-Theoretic Search ... 84
- Boolean Proximity Searching ... 86
- Caching Results ... 87

VI. FUTURE WORK ... 88

- Simulating an Adversary with Secure Index Access ... 88
- Mitigating Access Pattern Leaks ... 88

Semantic Search 89
Topic Search (Classification) 90
Letter N-Grams and Word N-Grams 91
Learning Optimal Parameters 91

VII. CONCLUSIONS 92

REFERENCES 94

LIST OF FIGURES

Figure Page

1. Overview of Encrypted Search on a Secure Index Database 16
2. Overview of Secure Index Construction 17
3. Perfect Hash Filter (Using A Minimal Perfect Hash) 20
4. The Perfect Hash Filter Secure Index (PSI) 22
5. The PSI Block-Based Secure Index (PSIB) 24
6. The PSI Frequency Secure Index (PSIF) 25
7. The PSI Postings List Secure Index (PSIP) 26
8. The PSI Minimum-Pairwise Distance Secure Index (PSIM) 28
9. Experiment #1 44
10. Unique Obfuscations vs Accuracy of Adversary Using MLE Attack 45
11. Obfuscation Rate vs Accuracy of Adversary Using MLE Attack 46
12. Experiment #2 46
13. Secrets vs Accuracy of Adversary Using MLE Attack 47
14. Experiment #3 47
15. History with Secrets vs Accuracy of Adversary Using MLE Attack 48
16. History vs Accuracy of Adversary Using MLE Attack 48
17. Number of Secrets vs Accuracy of Adversary Using MLE Attack 49
18. Experiment #4 49
19. Vocabulary Size vs Accuracy of Adversary Using MLE Attack 50
20. Experiment #5 50
21. Location Uncertainty vs Accuracy of Top 10 BM25 MAP Search Results 51
22. Mean Average Precision of Random Results 52
23. Experiment #6 52
24. Location Uncertainty vs Accuracy of Top 10 Mindist* Search Results 53
25. Experiment #7 53
26. Number of Secrets vs Compression Ratio 54

27. Number of Secrets vs Secure Index Build Time 54
28. Number of Secrets vs Secure Index Load Time 55
29. Experiment #8 55
30. False Positive Rate vs BM25 MAP 56
31. False Positive Rate vs Precision 56
32. Experiment #9 57
33. Obfuscations/Query vs BM25 MAP with 6 Terms/Query, 6 Words/Term 57
34. Obfuscations/Query vs BM25 Lag Time with 6 Terms/Query, 6 Words/Term 58
35. Obfuscations/Query vs BM25 MAP with 1 Term/Query, 2 Words/Term.......58
36. Experiment #10 59
37. Obfuscations/Query vs Mindist* Mean Average Precision 59
38. Obfuscations/Query vs Mindist* Lag Time 60
39. Experiment #11 60
40. Page Count vs Secure Index Size 61
41. PSIB and BSIB Intersection for Pages/Document vs Secure Index Size 61
42. Experiment #12 61
43. Blocks per PSIB/PSIB vs Secure Index Size 62
44. Blocks per PSIB/PSIB vs Compression Ratio 63
45. Experiment #13 63
46. Documents per Corpus vs Corpus Size 64
47. Experiment #14 64
48. Pages per Document vs Build Time 65
49. A Closer Look a Pages per Document vs Build Time 65
50. Experiment #15 65
51. Documents per Corpus vs Total Build Time per Corpus 66
52. Experiment #16 66
53. Pages per Secure Index vs Secure Index Load Time 67
54. Experiment #17 67

55. Documents per Corpus vs Corpus Load Time 68
56. Experiment #18 68
57. Pages per Secure Index vs Mindist* Lag Time 70
58. Experiment #19 70
59. Location Uncertainty vs Absolute Location Error..71
60. Experiment #20 72
61. Location Uncertainty vs Mindist* MAP...72
62. A Closer Look at Location Uncertainty vs Mindist* MAP 73
63. Experiment #21 73
64. Pages per Secure Index vs BM25 Lag Time...74
65. A Closer Look at Pages per Secure Index vs BM25 Lag Time 74
66. Experiment #22 75
67. Location Uncertainty vs BM25 MAP with 2 Words/Term, 3 Terms/Query ...75
68. Location Uncertainty vs BM25 MAP with 2 Words/Term, 2 Terms/Query ...76
69. Experiment #23 76
70. Pages per Secure Index vs Boolean Search Lag Time....................................77
71. A Closer Look at Pages per Secure Index vs Boolean Search Lag Time 77
72. Experiment #24 78
73. Percentage of Junk Terms vs Secure Index Compression Ratio......................78
74. Percentage of Junk Terms vs BM25 MAP 79
75. Experiment #25 79
76. Relative Frequency Error vs BM25 MAP with 2 Terms/Query 80
77. Relative Frequency Error vs BM25 MAP with 1 Term/Query........................80
78. Experiment #26 81
79. Compression Ratio vs Mindist* MAP 81
80. Experiment #27 82
81. Words per Term vs Precision and Recall...83

LIST OF TABLES

Table Page

1. Comparison of Confidentiality Techniques 3

2. Online Searching vs Offline Searching..7

3. Boolean Search vs Rank-ordered Search 9

4. BNF Query Grammar 9

5. BNF Hidden Query Grammar...16

6. Testbed System for Experiments 43

7. BNF Set-Theoretic Query Grammar...84

8. BNF Fuzzy Set-Theoretic Grammar...86

CHAPTER I
INTRODUCTION

Organizations are eager to take advantage of cloud storage. Cloud storage is:

- *Reliable*—storage management is delegated to expertise of cloud storage provider (CSP).
- *Scalable*—as storage needs change, pay more or less as needed.
- *Cost-effective*—cloud storage providers are efficient (division of labor).
- *Accessible*—storage can be accessed anytime and anywhere.
- *Sharable*—every resource (e.g., directories, files) has a URL.

However, a significant disadvantage to cloud storage is loss of control over confidentiality. An organization loses control over an unencrypted document's confidentiality when it is hosted in the cloud.

In [1], one of the earlier papers presented on *Encrypted Search*, the author observes that many individuals and organizations wish to exploit cloud storage services, but do not trust the CSP with their confidential data. That is, organizations trust the CSP with storage logistics but they do not trust the CSP with their need for *confidentiality*.

The naïve solution to regaining control over confidentiality of cloud-hosted documents is achieved using encryption. Before a document is uploaded into the cloud, it is encrypted. Subsequently, to access this document, clients download it to a trusted machine and decrypt it.

Often, the documents of interest are not known in advance. Consequently, the ability to perform searches over a collection of documents is needed. In the naïve solution, this entails the following sequence of actions:

(1) Download the collection of encrypted documents to a trusted machine.
(2) Decrypt the encrypted documents.
(3) Search through the decrypted documents using any available search facility on the trusted machine.

This approach breaks down if an organization has a large collection of confidential documents. It is both time consuming and costly in terms of transmission costs (i.e., downloading a large corpus) and energy costs (i.e., decryption is computationally demanding).

The larger the collection of confidential documents, the more costly the naïve solution is. This inefficiency is especially evident on resource-constrained machines, e.g., smartphones with limited bandwidth and energy.

What is sought is some way to allow the CSP to search the encrypted documents on behalf of clients, and returning only those documents relevant to client queries. Furthermore, this should be done without revealing the contents of documents (*data confidentiality*) nor the contents of client queries (*query privacy*). In other words, the CSP should be able to perform *oblivious* searches on

behalf of authorized users. Finally, the CSP should not be able to initiate meaningful searches except on behalf of authorized users.

The ability to search over a collection of encrypted documents without needing to decrypt them first is known as *Encrypted Search*[1]. In light of the advantages of cloud storage, *Encrypted Search* has recently gained a lot of traction in the research community.

Many solutions to this problem have been proposed. Chapter 2 describes existing work in *Encrypted Search* with attention paid to the strengths and weaknesses of various proposals.

[1] *Encrypted Search* is a narrow specialization of the computationally demanding field of fully homomorphic encryption (FHE) [41].

CHAPTER II
REVIEW OF LITERATURE

CONFIDENTIALITY

The first issue to address is confidentiality and its implementation. That is, the techniques employed to prevent disclosing information to unauthorized parties, like a server hosting the confidential documents. We consider three primary approaches: compression, obfuscation, and encryption.

Table 1 Comparison of Confidentiality Techniques

Methods	Advantages	Disadvantages
Compression [2], [3]	Very space efficient. Fast and easy to implement.	To serve as an obfuscator, users must maintain a separate symbol-mapping table.[2] Serves as a substitution cipher; may be broken through cryptanalysis.
Obfuscation [4], [5], [6]	Effective against non-skilled adversaries.	High insider risk. Too weak for cryptographic use.
Encryption [7], [8], [9], [10], [11], [1], [12], [12]	Provides strong guarantees on data confidentiality (assuming the key is kept private). Very well understood (modern cryptography).	Depending on the types of information leaks prevented, certain IR operations are problematic, e.g., if query privacy is provided ranking document relevancy is difficult. Strong encryption is slow and still subject to information leaks.

Compression. In [2], the idea of using a theoretically optimal Huffman [3] encoder is proposed in the context of information retrieval (without consideration given to the unique needs of *Encrypted Search*), where the symbols consist of words rather than letters. To obscure the contents of a document, one could substitute the words with Huffman codes and keep the symbol table a secret (which is essentially a large secret key) to serve as a weak substitution cipher.

When combined with an inverted index (see page 8), it is reasonably space-efficient and fast. Unfortunately, it cannot be taken seriously; it would be too easily compromised.

[2] Since users must already maintain a separate symbol-mapping table, they could just query this structure instead.

Obfuscation. The previous section discussed a method to obfuscate by remapping words to Huffman codes. In general, any symbol substitution technique may be used to obfuscate the contents of documents. The primary distinction between obfuscation and compression is obfuscation obscures the contents of documents by mapping symbols in the domain of D to other symbols in the domain of D [4] [5], whereas compression maps codes in one domain to codes in another domain (e.g., character strings to bit strings).

Conceptually, there are two types of obfuscation techniques. Encoding-based transformations apply general rules to each symbol, e.g., remapping the character sequence "$A\ B\ C$" to "$B\ C\ D$" by adding 1 to each of their ASCII codes. Alternatively, indexed-based transformations apply unique transformations to inputs, e.g., a one-to-one mapping from *John Smith* to *Person 192353*.

The symbol remapping instructions—essentially a secret key—must not be disclosed to unauthorized parties. In general, these techniques are vulnerable to substitution cipher attacks.

Encryption. The previously mentioned confidentiality techniques serve the same purpose: convert communicated information into a secret code so that unauthorized people cannot understand it. Cryptography is a more secure and general approach to accomplish this.

In cryptography, the unencrypted data is called plaintext. To encrypt (convert) plaintext into an unintelligible format, called ciphertext, the plaintext is input into an encryption function along with a secret key. The inverse of the encryption function is the decryption function, which takes the ciphertext and a secret key (either the same key, as in symmetric encryption, or another paired keyed, as in asymmetric encryption), and outputs the plaintext. For each cryptographic key k in the key-space (all possible keys), the encryption function maps a given plaintext c' to a different and unique ciphertext c.

$$enc_k(c_1') = c \neq enc_k(c_2'), c_1' \neq c_2'$$

To decrypt c, one must not only be in possession of the decryption algorithm, but also the secret key, i.e., $dec_k(c) = c'$. In designing security, one should assume Kirchhoff's principle, "only the secrecy of the key provides security." Specifically, the cryptographic algorithm is presumably already known to untrusted parties and, thus, for confidentiality the secret key must be kept private.

Without knowing the secret key, an unbroken encryption scheme requires an adversary to do a brute force attack over the entire key space; for a given cipher text c, iterate through all keys in the key space, feed the decryption function with the given key and cipher text, and decide on the plausibility of the decoded plaintext c'. For example:

$$\text{candidate key } k = \underset{k \in keyspace}{\text{argmax}} \text{ likelihood}\big(dec_k(c)\big),$$

where likelihood uses some language model to estimate the likelihood of the decoded ciphertext $c' = dec_k(c)$. To make brute force attacks intractable, the key-space must be extremely large.

Symmetric encryption. In the context of *Encrypted Search*, two different types of encryption have been used, symmetric encryption and asymmetric (public-key) encryption. Symmetric encryption uses a single key for both encryption and decryption. Compared to public key encryption, it is less computationally demanding. However, the downside is, a secure channel must be used to communicate the secret key if multiple parties need to be able to use it. The earliest examples of *Encrypted Search* used symmetric encryption [7] [1].

Public-key encryption (asymmetric encryption). Boneh [13] proposed an *Encrypted Search* scheme based on public-key encryption. Using public key encryption addresses a weakness found in earlier proposals. In symmetric encryption, if Bob wishes to make it so that $Alice$ can search a confidential document, they must first agree on a secret key. The problem with this approach is, how do they agree on a key without revealing it to others? In public-key encryption schemes, no such problem arises: he may use her public key. Thus, only $Alice$, who has the corresponding private key, may search the confidential document[3].

One-way cryptographic hash functions (trapdoors). In a secure index (see offline searching), which is independent of the document it represents, there is no need (nor desire) to be able to reconstruct the document from the information in the index. This particular relaxation offers a more attractive option: do not use a decipherable encryption scheme at all. Rather, use a one-way hash function [14] [15], ideally something that approximates a random oracle, and "filter" [16] the document (plaintext) through it, e.g., insert the one-way hash of each word in the document into the secure index. These one-way hashes are known as *trapdoors*—easy to compute and very difficult (if not impossible) to invert.

In theory, it is nearly impossible to determine which terms a document contains simply by looking at the secure index since the trapdoors are practically one-way. Indeed, in many constructions, they are truly one-way (non-invertible), e.g., multiple terms may map to the same cryptographic hash value (collision). In this case, it is impossible to determine, with certainty, which terms the document contains.

There are compelling advantages to this approach. First, it is far less computationally demanding; evaluating a one-way hash is far less computationally demanding than executing a decipherable encryption scheme. Second, because one-way hash functions are non-invertible and pre-image resistant[4], even if the secret keys are disclosed to unauthorized users, this does not necessarily compromise the contents of the actual document (except by allowing whatever search facilities the secure index permits).

Note that one or more secret keys, as with encryption, may be used to manage search authorization and to mitigate pre-image attacks (the secrets serve as hash salts).

[3] In this naïve scheme, only $Alice$ can search the encrypted document; in more sophisticated approaches, multi-user encrypted searching schemes are possible.
[4] Given a hash, it should be difficult to find an input for the hash function that outputs the given hash.

INFORMATION LEAKS

Goh [16] contends that *Encrypted Search* should not reveal any information about the contents of confidential documents unless one or more secrets are known. Furthermore, if such secrets are known, the only information that should be revealed is approximate knowledge about whether a given document is relevant[5] to given query. Thus, even if an untrusted party—like the CSP—examines a hidden search query, it should neither be able to determine the contents of the query nor the contents of the document, affording both data confidentiality and query privacy[6].

Most *Encrypted Search* schemes [16] [1] [13] [17] have this as the primary objective, but only a few solutions [18] considered maintaining this objective in the presence of a determined adversary who has access to user activity histories, e.g., hidden query histories.

There are many different and subtle ways information can be disclosed—or *leaked*. This remains true even if we assume an unbreakable encryption scheme is being used.

Document confidentiality. Even if a strong cryptographic scheme is being used, information may still be leaked. Consider the following. For each document in the collection, the words in a given document are passed through a one-way hash and that hash is directly inserted into the index. Since this is a substitution cipher for small blocks (words), it is vulnerable to substitution cipher attacks.

For instance, since it is likely word frequencies in the confidential corpus are similar to the word frequencies found in other corpora, statistical frequency analysis can be used to construct probable cipher string to plaintext word mappings. For reasons similar to this, Goh [16] argued traditional hash tables are not suitable for use as secure indexes. However, on page 37 we construct a theoretical adversary that may be able achieve reasonable success at compromising any secure index, emphasizing the need for high false positive rates and secure index poisoning.

Query privacy. The same argument for data confidentiality also applies to query privacy. In the extreme case, queries may be sent in plaintext, and thus no query privacy is afforded. Thus, an adversary can determine what users are interested in and which documents are relevant to a given query. For example, the adversary may construct a model of an encrypted document by taking a set plaintext queries and observing its relevance to each of them—in the case of Boolean search, does it contain this keyword? Thus, it is vulnerable to basic dictionary attacks. Indeed, they may be successful at reconstructing close approximations of the contents of confidential documents if phrase searching is supported.

A simple solution is to cryptographically hash each term in a query, as elaborated on page 5. However, since it is reasonable to assume the hash algorithm will be known to adversaries, they may proceed the same as before. Thus, each hash term should be concatenated with one or more secret keys. As long as the keys are kept secret, only authorized users may meaningfully submit queries to the secure index. However, note that if a cryptographic query for a specific term always looks the same, then an adversary may slowly build up a frequency table of cryptographic hashes by observing query histories and use this information to mount a substitution cipher attack. For

---

[5] Note that the reference to a confidential document can be encrypted to prevent disclosing information about the contents of the document based on its filename.

[6] *Encrypted Search* schemes that protect against document confidentiality leaks, query privacy leaks, and access pattern leaks may also be used to enable plausible deniability, especially if hash collisions (see page 34) are probable.

example, using a frequency table of plaintext English words, an adversary may be able to determine an accurate mapping from cryptographic hashes to plaintext English words. In our research, we will explore practical methods to mitigate the risk posed by such an adversary.

Access patterns and implicit information. To exacerbate matters, even if an *Encrypted Search* scheme provides robust data confidentiality and query privacy, access patterns may still leak information. For instance, what is the distribution of (encrypted) documents a user has retrieved over time? Users may show interest in different documents, in which case statistical clustering may reveal associations among users.

Implicit information may also be leaked. For example, if a hidden query is followed by another action, like checking stock prices, this correlation may be used to infer properties about the query and the corresponding documents that are returned in response to it. To mitigate implicit information disclosure related to a user's *Encrypted Search* activities, Pinkas [18] proposed the use of *Oblivious RAM*.

Online and Offline Searching

Table 2 Online Searching vs Offline Searching

Methods	Advantages	Disadvantages
Online [7], [10], [11], [1]	Exact phrase matching is easy—does not increase size like in other solutions. Reasonably space-efficient (if combined with Huffman coding).	Very vulnerable to substitution cipher attacks. Sequential search—impractically slow for large-scale use.
Offline: Inverted index [8], [2]	Efficiently supports query operations necessary for rank-ordered search. Reasonably fast trapdoor lookups—$O(log\ n)$ for a document with n words. Reasonably space-efficient (if using Huffman coding).	Very vulnerable to substitution cipher attacks. Potentially vulnerable to preimage attacks.
Off-line: Bloom filter [16], [19]	Space efficient (nearly optimal)—can trade accuracy for space-complexity. Reasonably fast trapdoor lookups.	k hash functions must be evaluated per trapdoor (per query term).

Online searching. Online search performs a sequential search on the document cipher [1]. To be able to perform encrypted searches on such a cipher, a large block cipher may not be used; rather, each searchable term must be encrypted separately to facilitate exact string matches on the encrypted query terms. Thus, this is an easily compromised substitution cipher.

Moreover, as pointed out in [17], proposals based on online searching, in light of their linear time complexity, are not appropriate except in limited contexts. For example, they may be appropriate if the purpose is to allow an email server to obliviously scan the "subject" field of incoming emails for the keyword "urgent."

Offline (index) searching. Offline indexes are data structures that store a representation of the document (or documents) in which rapid, efficient retrieval is facilitated. Most of the recently proposed *Encrypted Search* constructions are based on offline indexes [16] [19] like Bloom filters. An overview of offline-based solutions can be found in [13].

Inverted index. In [2], a possible approach to a secure index is given in the form of an inverted index. Previously, the inverted index was discussed in the context of using Huffman codes to serve as an efficient but non-secure (easily compromised through frequency analysis) substitution cipher.

The reason the Inverted index reduces to a substitution cipher is related to the way in which searching is performed—that is, a binary search on a sorted array of terms. Each element in the array must (nearly) uniquely identify a single term. Thus, even if encrypted, compressed, or obfuscated transformations of terms are stored in the index, it still amounts to uniquely substituting each term with some transformation of that term. This makes it susceptible to cryptanalysis (e.g., frequency analysis) and pre-image attacks. Despite the popularity of the inverted index in information retrieval, its vulnerability to cryptanalysis makes it an unsuitable choice for a secure index.

Bloom filter. A Bloom filter [20] is a probabilistic (approximate) set in which false positives on membership tests are possible. While it does not possess the theoretical optimal space efficiency of $n \log_2 \frac{1}{\varepsilon}$ bits for a set with a false positive rate ε and n members, it is still reasonably space efficient requiring only $1.44n \log_2 \frac{1}{\varepsilon}$ bits.

Operationally, it consists of a bit vector of size m (all initially set to 0) and k hash functions. For each member, use the k hash functions to map it to k (possibly non-unique) indexes in the bit vector, setting each mapped bit to 1.

To verify an element is a member of the set, check to see if each of its k hash positions is set to 1. On the one hand, If any are 0, then it is certainly not a member (false negatives are not possible). On the other hand, if all of them are set to 1, we assume it is a member with a false positive rate ε. That is, one or more actual members may have caused those k bit positions to be set to 1.

It is straightforward to construct a secure index from a Bloom filter. For each term in the document (words, n-grams, or other searchable terms), insert it into the filter. To prevent unauthorized users from querying the index, do not insert the plaintext terms; rather, insert some transformation of them. Ideally, a trapdoor will be used, which is just a one-way (cryptographic) hash function applied to each term concatenated with one or more secrets, e.g., $insert(bloomfilter, hash(term_a |\ secret))$.

To harden the Bloom filter from cryptanalysis, the same term in separate documents ought to map to different index positions in the Bloom filter. In [16], it is recommended that the document id be appended to the trapdoors during the construction of the secure index, e.g., $insert(bloomfilter, hash_2(hash_1(term_a |\ secret)\ |\ doc_id)$. Likewise, during the construction of hidden queries (which consist of trapdoors), the user must apply the same transformations. Since $secret$ is unknown to unauthorized parties, like the server, they are unable to meaningfully query the secure index.

Problems with the Bloom filter. However, there is a notable problem with secure indexes based on Bloom filters: they need to evaluate k hash functions per trapdoor, where $k = \log_2 \frac{1}{\varepsilon}$. For large corpora consisting of N documents, this could be a significant drawback, requiring $O(Nk)$ hashes per trapdoor.

Another problem with them, as previously described, is they do not efficiently support multiplicities (multi-sets) nor do they efficiently support other types of queries, e.g., *where is a member located*? This complicates scoring functions like term weighting and proximity weighting.

MAPPING QUERIES TO DOCUMENTS

The method in which a query is mapped to (or ranked according by) a set of documents is a very important topic in information retrieval, but it is often neglected in *Encrypted Search*. There are two primary ways to perform this mapping function: Boolean keyword searching or ranking documents according to their relevancy to a given query.

Table 3 Boolean Search vs Rank-ordered Search

Methods	Advantages	Disadvantages
Boolean keyword search [7], [11], [1], [13], [21], [22] [23]	Simple. Fast queries.	Documents are either relevant or irrelevant to a query (no degree of relevancy), so recall and precision suffer.
Rank-ordered search [24], [25], [26], [27], [28], [17]	Much better precision and recall on results. Draws from extensive research in the IR community.	Answering queries is more computationally demanding (this may especially important in the context of cloud computing, where every second of CPU time is charged). More information about documents must be leaked to support degrees of relevancy.

Definition of a query. A query represents an *information need*. In practice, it is a string of terms, where a term is either a keyword or an exact phrase surrounded by quotes. In BNF notation, the syntax for a query is:

<query>	::=	<term> | <term> <query>
<term>	::=	<exact_phrase> | <keyword>
<exact_phrase>	::=	"<keywords>"
<keywords>	::=	<keyword> | <keyword> <keywords>
<keyword>	::=	<alphanumeric><keyword> | <alphanumeric>
<alphanumeric>	::=	a|b|…|z|0|1|…|9

Table 4 BNF Query Grammar

Consider the following query:

> *volunteer "doctors without borders"*

This query consists of two terms: the keyword *volunteer* and the exact phrase *doctors without borders*. When conducting a search on a collection of secure indexes, it will look for that exact

phrase and keyword. It will not count *doctors with borders* as a hit since the phrase must exactly match[7].

To support exact phrase searching, a secure index may store both the unigrams (keywords) and bigrams found in the document it represents. Any query with an exact phrase consisting of more than two keywords may then be converted into a chain of bigrams (the *biword* model, page 12).

Boolean search. Encrypted Boolean search [1] looks for a particular word or words [13] in a set of documents. In Boolean search, a document is either considered relevant (e.g., contains all of the terms in the query) or is considered non-relevant to a given query.

In addition, while there are some Boolean search techniques that allow for approximate matches or are tolerant of errors (like typographical errors), most encrypted search techniques must still rely upon some form of exact string matching due to the way the words in documents are transformed (for confidentiality) using one-way hash functions (trapdoors). This need for confidentiality complicates many standard information retrieval techniques.

For example, if tolerance of typographical errors is desired (as measured by edit distance, see page 11), the Levenshtein distance algorithm may be used in non-encrypted searching. However, since the words in a document are cryptographically hashed, this algorithm is useless (i.e., two words with an edit distance of 1 should hash to completely different strings). The solution proposed by Li [11] addresses tolerance of typographical errors by pre-computing and including all error patterns up to k errors directly in the index[8], upon which simple exact string matching may thus be performed.

Conjunctive keyword search. In [7], the authors propose a system which permits secure conjunctive queries for certain keywords 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 encrypted keywords (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 improvement over the single keyword searching discussed in [9], but their solution is still rather limited. First, they still only perform exact string matching. Second, their solution inflexibly requires the document creator to tag specific keyword fields for search-ability[9]. Finally, like most other solutions, they do not consider untrusted parties that consider historical data [18].

---

[7] Unless approximate searching or error tolerance is allowed (see page 11).
[8] Given the space complexity of this approach, it may be more useful to include only probable typographical errors.
[9] Moreover, these field names—although not the actual values—are revealed to adversaries (information leak).

Approximate search. In [12], the authors point out that Google's primary problem is finding ways to return fewer, more relevant results so that users do not have to sift through too many pages (many users only check the first page of results) even if this reduces the recall of the system. Thus, Google is motivated to improve the precision at potentially significant cost to recall (see page 30 for a discussion on precision and recall).

However, in vertical search— such as *Encrypted Search* facilities on a store of confidential enterprise documents—there is far more concern over not missing or overlooking relevant documents since there may be so few relevant results to begin with. That is, unlike Google, recall is equally if not more important than precision. To this end, various techniques—like approximate keyword matching—may be used to increase the recall.

Locality-sensitive hashing. In locality sensitive hashing, the notion is to map similar (according to some distance measure) items to the same hash. This is an especially good fit in the context of *Encrypted Search* since, typically, only exact matches on hash strings are possible.

To avoid the curse of dimensionality, locality-sensitive hash functions may be used as a form of dimensionality reduction [29]; that is, use hash functions in which the probability of a collision is high for *close* (according to some distance measure) elements and low otherwise. Thus, LSH functions are not at all like most hash functions; most hash functions are designed to minimize the probability of collisions, but LSH hashes are designed to maximize certain kinds of collisions.

For instance, in [30] the authors observe that Bloom filters generally assume the use of hash functions that uniformly distribute over their domains. However, if this requirement is relaxed, then locality-sensitive hash functions may be chosen such that input that is *close* to actual members (in the Bloom filter) will tend to hash to the same values and thus test as positive.

Stemming. Stemming may be thought of as another form of locality sensitive hashing. In stemming, morphological variations of a word are mapped to a single base form. By reducing such variations to a single form, in which the different variations have the same essential meaning, recall and precision may both be improved.

For example, if a user searches for "computing grades", it would seem the user would find "computed grade" relevant also. By not including this variation in the result set, recall and potentially precision suffer: recall suffers because relevant documents will be overlooked, and precision may suffer because less relevant documents may be returned in their place. Stemming has demonstrated itself to be a fast and effective technique to improve precision and recall. [31]

Phonetic algorithms. Phonetic algorithms are another form of locality sensitive hashing. The notion is to map words that sound alike to the same hash. Soundex is one of the more popular examples of this; it is an especially useful trick for approximate matches on the names of people.

Edit distance. In [11], a mechanism is proposed to address the limitation in which only exact matches on keywords are performed. In particular, they propose a construction that allows for matches on typographical errors or typical spelling variations, e.g., $color$ versus $colour$.

To accomplish this, when constructing the secure index, for each term in the document, add all k-edit distance patterns, where an edit is an insertion, deletion, or substitution of a character. For example, for a 1-edit error tolerance, the keyword age is expanded to $\{age, *age, a*ge, ag*$

$e, age*, *ge, a*e, ag*\}$, where $*$ represents any character (wild-card). Thus, if age fails to match, the query can be automatically expanded to each of those variations in turn until a match is found.

Wild-card matching. Wild-card [32] searches can be quite useful. For example, if users are uncertain about how to spell a particular word, they can use wildcards to represent their ignorance, e.g., instead of "tomorrow", they may type "to*row". Or, as another example, the user may seek multiple variations of a word, e.g., "*night" for "night" or "knight". The solution proposed in [11] on edit distance may be repurposed to implement wildcard searching.

Exact phrase matching (word n-grams). Phrase searches consist of approximately 10% of web search queries, but many *Encrypted Search* schemes only allow matches on keywords. In [33] a method for secure exact phrase matching is considered. Unfortunately, clients must 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 reasonably 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 the server until a specific document is desired.

Biword model. As long as an index supports bigram queries, any exact phrase search may be approximated as a series of bigram Boolean queries. This is known as the *biword* [26] model. For example, to find the exact phrase, *hello dr fujinoki*, perform a Boolean search for the bigrams, *hello dr* and *dr fujinoki*. This allows for false positive, as this search will also match any document in which *hello dr* and *dr fujinoki* are present but non-adjacent to each other.

Rank-ordered search -- degrees of relevancy. As pointed out in [17], most *Encrypted Search* research focuses on Boolean search, where a document is either relevant or non-relevant to a given query—that is, there are no degrees of relevancy. For instance, a Boolean search may consider a document a match—that is, relevant—if and only if all of the terms in the query are matched in the document. This poses 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 rank-orders documents according to some measure of their degree of relevancy to a given query.

In modern information retrieval systems, scoring the relevancy of a document to a query is one of its most important functions. If standard scoring techniques can be employed in *Encrypted Search,* its utility could be significantly improved. However, in *Encrypted Search*, a server obliviously searches over a collection of encrypted documents. The server's ignorance (which is needed to preserve confidentiality—i.e., confidentiality of queries and documents) complicate many traditional scoring techniques.

For instance, a vector-space model is commonly used in which documents are represented as unit vectors, where each dimension of the vector corresponds to a term's normalized *tf-idf* weight (see page 13). With this representation, an efficient similarity measure between two documents (or between a document and a vectorized query) is the *cosine similarity*[10] [34] measure. However, in *Encrypted Search*, the terms in a document should not be known a priori. Rather, such information should only be approximately learned through user submitted queries (and only in terms of

---

[10] $\text{cosine_similarity}(d_1, d_2) = \vec{v}(d_1) \cdot \vec{v}(d_2)$, where $\vec{v}(d_i)$ is the unit vectorized representation of document d_i.

trapdoors—cryptographic one-way hashes). Thus, documents may only be modeled using this more limited information[11].

In this paper, we will explore other more compatible measures, e.g., instead of *cosine similarity* one can score the relevancy of a document only with respect to the terms in the query, e.g., a simple summation.

Term importance weighting. Term (keyword) weighting [27] is based on two fundamental insights. First, some of the terms in a query will occur more frequently in one document compared to another document. When scoring the relevancy of documents A and B to a query term t, if t appears more frequently in A than B then A should be considered more relevant all other things being equal.

$$\mathrm{tf}_{weight}(t,d) = f(\text{frequency of term } t \text{ in document } d),$$

where f is a monotonically increasing function with respect to t (e.g., the identify function).

The second insight is that some of the terms in the query will be in a larger proportion of the documents in the corpus. These terms, therefore, carry less meaning—that is, they have less discriminatory power since they appear in a larger fraction of the documents. Conversely, some of the terms in the query will be very rare or even unique in a corpus, and thus they have more discriminatory power.

For example, the term "the" is in nearly every document—it serves as linguistic glue— but the term "acatalepsy" is in very few documents. The more discriminatory power a term has, the more weight it should be given when scoring a document's relevancy to the query.

$$\mathrm{df}_{weight}(t,D) = g(\text{number of documents in } D \text{ containing } t),$$

where g is a monotonically decreasing function with respect to t (e.g., the log of the inverse).

Combining both of these insights, a measure that is sensitive to both of the *rarity* of a term in a corpus and the frequency of a term in an individual document may be devised.

$$\mathrm{tf_df}_{weight}(t,d,D) = h\left(\mathrm{tf}_{weight}(t,d), \mathrm{df}_{weight}(t,D)\right),$$

where $\frac{\partial h}{\partial tf_{weight}} \geq 0$ and $\frac{\partial h}{\partial df_{weight}} \geq 0$, e.g., a function that takes the product of the two inputs.

The tf-idf term weighting heuristic. A notable example of a term weighting heuristic is *tf-idf* and its variants. The *tf-idf* heuristic—term frequency, inverse document frequency—accounts for both the term frequency within a document and the inverse document frequency within a document collection when estimating the importance of a term:

$$\mathrm{df}_{weight}(t,D) = \mathrm{idf}(t,D) = \log\left(\frac{\|D\|}{\|\{t \in d, d \in D\}\|}\right)$$
$$\mathrm{tf}_{weight}(t,d) = f(t,d) = \text{raw frequency of term } t \text{ in } d$$

[11] The untrusted server could store the results of queries to learn more about the contents of documents over time, but this information should be both approximate (e.g., false positives on terms existing in documents) and incomplete. See chapter 0 for more information.

$$\text{tf_df}_{weight} = \text{tf_idf}(t, d, D) = \text{idf}(t, D) \times f(t, d)$$

Term proximity weighting. In [35], the importance of proximity of terms in keyword searches is considered. The fundamental principle can be demonstrated by considering the following: given two documents, $\text{doc}_1 = \text{A B C}$ and $\text{doc}_2 = \text{A D D D B C}$, doc_1 should be more relevant than doc_2 for the query $Q = \{A, B\}$ even though they both contain keywords A and B with the same frequency. Put simply, all other things being equal, the closer the query's terms are in a document, the more relevant the match.

Semantic search. In [25], the authors maintain that most search techniques—from simple Boolean search to vector-space tf-idf weighted scoring—are variations of syntactic search in which some form of string matching is performed combined with a method to estimate how important particular string matches are.

There are two major problems with the syntactic approach. First, different terms may be used to express similar meanings (depending on the context). This is referred to as synonymy. Second, the same term may be used to express different meanings (depending on the context). This is referred to as polysemy. Both of these problems may degrade recall and precision of search results.

Semantic search attempts to mitigate these problems by modeling the meaning of text, with the aim of more intelligently mapping an information need—as represented by a query—to a set relevant rank-ordered list of documents that satisfy the information need, e.g., does the meaning of the query correspond to any similar meanings in a document?

Modeling semantics is a more complicated problem than string matching. It may involve natural language processing to perform word-sense disambiguation, part of speech tagging, and named entity recognition. When combining this with ontological and semantic knowledge, the information retrieval system may begin to process search queries in a way that resembles a human's ability to understand the meaning of text.

There are also statistical techniques to model semantically related terms, like latent semantic indexing (LSI). There has been very little progress on either of these fronts in relation to *Encrypted Search*[12].

[12] Semantic search in the context of *Encrypted Search* may have an additional advantage: remove as many specifics as possible from the secure index but include its more general concepts. This may both improve relevancy of search results while erasing potentially compromising specifics.

CHAPTER III
RESEARCH OBJECTIVES

Our research will seek to contribute to *Encrypted Search* in three different ways. First, we propose several different secure indexes based on a probabilistic set similar to the Bloom filter, which we call the Perfect Hash Filter [36]. We compare them against each other, and against a secure index based on the popular Bloom filter data structure.

Second, we will explore the use of standard information retrieval scoring techniques while paying attention to confidentiality concerns. Specifically, in our experiments, we will assess the accuracy of various proposed secure indexes on different scoring techniques. This accuracy will be assessed with respect to the level of uncertainty (e.g., term location uncertainty), false positive rate, and secure index poisoning. The accuracy of a secure index's output will be measured with respect to a canonical index with perfect information implementing the same search criteria.

Finally, we will consider various ways in which information is leaked, and design and implement solutions to mitigate these leaks. We evaluate the effectiveness of these mitigation strategies by considering a hypothetical adversary who employs various attack strategies. We do this for both document confidentiality and query privacy.

SECURE INDEXES

Encrypted Search is facilitated by the concept of a *secure index*. Secure indexes are offline indexes. To make a confidential document searchable, a user must first construct a secure index for it (see page 17).

Submitting hidden queries to secure indexes. Assuming the *secure indexes* for a collection of confidential documents have been constructed and transmitted to the CSP, *Encrypted Search* proceeds as follows. Refer to Figure 1 for a visualization of the steps.

(1) Clients construct queries to find confidential documents relevant to their information needs. Queries will be sent over a secure channel to the *query processor.*

(2) First, the *query processor* concatenates a *secret* to each term. Second, it feeds each concatenated term into a one-way hash function. Finally, it generates an intermediate *hidden query* from the output of the one-way hash function and transmits the result to a *proxy query processor*. See page 16 for the definition of a hidden query.

(3) The *proxy query processor* concatenates a secret to each term in the intermediate hidden query with another secret and transmits the resulting *hidden query* transformation to the CSP. Note that the proxy only observes intermediate *hidden queries* (it does not know what the client is searching for), thus the proxy need not be fully trusted.

(4) The CSP iterates through the secure indexes in the DB, hashing each hidden term with the concatenation of the document ID (a reference, e.g., URIs), and performs the requested search function (e.g., rank-orders document IDs with respect to BM25).

(5) The CSP returns the top-k results[13].

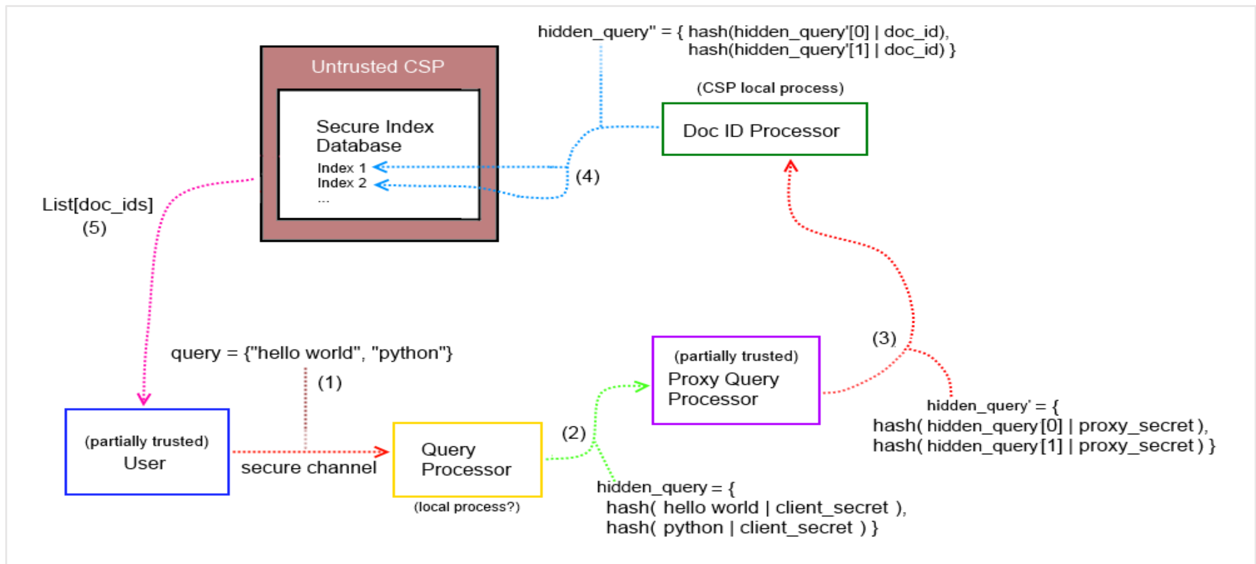


Figure 1 Overview of Encrypted Search on a Secure Index Database

Definition of a hidden query. A hidden query is a cryptographic transformation of a query (see page 9 for the definition of a query). Each term is converted into a hidden term; that is, each keyword is converted into a trapdoor using secret $s \in secrets$, i.e., $keyword \rightarrow$ "$hash(keyword|s)$", where the hash function is a cryptographic hash function that outputs $N = 16$ hexadecimal digits for any input string (term).

Our secure indexes only store the unigrams (keywords) and bigrams found in the documents they represent; any query with an exact phrase consisting of more than two keywords must be converted into a list of bigrams (a *biword* model). Thus, each exact phrase is converted into a list of trapdoors on the bigrams in the phrase, i.e., "$word_1\ word_2\ word_3 \ldots word_k$" $\rightarrow$ "$hash(word_1|word_2|s \in secrets)$", "$hash(word_2|word_3|s \in secrets)$", …, "$hash(word_{k-1}|word_k|s \in secrets)$".

In BNF notation, the syntax[14] for a hidden query is:

<hidden_query>	::=	{"hidden_query": [<hidden_terms>]}
<hidden_terms>	::=	<hidden_term> | <hidden_term>, <hidden_terms>
<hidden_term>	::=	[<trapdoors>]
<trapdoors>	::=	"<trapdoor>" | "<trapdoor>", <trapdoors>
<trapdoor>	::=	(0|1|…|9|a|b|c|d|e|f){16}

Table 5 BNF Hidden Query Grammar

[13] To avoid leaking information about specific user access patterns, the results can return through the same path the query took to get to the CSP.

[14] Note that this BNB grammar generates valid JSON.

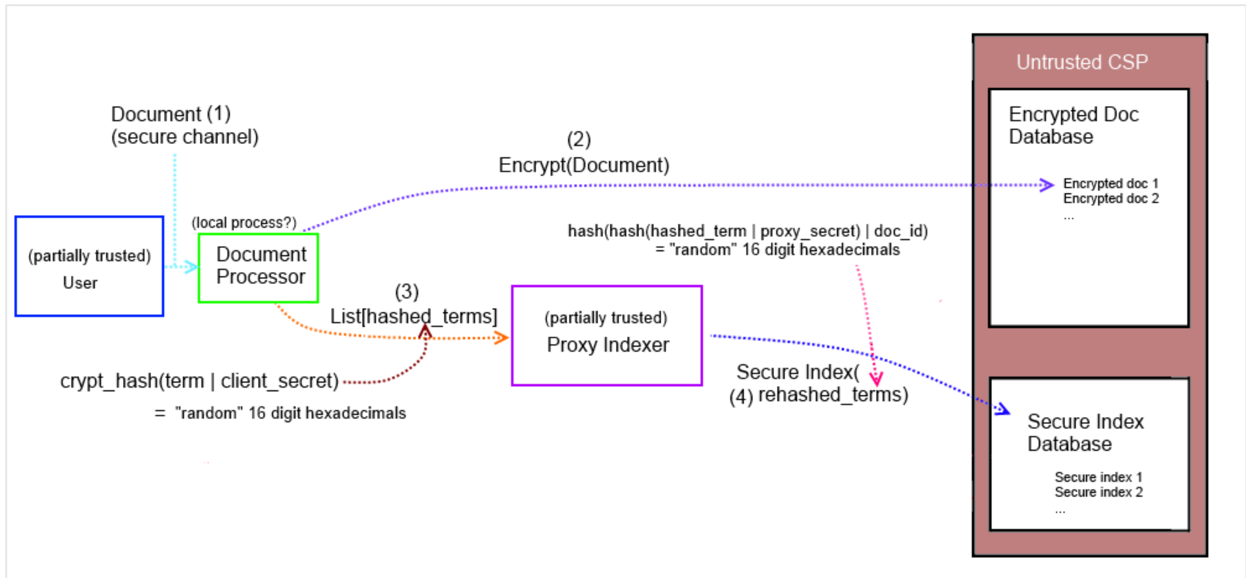


Figure 2 Overview of Secure Index Construction

Secure index construction. What follows is the sequence of actions needed to construct a secure index. Once constructed, the secure index may be stored on untrusted systems, such as a CSP. Refer to Figure 2 for a visualization of the steps.

(1) A client transmits a confidential plaintext document over a secure channel to a $document\ processor$. This is most likely a process running on the client's local machine, but in theory, it can reside anywhere. There are compelling reasons to decouple the $document\ processor$ from the client, e.g., centralized control to enforce uniformity of secure index construction.

(2) *Optional:* The *document processor* encrypts the document using whatever cryptographic algorithm is deemed appropriate and transmits it to an untrusted channel, e.g., the CSP or a system decoupled from the CSP. For *Encrypted Search* purposes, only a reference to the document needs to be provided as output.

(3) First, the *document processor* generates *searchable terms* for the document. These terms can be anything—Soundex hashes, trigrams, etc.—but in our implementation they are lower-case unigrams and bigrams (optionally stemmed) contained in the document (*biword* model). Second, the concatenations are fed into a cryptographic one-way hash function. Finally, it generates a list of hidden terms from the output of the one-way hash function and transmits the intermediate results to a *proxy indexer*.

More precisely, each unigram or bigram s_i in the target document D is concatenated with n_1 secrets—$s'_{i,j} = s_i + secret_j, i = 1\ to\ |unigrams \in D \cup bigrams \in D|, j = 1\ to\ n_1$. Every unigram and bigram in the document will thus be

searchable with n_1 different secrets by an authorized user. Then, each $s'_{i,j}$ is cryptographically hashed—$s''_{i,j} = cryptographic_hash(s'_{i,j})$[15]

(4) First, the *proxy indexer* concatenates one or more *secrets* to the intermediate hidden terms and feeds them into a one-way hash function. Second, the proxy concatenates the document's id (reference) to the outputs of the previous hash function and feeds them into another one-way hash. Third, a *secure index* is constructed from the hash function's output. Finally, it transmits the *secure index* to the CSP. At this point, the CSP stores the secure index and, optionally, its corresponding encrypted document in a database to facilitate efficient *Encrypted Search* operations in response to hidden queries.

More precisely, each $s''_{i,j}$ is concatenated with n_2 secrets—$s'''_{i,j,k} = s''_{i,j} + secret_k, j = 1\ to\ n_1, k = 1\ to\ n_2$. Every unigram and bigram in the document will thus be searchable with $n_1 n_2$ different combination of secrets. Then, each $s'''_{i,j,k}$ is concatenated with the document's identifier (e.g., hash of its filename) and is then re-hashed—$s''''_{i,j,k} = hash^{16}(s'''_{i,j,k} + doc_id) = \text{trapdoor for } s_i$ in the $secure\ index$ representing doc_id. Note that the final rehashing step is performed to prevent the same cryptographically hashed term in different documents from mapping to the same hash value[17]. Finally, the trapdoors $s''''_{i,j,k}$ are fed into the proxy indexer's secure index constructor to build a secure index and the result is sent to the CSP.

User revocation. In [37], the authors observe that most *Encrypted Search* implementations assume only one user will perform searches; or, if multiple users, then they share the same secret, and that secret by itself will allow them to query the secure indexes. However, it may be desirable to be able to revoke a user's ability to query a secure index. For example:

- $user_1$ makes a secure index for a document using $secret$
- $user_2$ is trusted to query secure index by providing her with $secret$
- $user_1$ loses trust in $user_2$ and wishes to revoke her ability to query the secure index, even when the raw contents of the secure index is otherwise accessible to $user_2$.

The motivation for partitioning *secure index* construction into two separate stages in Figure 1 ($query\ processor$ and $proxy\ query\ processor$) and in Figure 2 (a $document\ processor$ and $proxy\ indexer$) is to enable user revocation. Assuming the user and proxies do not collude[18], neither the proxies nor revoked users will be able to query secure indexes, even if they downloaded copies of the secure indexes to their local machines.

---

[15] Default implementation uses SHA256 and non-invertibly maps the hashes to $N = 16$ hexadecimal digits.
[16] A non-cryptographic hash function is preferable for efficiency reasons.
[17] Limits statistical inference to sampling from a single secure index rather than an entire corpus of secure indexes since each secure index has a unique and random way of mapping its unigrams and bigrams to hashes.
[18] A directed acyclic graph (e.g., a chain) of proxies may be used to mitigate the risk of collusion, but this introduces significant overhead.

In our architecture, this can be implemented through partial secrets. The following steps are required for secure index construction:

- A fully trusted $user_t$ provides a partially trusted $user_p$ with set of secrets S_u.
- $user_t$ provides partially trusted $proxy\ indexer$ with another set of secrets S_v.
- $user_p$ constructs a sequence of cryptographic hashes from the ordered unigrams in the document concatenated with secrets in S_u—$hash(w|s), w \in doc, s \in S_u$—and transmits the ordered sequence[19] to $proxy\ indexer$.
- $proxy\ indexer$ constructs a *secure index* from the sequence of cryptographic hashes (intermediate trapdoors) using secrets in S_v—$hash(h|s), h \in hashes, s \in S_u$.

Likewise, the following steps are required for submitting queries to the secure index:

- $user_t$ provides a partially trusted $user_p$ with set of secrets S_u.
- $user_t$ provides partially trusted $proxy\ query\ processor$ with S_v.
- $user_p$ submits hidden queries using any secret in S_u to $proxy\ query\ processor$
- $proxy\ query\ processor$ transforms the intermediate hidden queries using any secret in S_v and transmits the resulting hidden query to the CSP.

To revoke *Encrypted Search* authorization for $user_p$, instruct the $proxy\ query\ processor$ not to honor her queries. Even if $user_p$ had downloaded a local copy of a secure index, she cannot meaningfully query it since she does not know any secrets in S_v.

Additional notes. The prospect of constructing a secure index for the entire collection of documents sounds tempting. This *master index* would allow the CSP to avoid the cost of independently querying each secure index in the database. However, there is a significant problem with this idea: the same term in different documents will look the same. An alternative to the master index is a cache.

The *document processor* and *proxy indexer* may reside on the same machine; indeed, if the client is trusted both of them may reside on the client's local machine.

Moreover, the confidential documents and the secure indexes need not reside on the same server. A secure index and its corresponding confidential document are independent. Also, note that multiple secure indexes may be constructed for the same confidential document to provide tiered access, e.g., the lowest tier may consist of very approximate information (e.g., large location uncertainty) and only unigram terms (to facilitate only keyword queries with coarse term proximity), whereas the highest level may include exact phrase matching and wildcard queries.

[19] Note that the ordered sequence of trapdoors (cryptographic hashes) transmitted to $proxy\ indexer$ leaks significantly more information than the secure index representation since it is a simpler substitution cipher. Thus, $proxy\ indexer$ must be reasonably trusted. Since $proxy\ indexer$ will not be used nearly as frequently as $proxy\ query\ processor$, which can be less trustworthy, $proxy\ indexer$ may be more tightly controlled without as much cost.

Secure index types. There are several different types of secure indexes analyzed and explored in our research. As far as we are aware, with the exception of BSIB (Bloom filter secure index-block) none of these secure indexes have been previously investigated.

Perfect Hash Filter. In place of the Bloom filter [16], we propose the Perfect Hash Filter as the underlying data structure for our newly proposed secure indexes (note that the Perfect Hash Filter is not itself a secure index). Like the Bloom filter, it represents a probabilistic set.

Suppose we wish to represent a set A. First, a perfect hash function is generated to map the n members of set A to n unique integers. Second, the perfect hash of each member x is used to index into an array U and the corresponding element in U[20] is set to a M bit hash of x. More precisely:

(1) Each member $x \in A$ is uniquely hashed by a perfect hash function such that $perfect_hash(x) \neq perfect_hash(y) \leftrightarrow x \neq y, x \in A, y \in A$. That is, no collisions among any of the members of set A are possible. If using a minimal perfect hash, then $\forall_x perfect_hash(x) \in [1, n]$.
(2) Let U be a bit array with a minimum maximum index of $\underset{x}{\operatorname{argmax}}\, perfect_hash(x)$, where each element of U is allocated M contiguous bits. Then, $\forall_x U[perfect_hash(x)] = hash$[21]$(x) \bmod M$.

Refer to Figure 3 for a visual depiction of the Perfect Hash Filter. Note that because each $x \in A$ is represented by M bits, false positives are possible, i.e., $P[hash(x) \bmod M = hash(y) \bmod M \mid perfect_hash(\text{x}) = perfect_hash(y), x \in A, y \notin A] = \frac{1}{2^M}$. That is, a Perfect Hash Filter is a compact probabilistic set in which false positives occur with conditional probability of $P[positive \mid not\ a\ member] = \frac{1}{2^M}$.

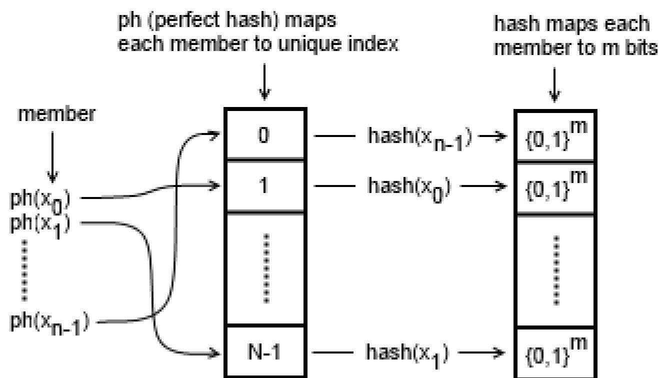


Figure 3 Perfect Hash Filter (Using A Minimal Perfect Hash)

---

[20] Array U is a bit vector of size nm bits such that elements can be stored with arbitrary bit alignment (i.e., byte-alignment is not necessary). Arbitrary bit alignment is generally true for every other data structure in the secure indexes subsequently described, e.g., the array of bit vectors representing blocks in PSIB is actually a single bit vector such that the blocks allocated to a term can use arbitrary bit alignment. This avoids unnecessary padding overhead needed for byte alignment.

[21] A simple $JenkinsHash$ was chosen, but any hash function that uniformly distributes over $[0, M-1]$ is suitable.

Space complexity. Like the Bloom filter, the Perfect Hash Filter can trade accuracy (false positive rate) for space complexity. Theoretically, the Bloom filter requires $O\left(1.44n\log_2\frac{1}{\varepsilon}\right)$ bits, where ε represents the false positive rate. Since there are $k = \left\lceil \log_2\frac{1}{\varepsilon} \right\rceil$ hash functions (assuming optimality), where each hash function may be compactly represented with r bits (e.g., 32 bits[22]), then the total (memory-resident, uncompressed) space required for a Bloom filter with a cardinality of n is $O\left[(1.44n + r)\log_2\frac{1}{\varepsilon}\right]$ bits. For instance, if $\varepsilon = 0.001$ then $9.97r + 14.35n + O(1)$ bits will be needed, where $O(1)$ denotes the constant overhead. If we suppose the number of hash functions is sufficiently small, we can simply the space complexity to $14.35n + O(1)$ bits.

The space complexity for the Perfect Hash Filter is $O\left(sn + n\log_2\frac{1}{\varepsilon}\right)$ bits, where s depends on the load factor[23] of the perfect hash. If it is a minimal perfect hash, the theoretical lower limit for s is 1.44, but in practice this has not been achieved. We use a state-of-the-art algorithm, CHD (Compress, Hash and Displace [38]) in which $s = 2.07$ (bits/key) when using minimal perfect hash. Thus, the Perfect Hash Filter, using the minimal perfect hash and $\varepsilon = 0.001$ requires only $12.0258n + O(1)$. As the false positive rate ε increases, the Perfect Hash Filter will pull even further ahead of the Bloom filter.

If a perfect hash—rather than a minimal perfect hash—is preferable (e.g., see poisoning on page 23), then even fewer bits per key are possible. For instance, if the load factor $r = 0.5$, then $s = 0.67$. However, since we are mapping each hash to an array index, there is a trade-off to consider. While fewer bits per key (member) are required for the perfect hash itself, a load factor $r = 0.5$ means instead of indexing into an array U of size n, we must index (approximately) into an array U of size $n^* = 2n$. Thus, the space complexity is $0.67n - 2n\log_2\varepsilon$ bits. The ratio of the Perfect Hash Filter with a load factor $r = 0.5$ to one with a load factor $r = 1$ is $\frac{0.67 - 2\log_2\varepsilon}{2 - \log_2\varepsilon}$. The limit of this ratio as ε goes to 0 is 2. Thus, a load factor $r = 0.5$, in the limit, takes up twice as much space as a load factor $r = 1$. This inverse relationship is true generally, e.g., a load factor r would require, in the limit, $\frac{1}{r}$ as much space as a load factor $r = 1$.

[22] One can simply hard code a single hash function and salt the keys, e.g., 32-bit integers.

[23] Load factor $r = \frac{cardinality\ of\ set}{maximum\ perfect\ hash\ index}$; a minimal perfect hash has a load factor of $r = 1$.

Computational efficiency. Arguably, more importantly than space efficiency is computational efficiency. The Perfect Hash Filter only requires computing $O(1)$ hash functions while the Bloom filter requires computing $k = \left\lceil \log_2 \frac{1}{\varepsilon} \right\rceil$ hash functions. We will see later, in the chapter on experimental results, that this gives the Perfect Hash Filter a significant computational advantage over the Bloom filter.

Perfect Hash Filter's flexibility. The Perfect Hash Filter is a probabilistic set, but since it provides a unique index for each member, this unique index may be efficiently used to tag a member with other kinds of information. For example, it is trivial to extend the Perfect Hash Filter to implement a multiset. We use this flexibility to derive several different secure index types.

Disadvantages. The Perfect Hash Filter has some disadvantages compared to the Bloom filter. First, it may leak more information. In a Perfect Hash Filter, there are no collisions between members, but in a Bloom filter not only may collisions occur, but they may also only *partially* occur. Thus, the same pattern of 1's and 0's in the Bloom filter's bit vector can occur in more ways than the Perfect Hash Filter's patterns of 1's and 0's. However, through techniques like index poisoning, this may be immaterial. Also, like with the Bloom filter, each member in a Perfect Hash Filter collides with an infinite set of non-members with a false positive rate ε. Second, the Perfect Hash Filter is an immutable data structure—no dynamic updates are possible, e.g., no adding or removing members. However, in the context of *Encrypted Search*, this limitation seems minor.

Perfect Hash Filter secure index (PSI). A secure index based on the Perfect Hash Filter capable of answering approximate Boolean queries. Once the trapdoors (cryptographic hashes of searchable terms, see page 5) have been constructed, they may be inserted into the Perfect Hash Filter.

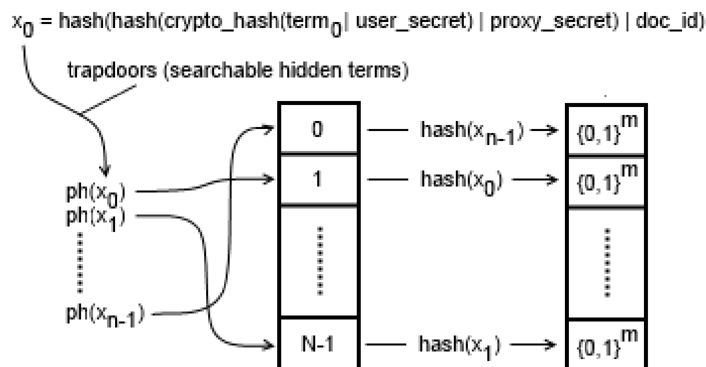


Figure 4 The Perfect Hash Filter Secure Index (PSI)

Space complexity. The space complexity of PSI is just the space complexity of the Perfect Hash Filter, where n—the cardinality of the set—is equal to the number of unique searchable atomic terms (set members) in the document. Thus, it has a space complexity of $O\left(s \cdot n + n \log_2 \frac{1}{\varepsilon}\right)$ bits, where s depends on the load factor r (if using a minimal perfect hash, $s = 2.07$). Note that the searchable atomic terms in our secure indexes are the unique unigrams and bigrams in documents, which allows us to take advantage of the *biword* model for exact phrase searching. Thus, if a document has h words, then there are $n \approx 2h$ searchable atomic terms.

False positives. The Perfect Hash Filter (PHF) is a probabilistic set in which false positives may occur. In addition, for the PSI there is a different way in which a false positive may occur when processing n-gram query terms, $n > 2$. If a document contains the words "A B B D", then the PSI will conceptually represent it as the set {"A", "B", "D", "A B", "B B", "B D"}. To determine if "A B" exists in the document, a single set membership test will suffice.

However, determining whether the query term "A B B" exists in the document is more complicated (it does not exist in the set if only unigrams and bigrams are members). To support exact phrase searches larger than bigrams, as in the trigram "A B B", a *biword* model is used in which n-gram query terms, $n > 2$, are decomposed into a set of $n - 1$ bigram tests, e.g., testing if "A B B" exists is transformed into a a conjunction of membership tests for "A B" and "B B". If all bigrams test as positive, the n-gram term is said to exist in the document. In this case, "A B B" will correctly test positively. But if the term is "A B D", then it will test positively both for "A B" and "B D" but nowhere in the document is the trigram "A B D" found. Therefore, this query term would cause a false positive to occur.

Poisoning. The perfect hash need not be a minimal perfect hash. If a perfect hash with a load factor r is used for a set with a cardinality of size n, then an array of size $\frac{n}{r}$ will be constructed so that only a proportion r of the entries will be addressed by perfect hashes of the n positive members. Instead of leaving the contents of the unaddressed entries zeroed out, the PSI randomizes them to make them appear indistinguishable from member entries.

In general, it is impossible to tell which entries in the array are for members (unigrams and bigrams in the document) as opposed to non-members (fake terms). This is called *secure index poisoning*. Let $p = 1 - r$ denote the proportion of fake elements in the array. Increasing p has no effect on the false positive rate and its space complexity is $f(p) = s(p) \cdot n + \frac{n}{p-1} \log_2 \varepsilon$, where s is a decreasing function of p. For example, if we let $\varepsilon = 2^{-10}$, then $f(0) \approx 12n$ and $f(0.5) \approx 20n$. The latter is only 1.7 times as large as the former despite having an array twice as large.

Additional notes. In our PSI secure index, we allow for any false positive rate $2^{-\mathrm{M}}$, where M is any positive integer. In practical implementations, it may be sensible to optimize the special case where M represents a byte-aligned number of bits, e.g. 16-bits, to take advantage of faster parallel bitwise operations.

PSI is not used in isolation in any of the experiments. Instead, we explore secure indexes derived from PSI—namely, PSIB, PSIF, and PSIP. Likewise, BSIB derives from BSI (Bloom filter secure index), but we do not explore BSI in isolation either. In retrospect, this may have been an oversight. We would also be curious to compare a PSI-derived index that is more directly comparable to BSIB, i.e., a secure index which constructs a perfect hash for each block-of-word segments as is done in BSIB.

PsiBlock (PSIB). PSIB uses a PSI for answering approximate Boolean queries, and on top of that provides an interface capable of answering approximate frequency and location requests for query terms.

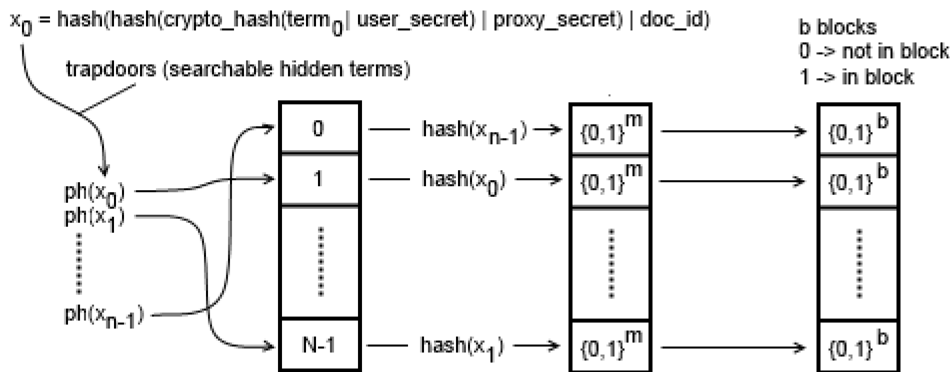


Figure 5 The PSI Block-Based Secure Index (PSIB)

To construct a PSIB, first a PSI is constructed. Then, the document is segmented into b blocks, and a bit vector of size b is constructed for each unigram and bigram in the document such that if a unigram or bigram resides in a block, the corresponding index representing that block in the bit vector for the given term is set to 1. Otherwise, it is set to 0. The larger b is, the more precisely PSIB can locate terms.

Space complexity. The space complexity is $O(sn + n\log_2\frac{1}{\varepsilon} + nb)$ bits, where $s = 2.07$ if a minimal perfect hash is used and n is the number of unique unigrams and bigrams in the document.

The bit vector representing the blocks a term resides in can become very sparse as the number of blocks b increases. However, note that sparse bit vectors are highly compressible—see compressed bit vectors. This represents a trade-off between space complexity and time complexity. We elected not to explore this trade-off in our experiments, but it would be interesting to see the effect of using various compression schemes on the sparse bit vector.

Frequency information. The frequency for a query term is approximated by summing the binary digits of the bit vector representing that term's approximate block locations. Note that for unigram and bigram query terms, the pre-computed bit vector may be used, but if the term is an k-gram, $k > 2$, then the bit vector representing locations for the term is derived from an AND operation on the corresponding bit vector entries for all of the bigrams of the k-gram query term.

Location information. The location for a query term is approximated in much the same way as the frequency, except instead of summing the binary digits of the bit vector, a list of approximate locations is returned. For example, if each block is of size m (each block has m words, except the last which may have fewer) and query term t exists in $blocks$ 0 and $blocks$ 5, then two locations will be returned, one in the range $[0, m-1]$ and the other in the range $[5m, 6m-1]$.

On false negatives. Unlike the PSI, false negatives are possible because we use the approximate location information in PSIB to eliminate positive matches that are most likely false positives. However, if an occurrence of a true positive spans two or more blocks, that occurrence will be eliminated by this imperfect heuristic. A method to eliminate the possibility of false negatives is to

check for whether the bigrams of a k-gram query term exist in adjacent blocks, e.g., for the query term "A B C", if "A B" exists in $block\ i$, check for "B C" in either $block\ i$ or $block\ i+1$. For sufficiently long query terms, a chain of adjacent blocks may also be acceptable.

Additional notes. We allow for an arbitrary number of blocks per document. However, like with the PSI, a more practical implementation could see a significant performance boost if byte-aligned sizes were used instead, e.g., $k-1$ parallel bitwise AND operations could determine which blocks contain all the bigrams in a k-gram query term.

PsiFreq (PSIF). PSIF uses a PSI for answering approximate Boolean queries, and on top of that provides an interface capable of answering approximate frequency requests for query terms.

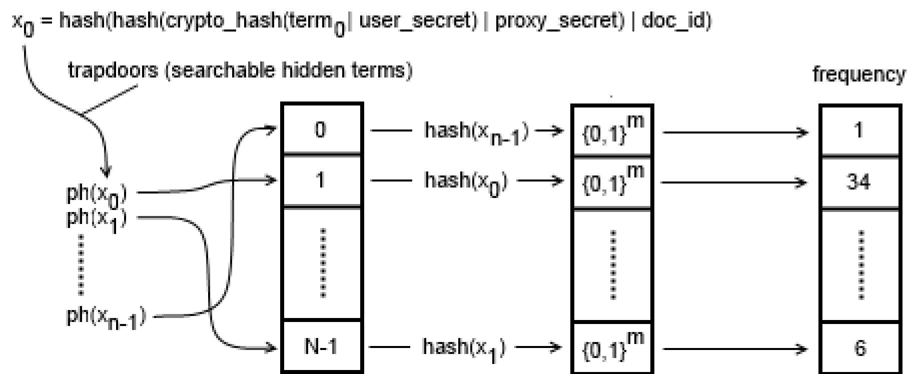


Figure 6 The PSI Frequency Secure Index (PSIF)

To construct a PSIF, first a PSI is constructed. Then, the frequency of each member (unigram and bigram) is calculated. These frequency counts are then stored in a bit vector in a memory efficient way.

Space complexity. The space complexity is $O\left(sn + n\log_2\frac{F}{\varepsilon}\right)$ bits, where s is 2.07 if a minimal perfect hash is used, n is the number of unique unigrams and bigrams in the document, and F is the maximum frequency[24] of any unigram or bigram.

Frequency information. To service frequency request for terms, if the term is a unigram or bigram, the PSI interface is used to index into the PSIF's bit vector that stores frequency information and the indexed value is considered to be the corresponding term's frequency. If the query term is not a unigram or bigram, then the frequency is considered to be the minimum frequency of all of the bigrams making up the query term.

Poisoning. PSIF stores explicit frequency information, unlike PSIB and BSIB, where frequency is approximated implicitly by location uncertainty (block size). If exact frequencies leak too much information about the document, then during the construction phase an approximation of the exact frequency may be stored instead, e.g., $approximate\ frequency = exact\ frequency + UNIF(-r, r)$.

PsiPost (PSIP). PSIP uses a PSI for answering approximate Boolean queries, and on top of that provides an interface capable of answering approximate frequency and location requests.

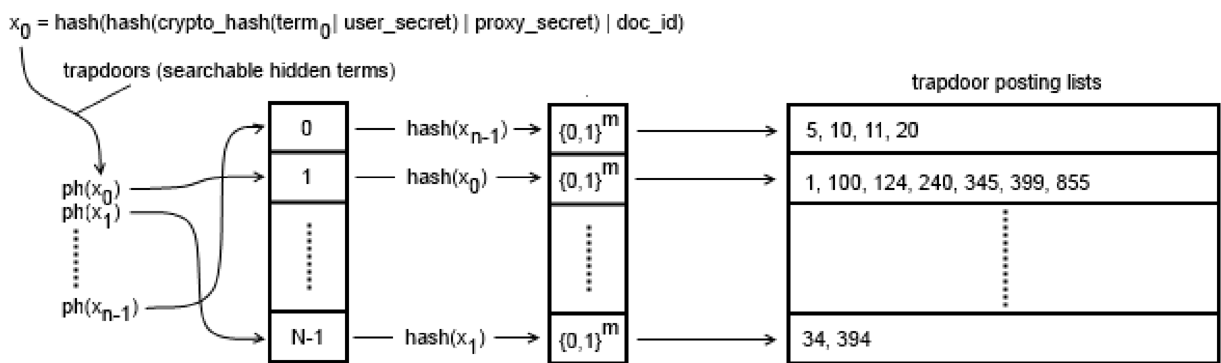


Figure 7 The PSI Postings List Secure Index (PSIP)

To construct a PSIP, first a PSI is constructed. Then, a postings list (a list of positions) for each unigram and bigram in the document is constructed.

Space complexity. The space complexity is $O\left(sn + n\log_2\frac{1}{\varepsilon} + w\sum_{x\in\{\text{unigrams}(doc)\}\cup\{\text{bigrams}(doc)\}}\text{frequency}(x)\right)$ bits, where s is 2.07 if a minimal perfect hash is used, n is the number of unique unigrams and bigrams in the document, and w (e.g., 32 bits) is the number of bits needed to store a reference to a term's location in the document.

A likely more efficient representation of a posting lists is a list consisting integers representing the size of the gaps between adjacent positions of a term. Since the list does not need to facilitate random access, i.e., operations on it can efficiently be performed sequentially, the gaps may also be compressed, e.g., Huffman coded.

[24] Note that a simple optimization would allow F to be maximum frequency minus the minimum frequency. The minimum frequency is at least and most likey is 1, so in practice this may result in very little savings.

The space complexity for a given document with these changes is:

$$sn + n\log_2\frac{1}{\varepsilon} + \left\{\sum_{x\in\{\text{unigrams}(doc)\}\cup\{\text{bigrams}(doc)\}}\left(\sum_{gap\in postings(x)}\text{huffman_code}(gap)\right)\right\} + O(1)$$

Moreover, inspired by PSIB and its block-based approach, instead of storing the exact number of gaps t between adjacent positions of a term, store $\lceil t/k \rceil$, where k is an integer denoting the block granularity size. Furthermore, positions that map to the same block may be ignored to further reduce space requirements (but at the cost of a loss of frequency information, which may be desirable anyway).

Frequency information. Frequency information is stored implicitly by posting lists. To service frequency request for terms, if the term is a unigram or bigram, the PSI interface is used to index into the PSIP's list of postings, and the size of the indexed posting is considered to be the corresponding term's frequency. If the query term is not a unigram or bigram, then the frequency is considered to be the minimum frequency of all of the bigrams making up the query term.

Note that PSIP does not exploit location information to eliminate some false positives as is done in location requests. This was done for the sake of speed, but in theory if the way documents are scored (according to queries) relies upon both location and frequency information, this could be done without any additional cost.

Location information. To service location requests for unigrams or bigrams, the PSI interface is used to index into the list of postings and the posting stored at that index is returned.

For an k-gram query term, a greedy algorithm is used to construct non-overlapping sets each with a diameter less than or equal to some constant that depends on the way in which the postings list was poisoned, as discussed in the next section on poisoning. The positions of a term are taken to be the center of each such non-overlapping set.

Note that since a greedy algorithm is used, this operation is fast as evidenced by experiments consisting of queries with a large phrase terms. Moreover, the use of a more sophisticated algorithm, e.g., an algorithm that produces the maximum number of non-overlapping sets, is not obviously an improvement in the context of greater accuracy.

Poisoning. Location information is explicitly stored for each unigram and bigram in the document. If storing exact locations leaks too much information about the document, then during the construction phase approximate locations may be stored instead. In particular, we achieve this by randomly changing each word's position by some offset, e.g., $approximate\ position = exact\ position + UNIF(-r, r)$. Note that in the experiments, we use a triangular distribution (with a mode equal to the exact position) instead of a uniform distribution. The triangular distribution has less variance and therefore preserves more information about location information.

If exact frequencies leak too much information about the document, then during the construction phase, insert random positions into the postings lists or calculate the average position of adjacent positions for a term and use that average position in place of the adjacent positions. Repeat this as many times as necessary to achieve the desired level of approximation, e.g., if a term appears N times, repeating the mean adjacency operation $N - 1$ times would result in storing only its mean position.

PsiMin (PSIM). PsiMin stands for PSI minimum pairwise distance.

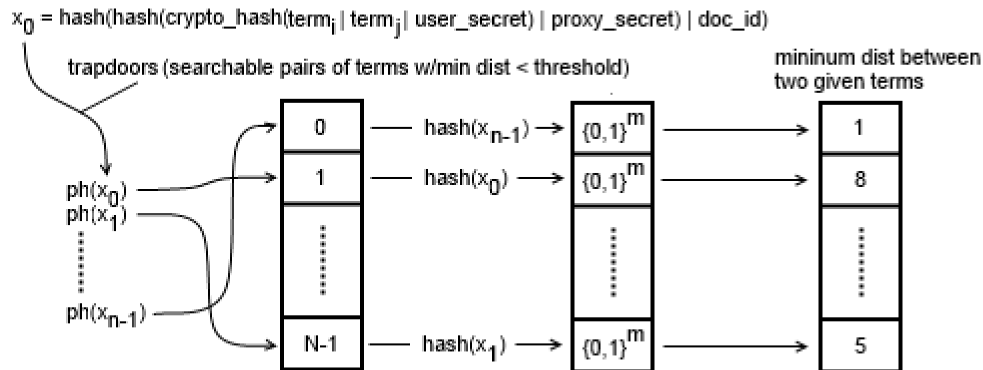


Figure 8 The PSI Minimum-Pairwise Distance Secure Index (PSIM)

To mitigate the threat from the adversary described on page 37, we should not store any location information. However, if proximity-sensitive searching is desired, some kind of proximity information must be stored. What if we only store relative position information (e.g., distances with respect to other terms) such that it is impossible to determine a word's approximate, absolute location? This is the central idea behind PSIM. Instead of storing location information, for each word, store the minimum distance to every other word in the document (as long as the minimum distance is less than some specified threshold distance). To reiterate, it only stores the minimum pairwise distances; all other distance and location information is lost[25]. Order information is also lost, i.e., in isolation it is not possible to determine (with certainty) which word comes first in a pair of words. This is a very problematic representation for the adversary since much of the information to draw inferences from is lost, e.g., the adversary's *jig-saw*-like attack would be ineffective.

Space complexity. Consider a document D consisting of a sequence of the following 10 (5 unique) words:

$$D = \text{"}word_0\ word_1\ word_2\ word_0\ word_3\ word_4\ word_1\text{"}$$

If we let the maximum threshold distance be 3, then PSIM represents document *D* as:

$$\begin{aligned} D^* = \text{psim}(D) = \{&\{word_0, word_0: 3\}, \{word_0, word_1: 1\}, \{word_0, word_2: 1\}, \\ &\{word_0, word_3: 1\}, \{word_1, word_2: 1\}, \{word_1, word_3: 2\}, \{word_1, word_4: 1\}, \\ &\{word_2, word_3: 1\}, \{word_2, word_4: 2\}, \{word_3, word_4: 1\}\} \end{aligned}$$

There are 10 minimum pairwise distances which are less than or equal to 3. In general, the number of minimum pairwise distances stored in D^* is upper-bounded by $\min\{\binom{n+1}{2}, vN\}$, where N and n are the number of words and the number of unique[26] words in D respectively. Thus, the space complexity of D^* is theoretically upper-bounded by $\min\{\binom{n+1}{2}, vN\} \left\lceil s + \log_2 \frac{v}{\varepsilon} \right\rceil + O(1)$ bits, where v is the distance threshold, ε is the false positive rate, and $s = 2.07$ if using a minimum perfect hash for the Perfect Hash Filter.

[25] Even the word count is lost.
[26] The number of unique words $\leq N$, where N is the total number of words in the document.

In general, given the ith and j[th] keywords in D, t_i and t_j respectively, if $\text{mindist}(D, t_i, t_j) \leq v$, we store their minimum distance in D^*:

$$\text{mindist}(\text{D}^*, t, u) = \begin{cases} \text{mindist}(D, t, u), \text{if mindist}(D, t, u) \leq v \\ K, \text{if mindist}(D, t, u) > v \end{cases},$$

where K is some constant.

Location information. Note that we only store the minimum pairwise distances between keywords (unigrams) for document D in $D^* = \text{psim}(D)$. Thus, if the terms whose minimum pairwise distance is being requested are unigrams, PSIM can return an exact answer. However, if the terms are n-grams, $n > 1$, their minimum pairwise distance may only be estimated.

Given k_0-gram term $\text{t}_0 = [w_{0,0}\ w_{0,1} \ldots w_{0,k_0}]$ and k_1-gram term $\text{t}_1 = [w_{1,0}\ w_{1,1} \ldots w_{1,k_1}]$, where $w_{i,j}$ is the jth word in t_i, then the minimum pairwise distance between the ith word in t_0 and the jth word in t_1 is estimated to be $d^*_{i,j} = \text{mindist}(D^*, w_{0,i}, w_{1,j})$. Then, a lower bound for $\text{mindist}(D, \text{t}_0, \text{t}_1)$ is:

$$\text{mindist}(D^*, \text{t}_0, \text{t}_1) = \max_{i,j} d_{i,j}, i \in [0, k_0], j \in [0, k_1]$$

This lower bound may also serve as a reasonable estimate of $\text{mindist}(D, \text{t}_0, \text{t}_1)$, recalling that maximum threshold distance v is expected to be small.

Poisoning. PSIM stores explicit minimum pairwise distance information, unlike PSIB, PSIP, and BSIB, where minimum pairwise distances are derived from (approximate) location information. If exact minimum pairwise distance information leaks too much information about the document, then during the construction phase an approximation of the exact minimum pairwise distance may be stored instead, e.g., $approximate\ minimum\ pairwise\ distance\ = exact\ minimum\ pairwise\ distance\ +\ UNIF(-r, r)$.

Furthermore, like every other secure index derived from PSI, fake terms—or in this case, fake minimum pairwise distances—may also be constructed at a small cost to compression ratio and no cost to accuracy.

Use cases. For large v, PSIM may not be very practical since it causes the size to be nearer to the upper bound and likewise for large documents[27], i.e., its size is exactly $\lim_{v \to N} \text{bits}(D^*) = \frac{n(n+1)}{2}\left\lceil s + \log_2 \frac{N}{\varepsilon}\right\rceil + O(1)$.

However, it may be usable in a more modest way, since experiments have revealed that in practice, for small $4 \leq v \leq 5$, the compression ratio for PSIM can be expected to be less than 1. Analytically, it should be around $vN\left\lceil s + \log_2 \frac{v}{\varepsilon}\right\rceil$. This may still be unacceptably large, but if confidentiality and accurate proximity-sensitive searching are priorities, PSIM may be a tempting.

First, the PSIM may be used to give a secure index S, like PSIB, more precise proximity sensitivity for nearby words without compromising document confidentiality, e.g., $v \in [4,6]$. If the minimum distance between two terms in a query is larger than the threshold distance v, then the approximate location information in the secure index S may be used instead (knowing that it cannot

---

[27] We anticipate that the worst-case space complexity will improbable even for large documents if v is reasonably small on natural language text since sentences are not random sequences of words.

be less than lower-bound v). Such an S that uses PSIM in this way (for small threshold distance v) has a space complexity upper-bounded by $vN\left\lceil s + \log_2 \frac{v}{\varepsilon}\right\rceil + \text{space complexity}(S)$.

Second, PSIM may be used to prevent some false positives from occurring due to the *biword* model. For instance, given a PSIM D^* with a maximum distance threshold v, if the user searches for an r-gram phrase, then for the secure index to contain that phrase, it must be the case that $\text{mindist}\left(D^*, word_i, word_j\right) \leq |i-j|, 0 \leq i,j < r, i \neq j, |i-j| \leq v$. For even small v, this would make false positives extremely unlikely, especially on r-grams where $r \leq v$.

Bloom filter secure index-block (BSIB) [19]. A secure index capable of answering approximate frequency and location requests for query terms. BSIB uses a Bloom filter as the underlying data structure (it is a popular representation of probabilistic sets). There is existing research on this particular secure index representation, so we implemented a design consistent with prior work [19].

Similar to PSIB, BISB is a block-based secure index and operates in a similar way—by segmenting a document into N blocks. However, unlike in PSIB, it constructs a Bloom filter for each block (the Perfect Hash Filter can be used more flexibly since it uniquely maps each member to an integer—see page 22).

Moreover, the Bloom filter requires computing $k = \left\lceil \log_2 \frac{1}{\varepsilon} \right\rceil$ hash functions. Since there are N such Bloom filters, in the worst-case scenario a BSIB must compute $\left\lceil \log_2 \frac{1}{\varepsilon} \right\rceil N$ hash functions. In comparison, the PSI-based secure indexes only need to compute $O(1)$ hash functions.

Relevancy Metrics

Encrypted Search efforts have largely ignored the problem of matching queries to relevant (according to standard information retrieval techniques) sets of documents, e.g., rank-order documents by a measure of how close they match a hidden query without compromising the data confidentiality and query privacy.

In information retrieval, finding effective ways to measure relevancy is of fundamental importance. To that end, past researchers have devised many clever algorithms and heuristics to rank-order documents by their estimated relevancy to a given query. We will explore term weighting and term proximity weighting heuristics in the context of our secure index constructions.

Precision and recall. Precision and recall are relevant metrics for Boolean searches; they do not rank retrieved documents like BM25 or MinDist*; a document is either considered relevant (contains all of the terms in a query) or non-relevant.

Precision measures the proportion of retrieved documents that are relevant to the query. It is defined as:

$$\text{precision} = \frac{|\text{relevant} \cap \text{retrieved}|}{|\text{retrieved}|}$$

Precision has a range of [0, 1]. Recall measures the proportion of relevant documents that were retrieved. It is defined as:

$$\text{recall} = \frac{|\text{relevant} \cap \text{retrieved}|}{|\text{relevant}|}$$

Recall also has a range in $[0, 1]$. It is trivial to achieve a recall of 1 (100%) by retrieving every document in the corpus. However, this comes at the cost of decreased precision. Thus, in general there is a trade-off between precision and recall.

Term importance: BM25 [34]. BM25, a well-established tf-idf variant, is one of the relevancy measures chosen for our experiments. Mathematically:

$$\text{BM25}(d, t, D) = \text{idf}(t, D) \times \frac{\text{tf}(t, d) \times (k_1 + 1)}{\text{tf}(t, d) + k_1 \times \left(1 - b + b \times \frac{|d|}{avgdl}\right)},$$

where $avgdl$ is the average size (in words) of documents in D.

Parameters b and k_1 are free parameters. In the experiments, they are set to typical values [34]; b is set to 0.75 and k_1 is set to 1.2. Ideally these parameters would be automatically tuned for each secure index. The function tf stands for *term frequency* and simply returns the number of occurrences of term t in document d. The function idf stands for *inverse document frequency.*

$$\text{idf}(t, D) = \log\left(\frac{|D| - \text{count}(t, D) + \frac{1}{2}}{\text{count}(t, D) + \frac{1}{2}}\right),$$

where $|D|$ is the number of documents in D and $count$ is a function which returns the number of documents in D which had one or more occurrences of term t.

When using BM25 to rank search results, each document d in D is ranked according to query Q by the function $\text{BM25Score}(d, Q) = \sum_{t \in Q} \text{BM25}(d, t, D)$, where t is a term in query Q. After giving every document a BM25 rank, the results are sorted in descending order of rank as the final output.

Note that BM25Score is not used in most real-world information retrieval systems; instead, they typically employ a vector space model, in which a vector $\vec{v}(d)$ for each document d in D is constructed, where each dimension represents a word t with a weight equal to $\text{BM25}(d, t, D)$. Subsequently, a document can be ranked according to a query by taking the *cosine similarity* [39] of their respective vectors. However, as explained on page 12, this representation is problematic in light of the limited information (for confidentiality) available in secure indexes.

Term proximity: MinDist [40].* MinDist, like BM25, ranks documents according to their proximity relevance to a given query. It is a less established ranking heuristic than BM25, but in experiments [40] it had performed well compared to other proximity heuristics. In our experiments, we add additional tunable parameters to MinDist and call it MinDist*.

MinDist* is a proximity heuristic in which the minimum distance between each existent pair of terms are summed over. Thus, it needs location information. So, for example, if query $Q = \{A, B, C\}$, where A, B, and C are the terms of Q, and document D = “A B D D A D C”, then the minimum pairwise distances are: $(A, B) = 1$, $(A, C) = 2$, and $(B, C) = 5$. The summation of these distances is simply $s = 1 + 2 + 5 = 8$. MinDist* is a scoring function dependent upon the minimum pairwise distance summation s.

The intuition behind MinDist* is the more concentrated the query terms are in a document, the more relevant the document is to the query, but only up to a certain point. For example, consider a query Q = {"$computer$", "$science$"}. Given two documents A and B, where A contains both "computer" and "science" on page 7 and B contains "computer" on page 7 and "science" on the page 20, it is obvious that A should be considered much more relevant to Q since the two keywords of interest are much closer together. However, consider a third document C containing "computer" on page 7 and "science" on page 100. Intuitively, this is not much worse than B; both documents are simply not that relevant; B is only marginally more relevant at best.

Mathematically, these intuitions are implemented in the following way. Let Q be the set of query terms, Q' be the subset of Q that exist in the given document, and s be the sum of the minimum pairwise distances between terms in Q'.

$$\text{MinDistScore}(Q) = \ln\left(\alpha + \gamma \cdot \exp\left\{-\frac{\beta s}{|Q'|^{\theta}}\right\}\right),$$

where $\alpha, \gamma, \beta, \theta > 0$[28]. To see if this function matches our expected intuition—a strictly decreasing function that flattens out as s increases—it may be instructive to consider the limits and partial derivative of $MinDistScore$ with respect to s.

As s converges to 0, $MinDistScore$ converges to $\ln(\alpha + \gamma)$. As s converges to ∞, $MinDistScore$ converges to $\ln(\alpha)$. To see if these end points are the maximum and minimum values respectively, let us consider the partial derivative with respect to s.

$$\frac{\partial}{\partial s}\text{MinDistScore}(Q) = -\frac{1}{|Q'|^{\theta}}\left(\frac{\beta\gamma \cdot \exp\left\{-\frac{\beta s}{|Q'|^{\theta}}\right\}}{\alpha + \gamma \cdot \exp\left\{-\frac{\beta s}{|Q'|^{\theta}}\right\}}\right)$$

This function, for all positive values of s, is negative. It approaches $-\frac{1}{|Q'|^{\theta}}\left(\frac{\beta\gamma}{\alpha+\gamma}\right)$ as s approaches 0 and it asymptotically approaches 0 as s approaches ∞. This matches the desired intuition; for small s, a small increase in s corresponds to a large decrease in $MinDistScore$; and, for large s, a small increase in s corresponds to small decrease in s.

It is also reassuring to note that for large $|Q'|$, the function will decrease less rapidly than for small $|Q'|$, which is the desired behavior. Recall that $|Q'|$ corresponds to a document matching more of the terms in query Q. We do not wish to penalize (at least not too harshly) a document that contains more of the query's terms but spread out over a larger region.

MinDist* may be used as a way to add proximity sensitivity to already established scoring methods, like BM25. For example, a linear combination of their scores can be used as the final output of a scoring function that is both sensitive to proximity and term frequencies[29]:

$$\text{Score}(d, Q, D) = \alpha_1 \cdot \text{MinDistScore}(Q) + \alpha_2 \cdot \text{BM25Score}(d, Q, D)$$

---

[28] The original MinDist scoring function is equivalent to $MinDistScore(Q;\ \gamma = 1, \beta = 1, \theta = 0)$.

[29] If $\alpha_1 + \alpha_2 = 1, \alpha_1 > 0, \alpha_2 > 0$ then $MinDistScore$ and $BM25Score$ may be normalized such that they share the same minimum and maximum values and the $Score$ has a range $[0, 1]$.

Mean average precision (MAP). MAP is a popular way to measure the performance of information retrieval systems with respect to degrees of relevancy. We use MAP to measure a secure index's BM25 (or MinDist*) output. In particular, we do this by measuring how closely its BM25 (or MinDist*) output matches the BM25 (or MinDist*) output for a non-secure, canonical index that retains perfect location and frequency information.

The more approximately a secure index represents a document, the less information one can infer about the document from the secure index. Thus, to what extent the secure index can approximate a document while still achieving high MAP scores is an important question.

MAP is calculated by taking the mean of the average precisions on over 30 queries. The precision at k is:

$$\text{precision}(q,k) = \frac{|k \text{ most relevant docs to query } q \cap \text{top } k \text{ retrieved docs for query } q|}{k}$$

The average precision for the top n documents is:

$$\text{avg_precision}(q,n) = \frac{1}{n}\sum_{k=1}^{n} \text{precision}(q,k)$$

The mean average precision (MAP) for the top n documents over a query set Q is:

$$\text{map}(Q,n) = \frac{1}{|Q|}\sum_{i=1}^{|Q|} \text{avg_precision}(q_i,n)$$

Thus, to estimate a secure index type's MAP score, we need a set of documents D and a query set Q. Then, we construct a set of secure indexes SI for D, and rank-order both SI and D according to each $q \in Q$. Finally, we calculate the mean average precision (MAP) over these rank-ordered ouputs using the rank-ordered outputs for D as the $true$, canonical output.

Consider the following. Suppose the ranked list of relevant documents to a query is $[3,0,1,2,4]$, and the retrieved ranked list (by a secure index) is $[2,4,3,0,1]$. The precision at $k=1$ is $\frac{|\{3\}\cap\{2\}|}{1} = \frac{|\emptyset|}{1} = 0$; the precision at k=2 is $\frac{|\{3,0\}\cap\{2,4\}|}{2} = \frac{|\emptyset|}{2} = 0$; the precision at $k=3$ is $\frac{|\{3,0,1\}\cap\{2,4,3\}|}{3} = \frac{|\{3\}|}{3} = \frac{1}{3}$, the precision at $k=4$ is $\frac{|\{3,0,1,2\}\cap\{2,4,3,0\}|}{4} = \frac{|\{3,0,1\}|}{4} = \frac{3}{4}$, and the precision at $k=5$ is $\frac{|\{3,0,1,2,4\}\cap\{2,4,3,0,1\}|}{5} = \frac{|\{3,0,1,2,4\}|}{5} = \frac{5}{5} = 1$. Thus, the average precision is $\frac{0+0+\frac{1}{3}+\frac{3}{4}+1}{5} = \frac{5}{12}$. The mean average precision would simply be the mean of the average precisions for M queries.

Note that the average precision for the last value of k is necessarily 1 if, by that iteration of k, the relevant set and the retrieved set contain the same elements. However, in general, this is not the case; for instance, if the relevant ranked list of documents to a query is (A, B), and the retrieved ranked list is (D, C, B, A), then if the mean average precision goes from $k=1$ to $k=2$, the average precision is 0. In our simulations, we do a variation of this.

Suppose the relevant ranked list of documents to a query is (A=0.9, B=0.85, C=0, D=0), and the retrieved ranked list is (A=0.9, C=0.85, D=0.5, B=0). Then, I calculate the average precision for the top $k=3$ instead of the top $k=4$ or top $k=2$. In this example, document B is not included in any of the precision at $k=1$ to $k=3$ calculations.

Finally, in one of the experiments, we conduct a "page one" MAP test, i.e., we find the mean average precision using only the top 10 results. The randomized algorithm does much more poorly in this instance, e.g., with over 85% probability, the mean average precision will be less than 0.05.

INFORMATION LEAKS

To mitigate document confidentiality and query privacy information leaks, several different techniques will be explored.

Query privacy leaks. Hidden queries, as substitution ciphers, represent one of the more vulnerable parts of the system. We consider below two general strategies an adversary may employ to compromise query privacy: cryptographic hash attacks and maximum likelihood attacks. Note that we only provide a theoretical treatment on cryptographic hash attacks, but provide both a theoretical and empirical (simulation) treatment on adversaries employing the proposed maximum likelihood attack.

Cryptographic hash attacks. Cryptographic hash functions take as input an arbitrary-length string and output a fixed-length string (hash value). In general, cryptographic hash functions have the following properties.

Pre-image resistance. Given a hash h, finding an m such that $hash(m) = h$ should be intractable. Lacking this property, an adversary may observe h and find one or more candidate m. On the one hand, in the context of hidden queries, lacking pre-image resistance, an adversary may be able to discern which keywords or phrases a target has an interest in finding. On the other hand, it may be productive to increase the collision rate so that it is trivial to find an m such that $hash(m) = h$. This is especially relevant when the adversary knows one or more secrets—in that case, simple dictionary attacks become possible. The collision rate can be controlled such that a dictionary attack will produce some set M such that $hash(m) = h, m \in M$. In this case, the adversary may not have enough information to determine which $m \in M$ the user was actually interested in.

Thus, there is a case to be made that pre-image resistance is undesirable in this context. Indeed, since most queries will consist of common terms, if the adversary knows any secrets he can hash a dictionary of common terms to discover what other users are searching for.[30] However, if too many collisions on legitimate queries occur, then this may have a negative effect on the accuracy of search results. Collision resistance therefore represents a trade-off between privacy and accuracy of search results.

We do not explore this trade-off experimentally, but a simple approach to exploring consists of changing the size of the fixed-length output of the cryptographic hashes (trapdoors); if a hash function maps all input to n bits, then a smaller n corresponds to a larger collision rate. For example, if there are 64 query terms which are mapped to 4 bits each, then on average (assuming a good uniform hash function) each term will collide with $\frac{64}{2^4} = 4$ other terms in the population. How this will in practice effect outputs of interest is difficult to estimate without performing experiments.

---

[30] Note that this assumes the adversary has access to the hidden query stream. If the hidden query steam is taking place over a secret channel, the secure index server and the adversary must share information to make this an effective kind of attack.

Collision resistance. Finding strings m_1 and m_2 such that $hash(m_1) = hash(m_2)$. A cryptographic hash function should make this infeasible. In the context of *Encrypted Search*, this does not seem especially relevant to enable any form of attack[31].

Maximum likelihood attack. Instead of mounting an attack that depends on finding cryptographic hash collisions, an adversary with access to hidden query histogram data can mount a maximum likelihood attack.

Suppose there is a sample of n independent and identically distributed observations t – that is, a history of query terms – coming from some distribution f, where f is a probability mass function denoting how probable a randomly sampled query term t is.

$$P[\text{randomly sampled query term is } t] = f(t)$$

Thus, the probability of seeing a particular history of terms $t_1, t_2, \dots, t_n$ is[32]:

$$P[\bar{t}] = P[t_1 \wedge t_2 \wedge \dots \wedge t_n] = f(t_1)f(t_2)\dots f(t_n)$$

Each term t is mapped to a hidden term h. The objective of the adversary is to find a function g which maps each hidden term h to a term t.

$$g(h) = \begin{cases} t_{i_1} \; if \; h = h_1 \\ t_{i_2} \; if \; h = h_2 \\ \quad \cdot \\ \quad \cdot \\ \quad \cdot \\ t_{i_n} \; if \; h = h_n \end{cases}, t_{i_j} \neq t_{i_k} \; if \; j \neq k$$

To accomplish this goal, we simulate an adversary in which distribution f is known (and, in the simulation, is a Zipf distribution); since f may be estimated by examining queries in an information retrieval system that does not use hidden queries, it is plausible the adversary can learn a reasonable approximation of f.

For a given $\tilde{g} \in g$, the probability of seeing a particular history of hidden terms $\bar{h}$ is:

$$P[\bar{h}] = P[h_1 \wedge h_2 \wedge \dots h_n] = \prod_{i=1}^{n} f(\tilde{g}(h_i))$$

To discover the most likely mapping function $\tilde{g} \in g$, the adversary will use maximum likelihood estimation; that is, it will explore the space of g and choose a $\tilde{g}$ which maximizes the probability[33] of seeing $\bar{h}$.

$$\tilde{g} = \underset{g}{\operatorname{argmax}} \prod_{i=1}^{n} f(g(h_i))$$

---

[31] Thus, *Encrypted Search* is not vulnerable to birthday attacks.

[32] Since order is irrelevant (i.i.d. distribution), the actual probability is $\frac{n!}{(k_1!k_2!\dots k_n!)} f(t_1)f(t_2)\dots f(t_n)$, where k_i represents how many times t_i appears in the query history set. However, $\frac{n!}{(k_1!k_2!\dots k_n!)}$ is a constant for a given history of n and k, so we can safely ignore it in our maximum likelihood attack.

[33] When simulating the adversary, the log of the maximum likelihood will be used instead.

Since the space of g is $O(n!)$, a subset of the space must be explored which has a high likelihood of finding local maxima. In the simulations, we use a hill-climbing algorithm, in which the neighbors to a point in this space is operationally defined as the swapping of any two t_{i_j} and t_{i_k} in $\tilde{g}$.

Note that an excellent initial starting point in this space, especially given a sufficient number of samples, is to collect all of the hidden terms, sort them by frequency, and pair them up to the terms $t \in f$ sorted by probability. However, we do not use this initial estimator in the simulation[34].

Multiple secrets and query obfuscations. When we add m secrets per term, i.e., hidden terms for term t consist of the set $\{h(t|secret_1, secret_2, \ldots, secret_m\}$, then g has the form such that each plaintext term maps to m hidden terms. Thus, the space of g is now $O((nm)!)$ instead of $O(n!)$, and it is expected that more samples of hidden terms will be needed for a given level of accuracy (where accuracy is defined as the percentage of hidden terms which have been correctly mapped to plaintext terms).

Furthermore, when we add k obfuscations to the vocabulary of hidden terms (without multiple secrets), g takes the form:

$$g(h) = \begin{cases} t_{i_1} \ if \ h = h_1 \\ t_{i_2} \ if \ h = h_2 \\ \quad . \\ \quad . \\ \quad . \\ t_{i_n} \ if \ h = h_n, t_{i_j} \neq t_{i_k} \ if \ j \neq k \\ o \ if \ h = h_{n+1} \\ \quad . \\ \quad . \\ \quad . \\ o \ if \ h = h_{n+k} \end{cases}$$

In the above function g, h_{n+1} to h_{n+k} do not actually map to any plaintext term; they all map to class *obfuscation*. Thus, if a specific set of k hidden terms map to class *obfuscation*, there is only one way for each of those hidden terms to be mapped to it. Thus, the space for g is $\frac{(n+k)!}{k!} = P(n + k, n)$. This is equal to or larger than $n!$ for all non-negative integer values of n and k; as a degenerate case, when $k = 0$ (no obfuscations), it reduces to $n!$.

Obfuscations introduce additional unknowns for the adversary that either must be given or estimated. As with the distribution of plaintext terms being given, the probability that a random hidden term is an obfuscation term will also be given. That is, $\text{obfuscation rate} = P[obfuscation] = c, 0 < c < 1$.

There are many ways to complicate matters for the adversary when dealing with obfuscations, e.g., making it so that the distribution of individual obfuscation hidden terms are similar to the distribution of non-obfuscated hidden terms. However, in our experiments, each obfuscation term has a uniform probability.

[34] If this is done, using multiple secrets and obfuscations would result in an even greater advantage with respect to mitigating the effectiveness of maximum likelihood attacks.

When combining both obfuscations and secrets, the space of g is $\frac{(nm+k)!}{k!} = P(nm+k, nm)$. In any case, the space of g explodes as m or k grows (the original space was already exponential with respect to n). In our experiments (see chapter 4), we only consider increasing m or k separately, i.e., when increasing k, m is fixed at 1, and when increasing m, k is fixed at 0.

Document confidentiality leaks (reconstructing documents from secure index information). There are many possible ways an adversary could compromise the confidentiality of the secure indexes. First, the adversary needs some way to meaningfully query the secure index; it can do this with the mapping learned in using maximum likelihood estimation, or it may simply have access to one or more secrets (e.g., the adversary can be a legitimate user). Once the adversary has the capability to meaningfully query secure indexes, it may systematically analyze the information contained in them to classify or (partially) reconstruct the confidential documents.

A secure index contains an approximation of a document's word[35] frequencies without revealing which words are in the document. In addition, it may contain location information about each of the words. If the secure index only provides approximate frequency information, then a line of attack may consist of the following steps. First, sample from a word distribution to automatically determine (via queries) some fraction of the unigrams or bigrams in the given document and their respective frequencies. Then, using a bag-of-words model, classify the document, e.g., $P[\text{medical document} \mid \text{bag of words}]$[36]. More specific classes are possible also, e.g., a single plaintext document can serve as a class.

However, more sophisticated attacks exist. For instance, note that bigrams are more informative feature classifiers than unigrams. Moreover, trigrams are more informative than bigrams. Indeed, the larger the n-gram, the more informative it may be as a feature. The limiting case for this is an n-gram the size of an entire plaintext document. Finding a match on this in a secure index would indeed be very informative.

With these insights as motivation, the adversary could use a secure index's bigrams in conjunction with a language model to partially reconstruct a document from the information contained in its secure index. First, the adversary can use a generative language model, like the trigram language model, to probe the secure index for plausible n-grams. For instance, if the adversary finds a positive hit on the bigram "A B", this information can be used to generate plausible trigram phrases, e.g., sample word *x* from the conditional distribution, P[*x* | "A", "B"]. If "C" is plausible given the previous two words were "A" and "B", then check for a hit on the trigram phrase "A B C". If this trigram tests positively in the secure index then generate and test plausible 4-gram phrases by sampling from the distribution, P[x | "B", "C"].

Repeating the above steps, a large set of n-gram phrases (that test positive in the secure index) may be constructed. Furthermore, some of the discovered n-grams may overlap in some way, in which case the adversary can automatically stitch the pieces together in various ways. The plausibility of a stitching can be estimated using the language model, especially if multiple consistent stitchings of the same size are possible.

---

[35] Actually, a secure index using a *biword* model stores the unigrams (words) and bigrams in a document.
[36] On the other hand, this may be a desirable search method. See Topic searching in chapter 0.

Already, this may reveal a significant amount of details about the document. However, if the secure index also provides location information, the adversary has a much easier job. If exact location information is provided for each word, then as the adversary finds words as previously described, it puts them in their proper place (like a jig-saw puzzle). As words are placed, larger and larger n-grams are constructed, and the language model can be used to generate plausible candidates for the missing words. Alternatively, since the problem has been vastly simplified, the adversary may exhaustively check each word in a dictionary.

Since false positives are possible, each position may have multiple candidates. To deal with this eventuality in a reasonably straightforward way, the adversary can find an assignment of candidates that (approximately) maximizes the likelihood (given a language model) of a given assignment of candidates to each position.

Assuming this form of attack is reasonably successful[37], the reported positions for a word should have some degree of uncertainty—e.g., only reporting that a word falls within some range (block), as PSIB and BSIB do, or scrambling the positions in some random way, as PSIP does.

Problems with the block-based approach. The block-based approach used in PSIB and BSIB reduces the problem to treating each block as a small document, and solving each one independently without location information using the techniques described on page 37. Since the document is much smaller, the reconstruction effort may be significantly easier than trying to do this for the entire document.

To paint a clearer picture, if the document consists of N words, and the words are segmented into k blocks, then there are $n = N/k$ words per block. If all n words in the block are discovered (and ignoring word multiplicities), then there are $n! = \left\lceil \frac{N}{k} \right\rceil!$ ways to order a block (and globally there are $(n!)^k$ ways to arrange the words in the entire document given the constraint information provided by the blocks). Thus, for each permutation, the adversary calculates its likelihood given the chosen language model (e.g., trigram language model), and saves the permutations with the highest likelihoods. Since $n!$ and $(n!)^k$ represent vastly smaller spaces than $N!$, the adversary should find the reduced problem significantly easier. For instance, suppose $N = 500$ and $k = 10$, then $n! = \left(\frac{500}{10}\right)! = 50!$, and $(n!)^k = (50!)^{10}$, as opposed to $N! = 500!$, which is a factor of 1.8×10^{489} times the size of the reduced space. Moreover, it is a factor of 4.0×10^{1069} times the size of each independent block.

[37] Unlike where we simulate an adversary performing an MLE attack on the query steam, we only provide a theoretical analysis of attacks exploiting the approximate information in the secure indexes.

An alternative solution. A significant problem with the block-based approach is the adversary's ability to treat each block as a separate, independent problem—this has the effect of reducing the adversary's curse of dimensionality. PSIP is designed, in part[38], to overcome this problem. In PSIP, there is no such block delineation—instead, words are offset from their true position according to some random variate. This makes it harder to treat the document as a set of smaller independent sub-problems.

For instance, suppose document D = "A B C D E F G H". To simplify matters for D'—the secure index approximation of D—suppose we can swap any word in D with any other word in D as long as the words final position is within two units of its starting position. Then, let scrambled document D' = "B A E D C G F H". Is it possible to break this larger problem down into two smaller independent problems?

d_1' = "B A E D" and d_2' = "C G F H" will not work since, in the original document, the first set should contain elements from {A, B, C, D}, but it is missing B and has an additional E. Any 4-gram ordering on these two sub-problems cannot match the ordering in the original document; indeed, in this case, the only sub-ordering that matches the original ordering is "A B" and "F G H". These two sets cannot be stitched together since they have no overlapping components.

Another possible division is d_1' = "B A", d_2' = "E D C", and d_3' = "G F H". This is a legitimate way to reduce the larger problem into a set of smaller independent problems, but the adversary has no way of knowing this beforehand. For instance, if instead A had been swapped with C, this would no longer be a legitimate partition.

This will blow up the search space for the adversary. The adversary may still use the location information to do things like eliminate impossible stitchings, but it is more difficult to use the location information to create independent sub-problems. Of course, it may be acceptable to reduce the original document into sub-problems with a size dependent upon the location uncertainty and settle for more approximate solutions. It is also possible to parameterize the segmentation points and include those as additional parameters to optimize, but this has the effect of blowing up the search space even more.

On the effect of false positives. As demonstrated in the previous section, false positives create a problem for the adversary attempting to (partially) reconstruct a document from the information in its secure index.

Given a secure index of with N words, each unique (in order to simplify the discussion), there are $N!$ permutations. The adversary wishes to find some N words (which will test as positive in the index) and then find a permutation that maximizes the likelihood of observing that sequence of words given a chosen language model.

An exact solution is already computationally intractable—$O(N!)$. Adding false positives complicates matters even more for the adversary, although it is still in $O(N!)$. Suppose false positives occur at a rate of $0 < \varepsilon < 1$, and the adversary wishes to perform an exhaustive search on the secure index by iterating through a dictionary consisting of $N + k$ words, where the N words in the document re a subset of the $N + k$ words in the dictionary. Then, to find the N words in the document, N of those

[38] It is also designed to provide more accurate location information by allowing the mean error of approximate word positions to be 0—i.e., PSIP changes each word's position with respect to its true mean.

words from the dictionary will necessarily be true positives and it is expected that there will be $\|\varepsilon \cdot k\|$ false positives.

In total, it is expected that $N + \varepsilon \cdot k$ words will positively match. Each one of these words is a candidate, and thus instead of the adversary needing to explore a space consisting of $N!$ possibilities, the adversary must explore a space of $\frac{(N + \|\varepsilon \cdot k\|)!}{\|\varepsilon \cdot k\|!} = P(N + \|\varepsilon \cdot k\|, N)$ possibilities. The degenerate case $k = 0$ evaluates to $N!$, but as k grows it quickly diverges from $N!$ for a given ε.

Thus, we see that on the one hand, a high false positive rate mitigates reconstruction attacks. On the other hand, as some of the experiments were designed to probe, a high false positive rate may cause the searching apparatus to return unacceptably poor results if it is unduly affected by the false positive hits. This represents a trade-off—the least disclosing ε is 1 and the most accurate ε is 0.

Secure index poisoning. The intent of poisoning a secure index is to mitigate frequency analysis attacks and *jig-saw*-like (using location information) attacks. Namely, we wish to cause the hypothetical adversary described previously to be less successful at reconstructing a document from the information in its secure index. This can be done in a few different ways.

Fake terms. We will insert fake terms (unigrams and bigrams) into a secure index. Theoretically, with respect to its mitigating effect on the threat posed by the adversary, this is similar to increasing the false positive rate. However, unlike increasing the false positive rate, this can be done in a way that should theoretically not affect search accuracy (as the experiments has corroborated).

Approximate frequency information. Knowing precise frequency information is very informative for the adversary, as previously described. PSIB and BSIB naturally—as a byproduct of location uncertainty—provide approximate frequency information. However, PSIP and PSIF must be explicitly instructed to approximate frequencies.

This constitutes an advantage for PSIP and PSIF, especially since it may be precisely controlled, e.g., give each trapdoor's frequency a particular range of uncertainty.

Approximate location information. Knowing precise location information is very informative for the adversary, as previously described. This is controlled by location uncertainty. Note that PSIF does not store location information, so this parameter is not applicable to it.

CHAPTER IV
EXPERIMENTS

Our experiments are intended to explore how one or more inputs relate to one or more outputs. To keep things simple, our experiment designs consist of changing one input (while the other inputs are held constant) and observing how one or more outputs respond with respect to the change in the given input.

INPUTS

- *Secure index*. The type of secure index. It is either PSIB, PSIF, PSIP, or BSIB. In most of the experiments, multiple secure indexes and their respective outputs are compared to one another.

- *Documents (documents/corpus)*. Number of documents in the corpus. A variable corpus size should effect most outputs in a linear way, e.g., MinDist* lag time should depend linearly on the number of documents (assuming documents are of fixed size). However, MinDist* scoring and BM25 scoring may be effected in a non-linear way, thus we make this variable to see how such outputs respond.

- *Pages*. The number of pages in each document in the corpus. A variable page count will be used to see how each *secure index* scales with document size with respect to a number of parameters.

- *Terms/query*. The number of terms in a query, where a term is either a keyword or an exact phrase.

- *Words/term*. The number of words in each term.

- *Secrets*. Number of secrets that can be used to search for query terms in the secure index database.
- *Obfuscations*. This input is used in two different senses. In the context of the attack simulation, this input refers to the number of unique obfuscations; otherwise, it refers to the number of obfuscated terms added to a query.

- *Obfuscation rate*. In history attack simulations, obfuscation rate refers to the probability that a random term in the history set will be an obfuscated term. This is the parameter that, in practice, will also be used by the client's hidden query constructor, i.e., add obfuscations to hidden queries such that for large hidden query histories:

$$\text{obfuscation rate} = \frac{\sum_{q \in history} count(obfuscation\ terms \in q)}{\sum_{q \in history} count(terms \in q)}$$

- *Location uncertainty*. Unigram or bigrams in the document have exact positions. Exact positions reveal too much information about the contents of the document; thus, positions should only be known approximately. Location uncertainty refers to range of uncertainty (in word positions)

of a term's true location (see page 37).

- *False positive rate*. A word not in a secure index document will have a probability (the false positive rate) of testing positively as belonging to it. This probability can be controlled by increasing or decreasing the input for false positive rate. On the one hand, a low false positive rate should improve search accuracy; on the other hand, a high false positive rate should improve confidentiality (e.g., more difficult for an adversary to reconstruct the document).

- *Junk percentage*. This is the percentage of fake terms in a secure index, as described on page 40.

- *Relative frequency error*. PSIB and BSIB implicitly approximate frequencies through their location uncertainty (block size). However, PSIP and PSIF can explicitly control this parameter.

- *Vocabulary size*. The number of unique keyword search terms. In our adversary experiments, this is made to be relatively low number (~ 50) so that we can simulate an adversary implementing the maximum likelihood attack described on page 35.

- *Query history size*. In our adversary simulations, we simulate an adversary employing a maximum likelihood attack. The more data (history of query terms) the adversary has access to, the more the data begins to look like the true distribution and thus the more accurate the maximum likelihood estimation becomes.

- *Absolute location error*. A secure index should only provide approximate location information, thus $absolute\ location\ error = \|approximate\ location - actual\ location\|$.

Outputs

- *Secure index size.* The size (e.g., bytes) of the secure index database for a corresponding corpus (collection of documents).

- *Build time.* Time taken to build the secure index database for a given corpus.

- *Load time*. Time taken to load a secure index database for a given corpus.

- *Boolean search precision.* Proportion of retrieved documents relevant to the Boolean search (all of the terms must be in a relevant document).

- *Boolean search recall*. Proportion of relevant documents retrieved.

- *BM25/MinDist* rank-ordered MAP.*

- *BM25/MinDist*/Boolean search lag time.* Time taken for the corresponding kind of query to complete.

- *Compression ratio*. The ratio of secure index size to the size of the actual document the secure index is representing. Smaller values are preferable.

- *Accuracy*. This output is for the adversary simulations. It refers to proportion of hidden terms that the adversary is able to learn to accurately map to plaintext terms.

PLATFORMS

Table 6 Testbed System for Experiments

MACHINE A	
OPERATING SYSTEM	Windows 7 Service Pack 1
PROCESSOR	AMD A6-6400K APU 3.9GHz
INSTALLED MEMORY (RAM)	8.00 GB
STORAGE DEVICE	Kingston SSDNow V300 Series SV300S37A/60G 2.5" 60GB SATA III SSD
COMPILER	Visual Studio 2013; 32-bit target; command line = ” /MP /GS /GL /W3 /Gy /Zi /Gm- /O2 /fp:precise /GF /GT /WX- /Zc:forScope /arch:SSE2 /Gd /Oy- /Oi /MD”

GLOBAL PARAMETERS

- The number of unique words per corpus (corpus dictionary) is fixed at $10{,}000$ words. The unique words in the dictionary follow a Zipf distribution, and they are randomly generated with an average length of 6.5 alphabetic characters. Such a dictionary is uniquely constructed for each trial of every experiment.

- For BM25 scoring, parameter b is set to 0.75 and parameter k_1 is set to 1.2. As discussed on page 31, these are typical values.

- Each document of size n (n words) in the corpus separately samples $m = 12n^{\frac{1}{2}}$ unique words from the corpus dictionary, conforming to Heap's law with $K = 12$ and $\beta = 0.5$. Once m unique words are sampled, they are renormalized to make them into a proper distribution. It is this distribution that is used to generate the sequence of words for a document.

  We did this with the intention of making each document approximately follow a Zipf distribution, but with a different subset of words to account for different authors with different but overlapping vocabularies. In hindsight, it would have been sufficient (and perhaps preferable) to have simply sampled n words directly from the corpus dictionary.

- When calculating the outputs for a given input we always use a query set consisting of 30 queries. We then average the outputs over all of those queries where appropriate.

- For each query term in a query in a query set, we seed a document in the corpus with that term with probability p = 0.2 except where otherwise noted. Thus, for a query with k terms, the probability that one or more of its terms occurs in the document is $P[at\ least\ one] = 1 - P[none] = 1 - (1-p)^k$. For $k = 1$, $P[at\ least\ one] = p = 0.2$; for $k = 2$,

$P[at\ least\ one] = 2p - p^2 = 0.36$. Thus, if a query consisting of k terms seeds a corpus of size N, then on average $N\left(1-(1-p)^k\right)$ documents will be seeded with the query term.

In general, this should mean when doing a mean average precision calculation, the expected number of documents relevant to a query with k terms will be $N\left(1-(1-p)^k\right)$. The remainder should be non-relevant, i.e., MAP scores of 0, which means it is irrelevant how they are ranked and can thus be ignored in the mean average precision calculation. Of course, if the secure index does score them with a score of non-zero, that is a sign of a false positive, and should and does subsequently degrade the MAP score. This happens with probability $1-(1-\varepsilon)^k$ for those documents which are nonrelevant, where ε is the false positive rate. On average, for a query with k terms, this will happen $N(1-p)^k\{1-(1-\varepsilon)^k\}$ times.

Once a document is targeted to be seeded by a given query term, all such occurrences of the term will occur within a window size of $w \sim U(min(size(doc), 2000), max(size(doc), 6000))$ except where otherwise noted.

Finally, the number of occurrences of the term within that window will be $n = min\{1, w^{\frac{4}{5}} \times UINF(0.01, 0.001)\}$.

- When measuring precision, MinDist* MAP, or BM25 MAP, each query in the query set (as previously mentioned, there are 30 queries per query set in total) is submitted 10 times for the block-based secure indexes, and the average of those 10 MAP scores is taken to be the actual output.

ADVERSARY SIMULATION RESULTS

In this experiment section, we explore how effective a simulated adversary is at compromising query privacy using the maximum likelihood estimation (MLE) attacks described on page 35.

Obfuscations vs Accuracy

EXPERIMENT DETAILS	
INPUT	unique obfuscations (unique strings in the uniform distribution being sampled from)
OUTPUT	accuracy (proportion of hidden terms correctly mapped to the corresponding plaintext term)
CONSTANTS	1 secret 50 word search vocabulary (every query composed from same 50 unique terms) 50,000 query term history (to be used as data points in MLE) 150,000 samples (in the Monte Carlo simulation to approximate MLE)

Figure 9 Experiment #1

In this experiment, we are interested in seeing how accurately a hypothetical MLE adversary can learn a mapping from hidden terms to plaintext query terms with respect to the unique number of obfuscations for several obfuscation rates (which is just the probability that a random query term will be an obfuscated term). Whenever an obfuscation is injected into a query, we sample the obfuscated term from a discrete uniform distribution consisting of N unique strings (see page 31).

From Figure 10, one may conclude that for a given obfuscation rate, there comes a point at which increasing the number of unique obfuscations has little effect on mitigating the adversary. The lower

the obfuscation rate, the sooner this point is reached. Additionally, the lower the obfuscation rate, the larger the adversary's limiting accuracy as n goes to infinity.

For high obfuscation rates, it is a mistake to let the number of unique obfuscations N be small. We imagine the reason for this is related to the fact that the obfuscations will tend to have a higher frequency than most of the real terms if $\frac{P[obfuscation]}{N}$ is larger than the probability for most real terms. Thus, mapping a non-obfuscated hidden term to the obfuscation class will cause the likelihood of seeing the given history much lower. This presents another area to explore. Instead of sampling the obfuscated terms from a uniform distribution, sample them from a distribution, which is designed to resemble, in some way, the real distribution such that incorrect mappings are less penalized in the MLE calculation.

In Figure 11, we see that (with the same fixed constants as before) the optimal combination of number of obfuscated terms and obfuscation rate—the combination that minimizes the adversary's prediction accuracy—is 50 obfuscated terms and 0.2 obfuscation rate.

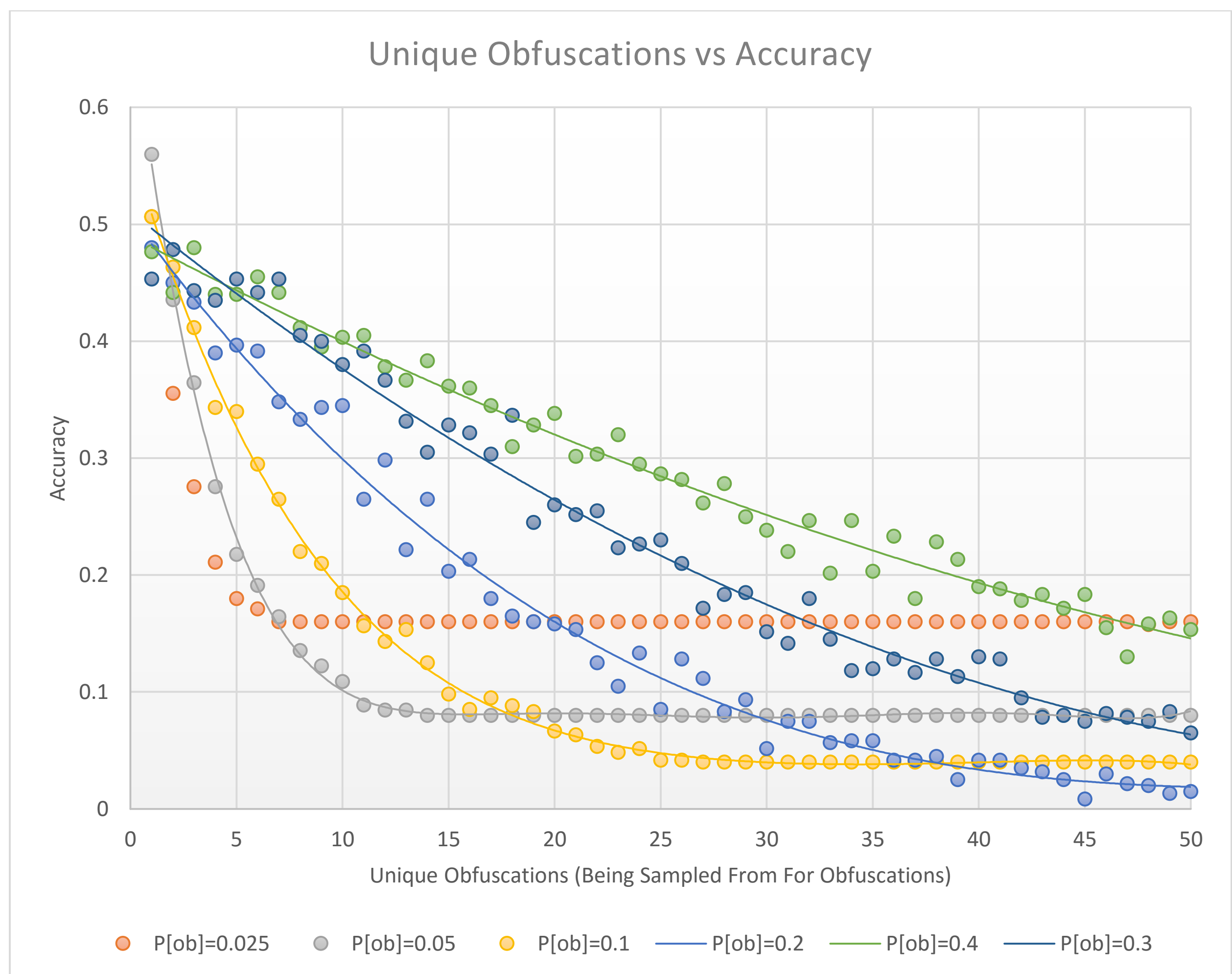


Figure 10 Unique Obfuscations vs Accuracy of Adversary Using MLE Attack

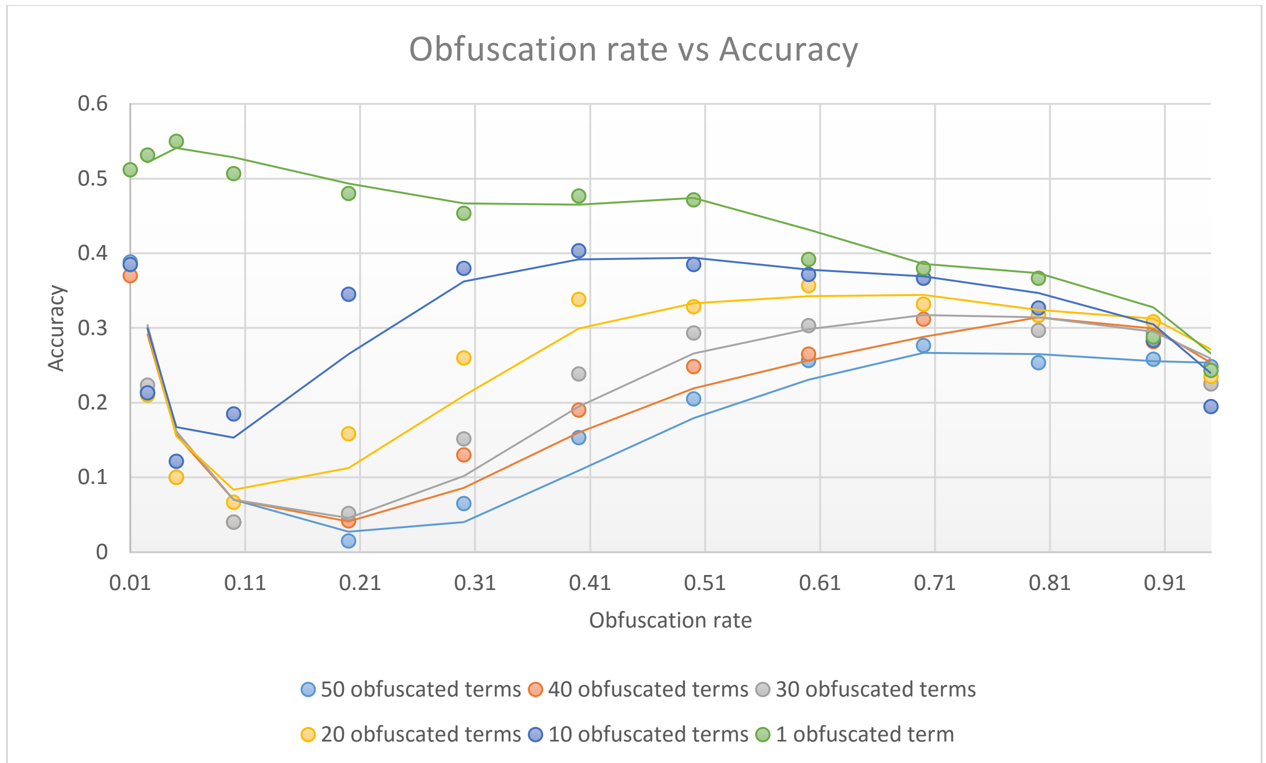


Figure 11 Obfuscation Rate vs Accuracy of Adversary Using MLE Attack

Secrets vs Accuracy

EXPERIMENT DETAILS	
INPUT	secrets
OUTPUT	accuracy (proportion of hidden terms correctly mapped to the corresponding plaintext term)
CONSTANTS	no obfuscations
	50 word search vocabulary (every query composed from same 50 unique terms)
	50,000 query term history (to be used as data points in MLE)
	150,000 samples (in the Monte Carlo simulation to approximate MLE)

Figure 12 Experiment #2

In this experiment, we are interested in seeing how effective secrets are at mitigating the adversary discussed on page 35. Increasing secrets does seem to mitigate the adversary's MLE attack, and as shown in the experiment on page 53, it comes at little to no cost to MAP accuracy and query lag time (except for BSIB's lag, but only slightly). However, it does cost in terms of inflating the secure index size.

Additionally, the marginal value of secrets has diminishing returns. Eventually, there comes a point where it hardly makes a difference at all, but you are likely to run out of memory space before that happens.

It is worthwhile pointing out that the secrets for a given term are sampled from a discrete uniform distribution. This probably limits the effectiveness of having secrets. The adversary may be able to infer the underlying plaintext distribution using big data and statistics, but the adversary cannot know (just as with obfuscations) the distribution of an individual user's secret distribution which may be randomly re-defined periodically (not only an unknown distribution, but a moving

distribution). Indeed, each user can have their own way of sampling secrets (and the same is true for sampling obfuscated terms). We expect that any experimental outcomes that do this would look even more promising.

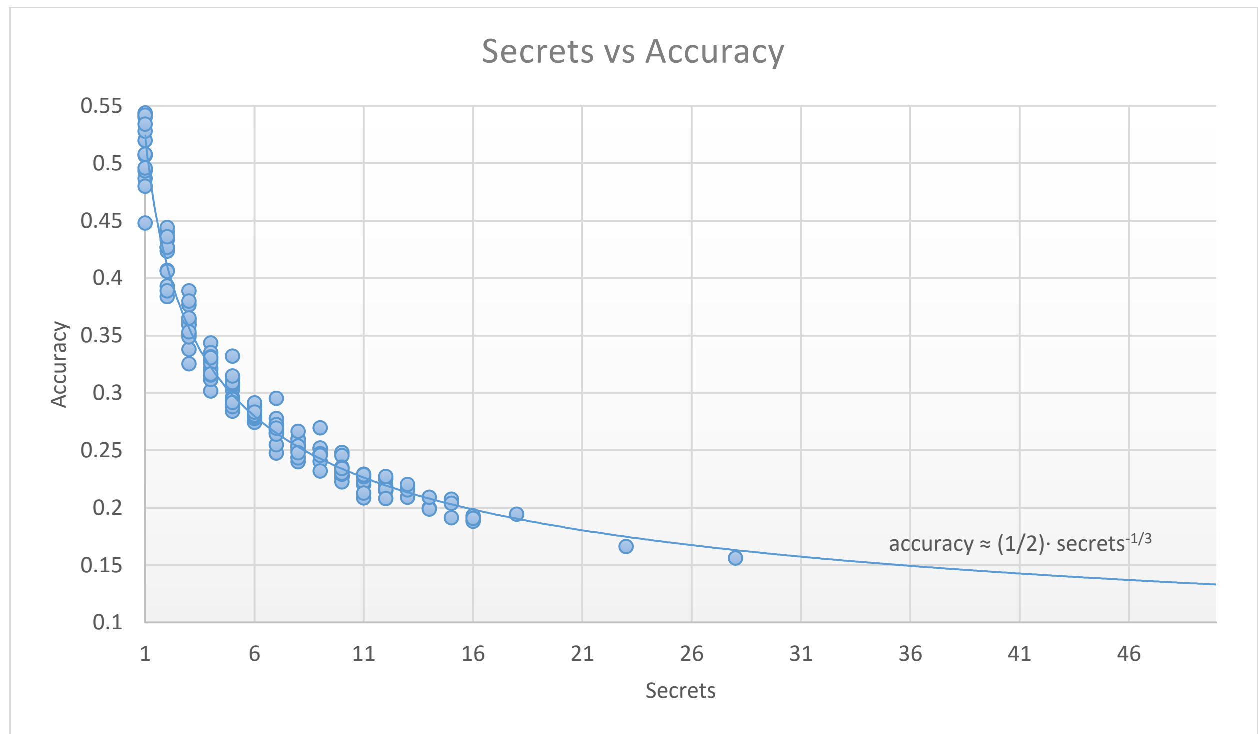


Figure 13 Secrets vs Accuracy of Adversary Using MLE Attack

History Samples vs Accuracy

EXPERIMENT DETAILS	
INPUT	secrets, history size
OUTPUT	accuracy (proportion of hidden terms correctly mapped to the corresponding plaintext term)
CONSTANTS	no obfuscations or obfuscation rate $= 0.2$
	50 word search vocabulary (every query composed from same 50 unique terms)
	150,000 samples (in the Monte Carlo simulation to approximate MLE)

Figure 14 Experiment #3

As the number of query samples (history) increases, so too does the predictive accuracy of the adversary as expected; the more data the adversary has to learn the model (the mapping from hidden terms to plaintext terms), the more accurate the model should be.

Indeed, in Figure 17, for a history size of 512k, if only using one secret the adversary has a 93% accuracy rate. Increasing the number of secrets to 16 reduces the adversary's accuracy rate to 36%. This is certainly an improvement, but preferably, it would be lower yet. Granted, this is a toy problem; there are only 50 words in the user's search vocabulary, for instance. However, according to equation for the curve representing 512k history samples, we would need over 700 secrets to reduce the adversary's accuracy to 10%. Since secrets inflate the size of the index, this is not a viable option.

However, as discussed elsewhere, if the secrets were not sampled uniformly, much better results could probably be realized. And, of course, secrets may be combined with obfuscations without inflating the secure index size.

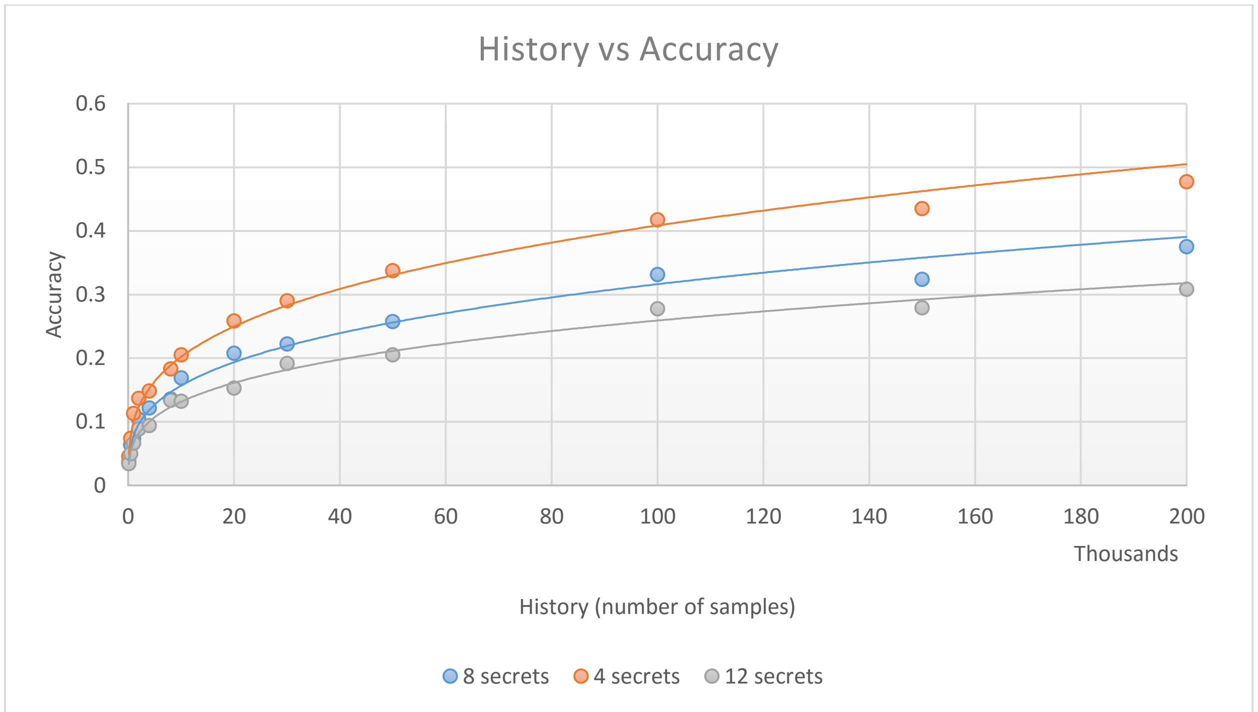


Figure 15 History with Secrets vs Accuracy of Adversary Using MLE Attack

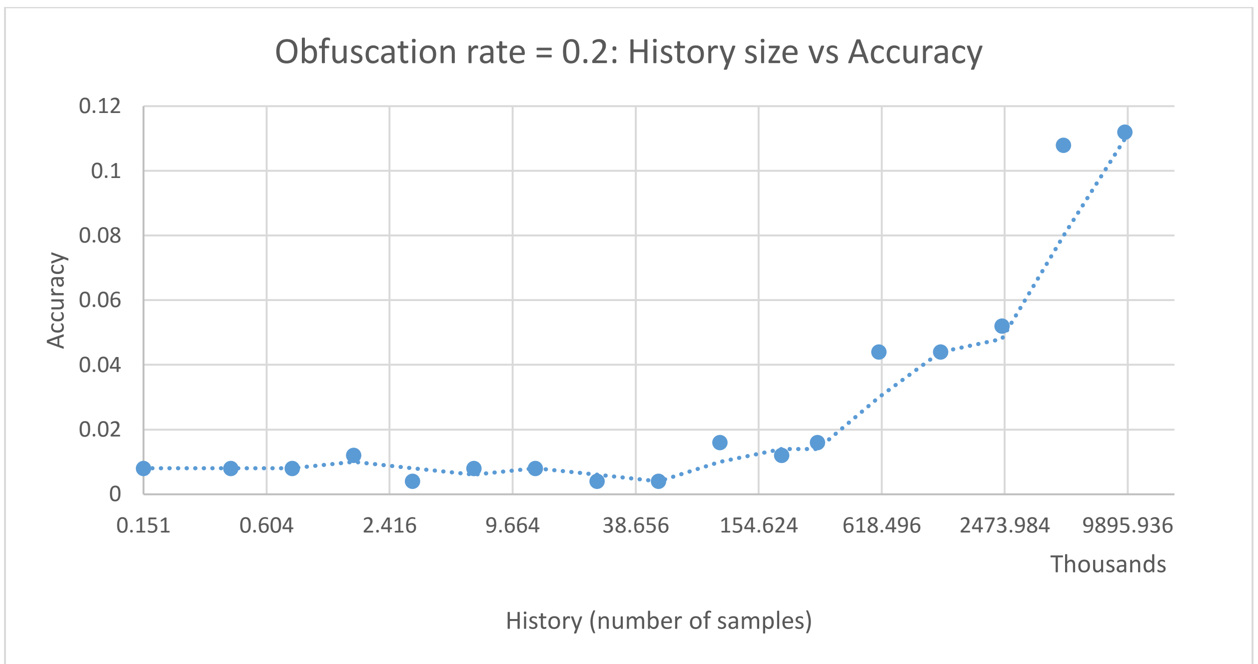


Figure 16 History vs Accuracy of Adversary Using MLE Attack

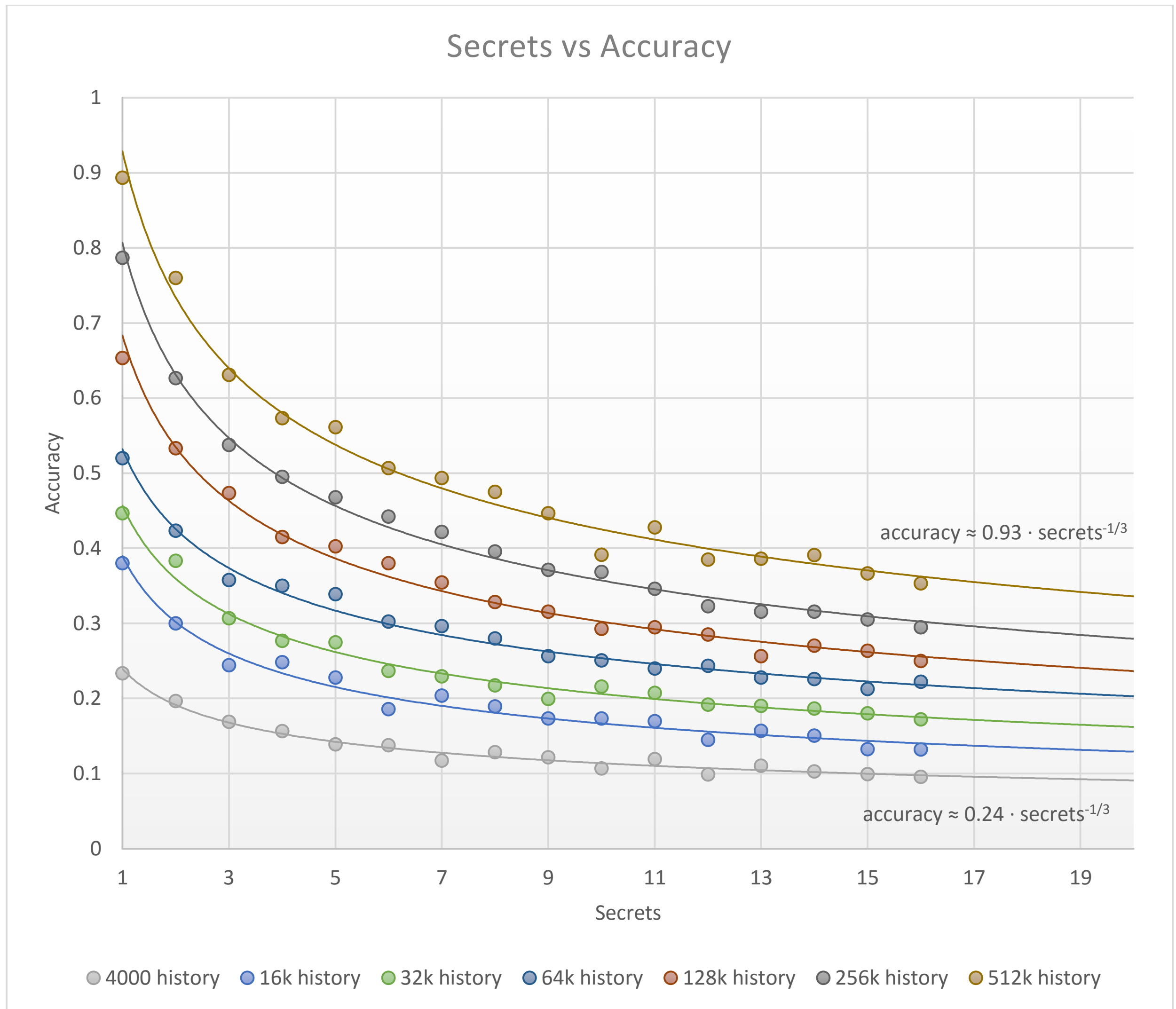


Figure 17 Number of Secrets vs Accuracy of Adversary Using MLE Attack

Vocabulary Size vs Accuracy

EXPERIMENT DETAILS	
INPUT	vocabulary size (number of unique keyword search terms)
OUTPUT	accuracy (proportion of hidden terms correctly mapped to the corresponding plaintext term)
CONSTANTS	no obfuscations
	150,000 samples (in the Monte Carlo simulation to approximate MLE)

Figure 18 Experiment #4

In Figure 19, we see that as the vocabulary size increase, as expected in general less accuracy is achieved. Also as expected, we see that more secrets also consistently degrades the accuracy of the attack.

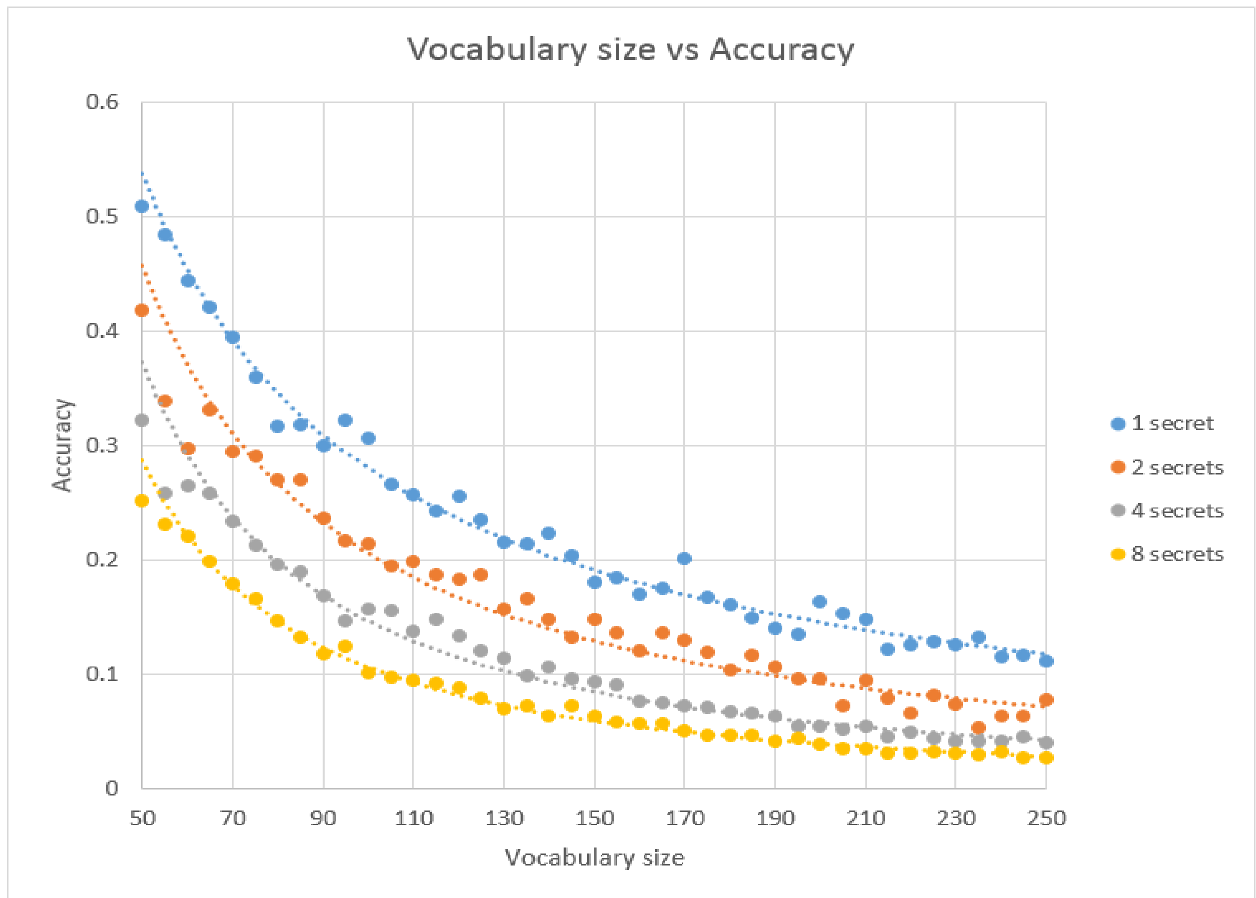


Figure 19 Vocabulary Size vs Accuracy of Adversary Using MLE Attack

SECURE INDEX RESULTS

In this experiment section, we comprehensively compare and analyze the performance of the different secure index types according to a number of different inputs and outputs.

BM25 MAP "Page One" Results

EXPERIMENT DETAILS	
INPUT	location uncertainty vs BM25 Top 10 MAP (first page of results)
OUTPUT	BM25 Top 10 MAP
CONSTANTS	1 secret, 0 obfuscations
	0.001 false positive rate
	1 or 2 words/term, 3 terms/query
	16 pages, 1000 documents (documents/corpus)
TESTBED	machine A

Figure 20 Experiment #5

This is the "page one" Google test. Search users do not want to dig through multiple pages to find what they want. Indeed, studies have shown that Google's second page of results only receives 1.5% of click-through rate.

In this experiment, we get the top 10 results according to the canonical index, and then get the top 10 results each secure index and restrict the mean average precision to only those top 10. This is a much more demanding measure than taking the mean average precision over all of the results.

In Figure 21, PSIP (and PSIF, which tracks PSIP but is not included in this experiment) came out on top, as expected, since it can optionally retain perfect frequency information for words (unigrams and bigrams) in the document (although false positives are still possible). Indeed, it rarely scored under 95%. Also, note that PSIP and PSIF are independent of location uncertainty—PSIF does not even store location information, and PSIP's frequency information is independent of the location uncertainty. The block-based indexes, PSIB and BSIB, also score above 95%, but their scores expectedly trail off as the location uncertainty increases.

To ground the results in Figure 21, let us consider how a completely random top 10 list would score on MAP. In Figure 22, we return 10 random documents out of 250 documents and then calculate its MAP score (note that the real experiment is even more unforgiving since it draws the top 10 results from 1000 documents). Figure 22 shows a histogram of the results. Note that over 90% of the results have a MAP between 0.0 and 0.1. Compare the random results with the results returned from the secure indexes in Figure 21. They all do remarkably well by comparison—clearly much better than random since no trial out of the millions tested had a score higher than 0.4 in the random tests, while no trial had a score less than 0.7 on the secure index tests.

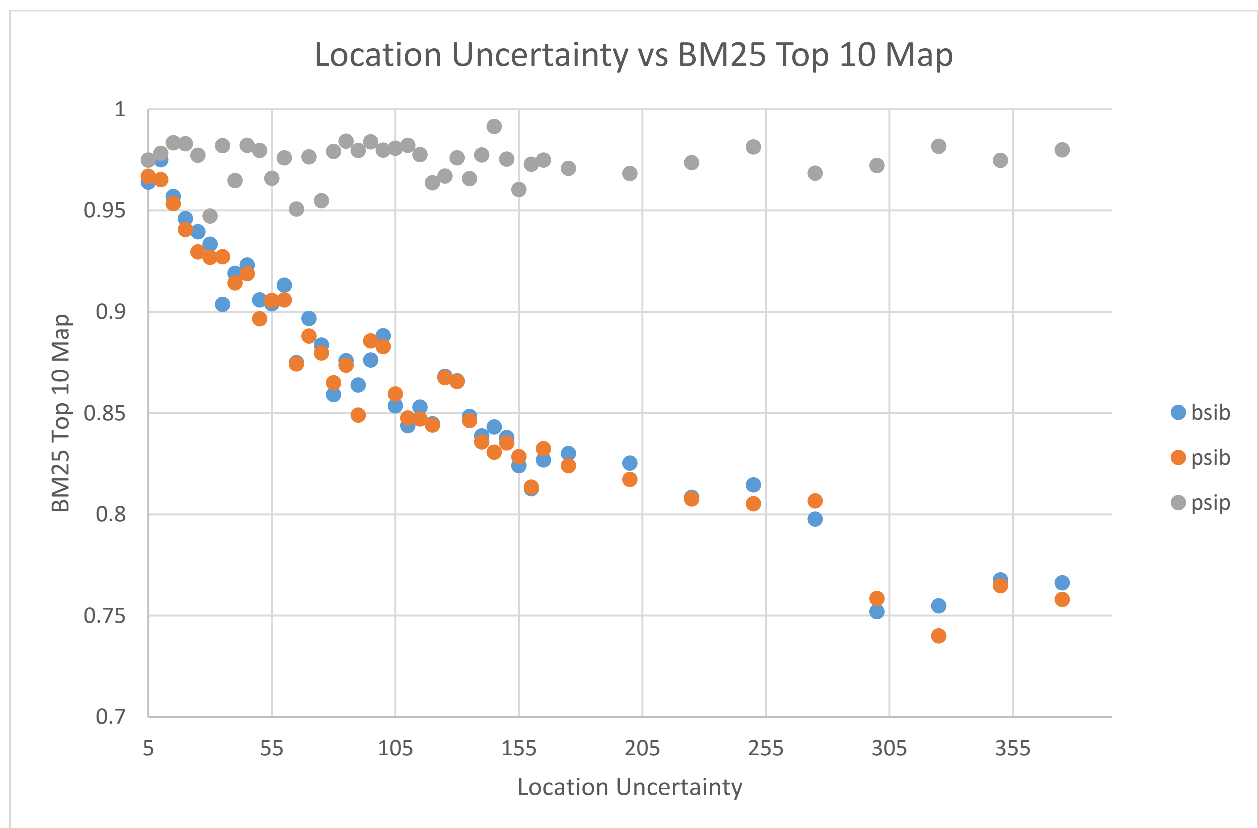


Figure 21 Location Uncertainty vs Accuracy of Top 10 BM25 Search Results

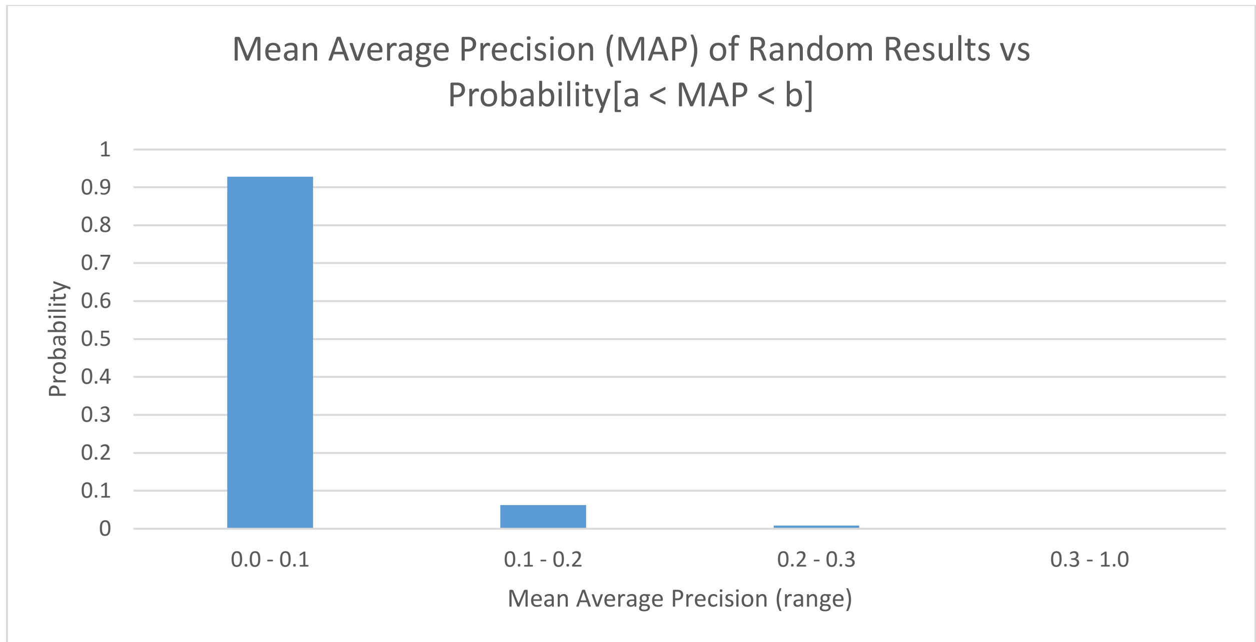


Figure 22 Mean Average Precision of Random Results

MinDist "Page One" Results*

EXPERIMENT DETAILS	
INPUT	location uncertainty vs MinDist* Top 10 MAP (first page of results)
OUTPUT	MinDist* Top 10 MAP
CONSTANTS	1 secret, 0 obfuscations
	0.001 false positive rate
	1 or 2 words/term, 3 terms/query
	16 pages, 1000 documents (documents/corpus)
TESTBED	machine A

Figure 23 Experiment #6

Once again, we see PSIP pulling ahead. However, discouragingly, no matter which secure index is chosen, location uncertainty must be quite modest for MinDist* to achieve competent scores.

The MinDist* measure is more sensitive to location uncertainty than BM25 is to frequency uncertainties. This makes sense. When two terms are only separated by a couple of words, they are likely mutually relevant, but as the distance between them grows they rapidly decrease in mutual relevance. MinDist* captures this intuition: it only scores documents high when they contain the terms (in the query) at a sufficiently close distance.

However, when location uncertainty is moderately large, two terms that are approximated to be only a couple words apart may actually be much further apart (even pages apart). This can have a large, negative impact on the MinDist* ranked output. Unfortunately, large location uncertainties are desirable for confidentiality (see page 37).

Despite these observations, the secure indexes—especially PSIP—still do reasonably well (e.g., they do much better than random chance, as demonstrated in Figure 22). Moreover, encrypted search users will probably be more willing to dig deeper into the results to find what they are searching for.

Note that the intuition behind MinDist* proximity sensitivity is the primary motivation for PSIM (see page 28), which can optionally preserve perfect minimum pairwise distance information for terms in the document that are up to k words apart.

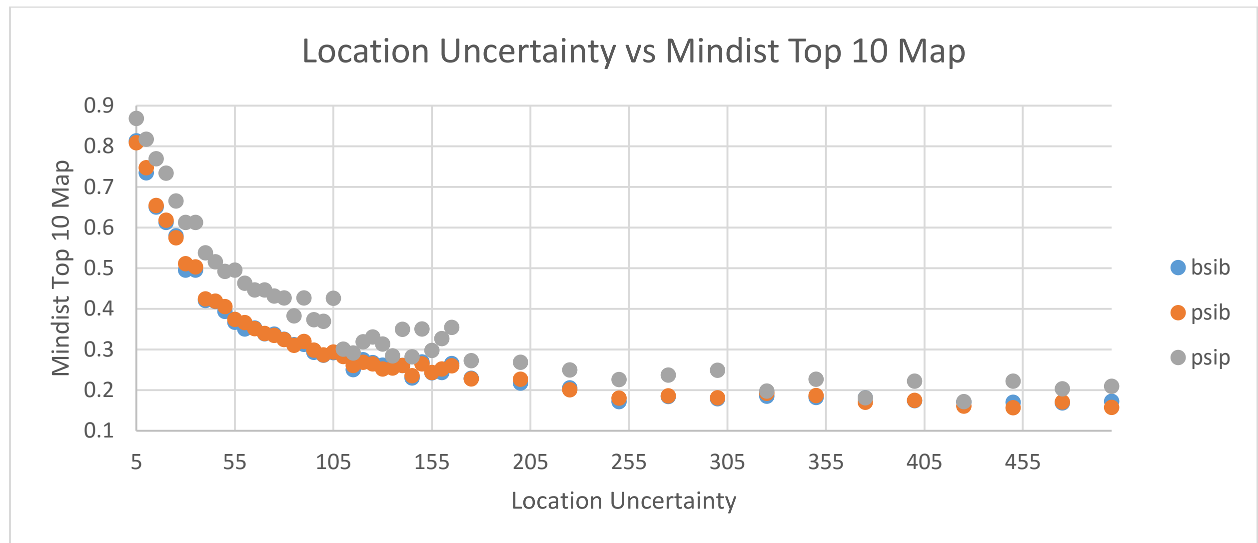


Figure 24 Location Uncertainty vs Accuracy of Top 10 MinDist Search Results*

Secrets vs Compression Ratio, Build Time, and Load Time

EXPERIMENT DETAILS	
INPUT	secrets
OUTPUT	compression ratio (ratio of secure index size to document size), load time, build time
CONSTANTS	12 pages, 256 location uncertainty

Figure 25 Experiment #7

The only outputs secrets affected were build time, load time, and secure index size. For PSIB and PSIF, load time is nearly constant with respect to secrets. However, all of the secure indexes flatten out as the secrets increase (as they do for build time, also).

The compression ratio output is linear with respect to the number of secrets; this certainly makes sense, as each secret variation of each term will be dedicated a constant number of bits.

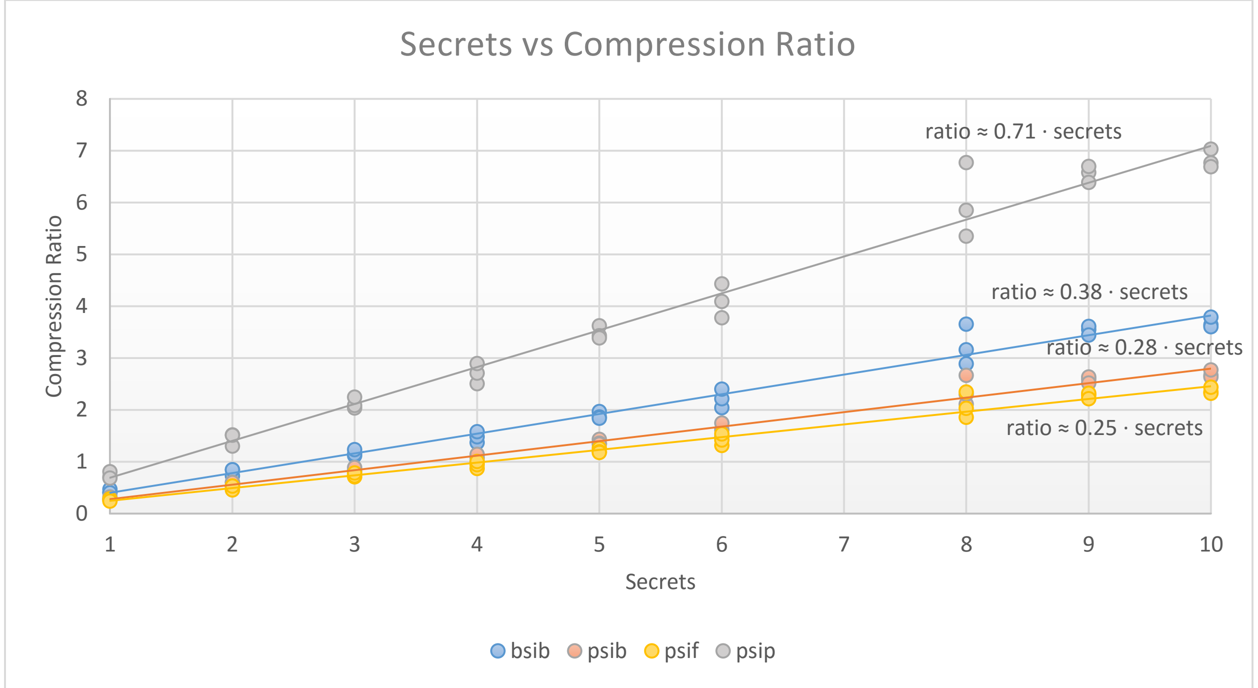


Figure 26 Number of Secrets vs Compression Ratio

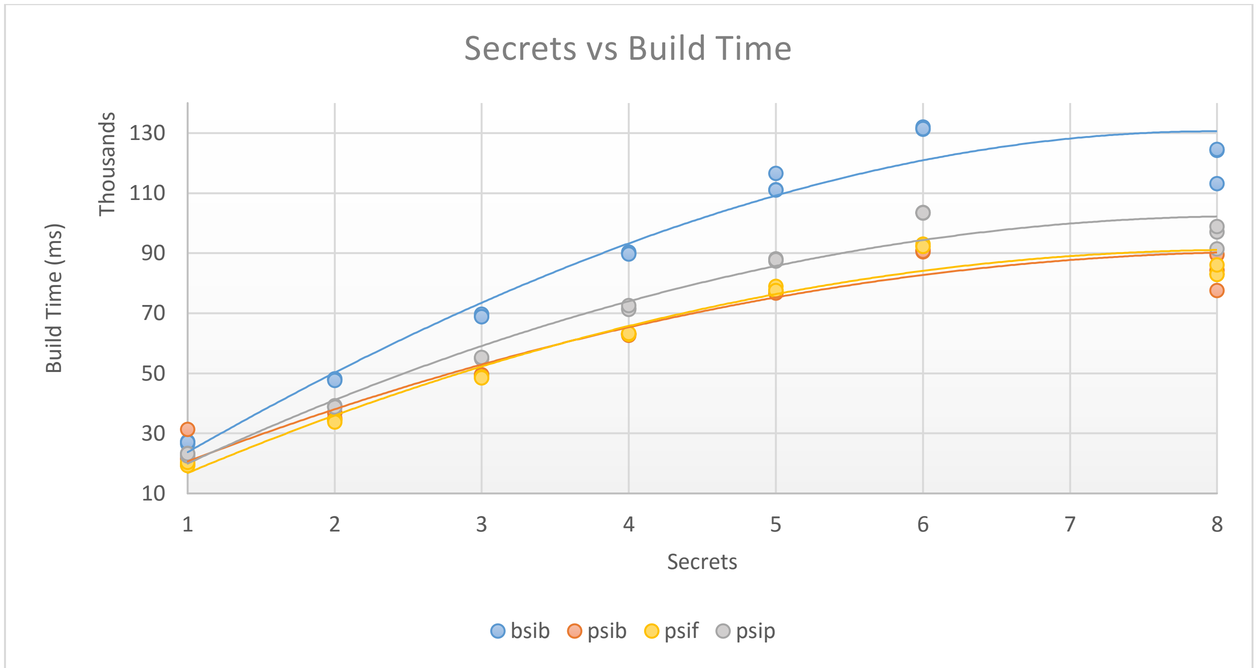


Figure 27 Number of Secrets vs Secure Index Build Time

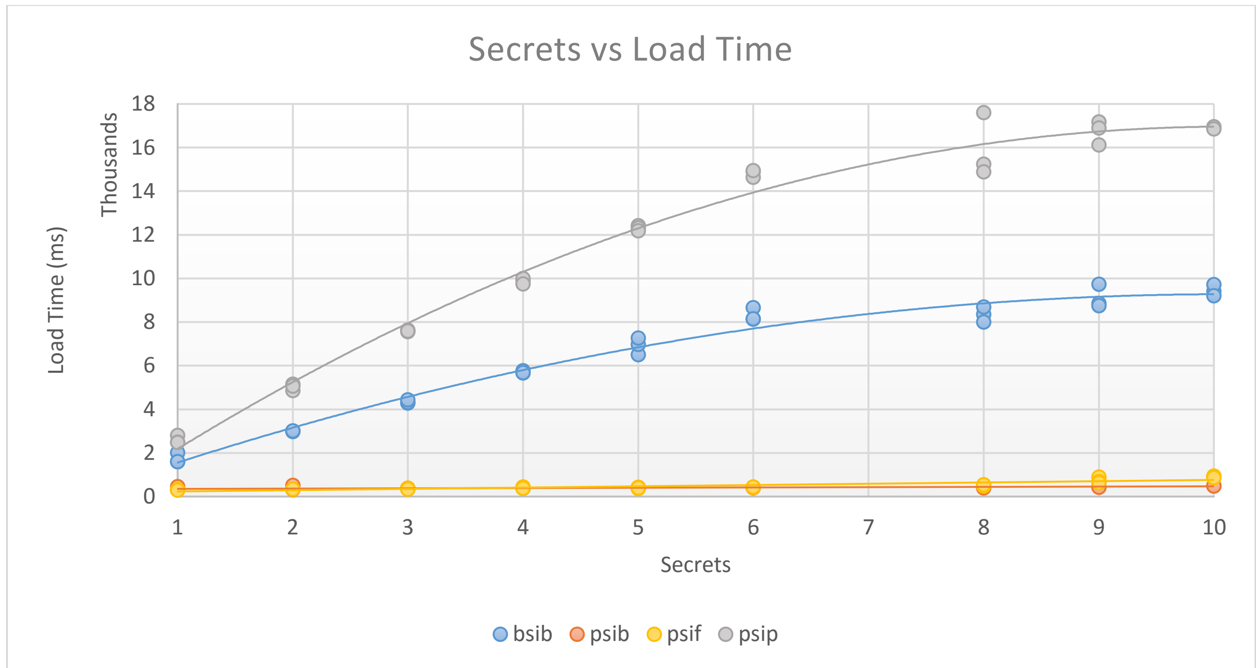


Figure 28 Number of Secrets vs Secure Index Load Time

False Positive Rate vs BM25 MAP and Precision

EXPERIMENT DETAILS	
INPUT	pages (~250 words/page)
OUTPUT	BM25 MAP, precision
CONSTANTS	1 term/query, 1 or 2 words/term
	1 secret, 0 obfuscations
	128 location uncertainty
	0.001 false positive rate
	12 pages, 1000 documents (documents/corpus)
TESTBED	machine A

Figure 29 Experiment #8

While not much space is saved by decreasing the false positive rate, the primary advantage in having a high false positive rate is its effect on confidentiality. The higher the false positive rate, the less certain the information in the secure index is (see page 39 for more analysis).

On the one hand, Figure 30 paints an encouraging picture for BM25 scoring. Indeed, false positives occurring even a quarter of the time on negative examples still result in a BM25 MAP of ~0.7. And this is only for 1 term/query and 1 or 2 words/term; BM25 tends to perform better when given more terms, as other experiments demonstrate.

On the other hand, Figure 31 paints a less encouraging picture for precision (for Boolean search). If false positives occur a quarter of the time here, only 50% accuracy is achieved. Compared to precision, BM25 is far less sensitive to the false positive rate. This makes sense; if a false positive happens when measuring precision, it will admit a term that should not be included in the result set, which will certainly effect its precision negatively. However, BM25 is ranking the documents. Thus, even if a document is falsely hitting on a search term, it is the order that counts—how the document is ultimately ranked.

For instance, since BM25 scoring accounts for “rarity” (see page 13), other documents in the corpus will also falsely hit on the term with probability[39] 0.25, which will cause the false hit to not be very discriminating—that is, it is not rare in the corpus since 25% of the documents have it.

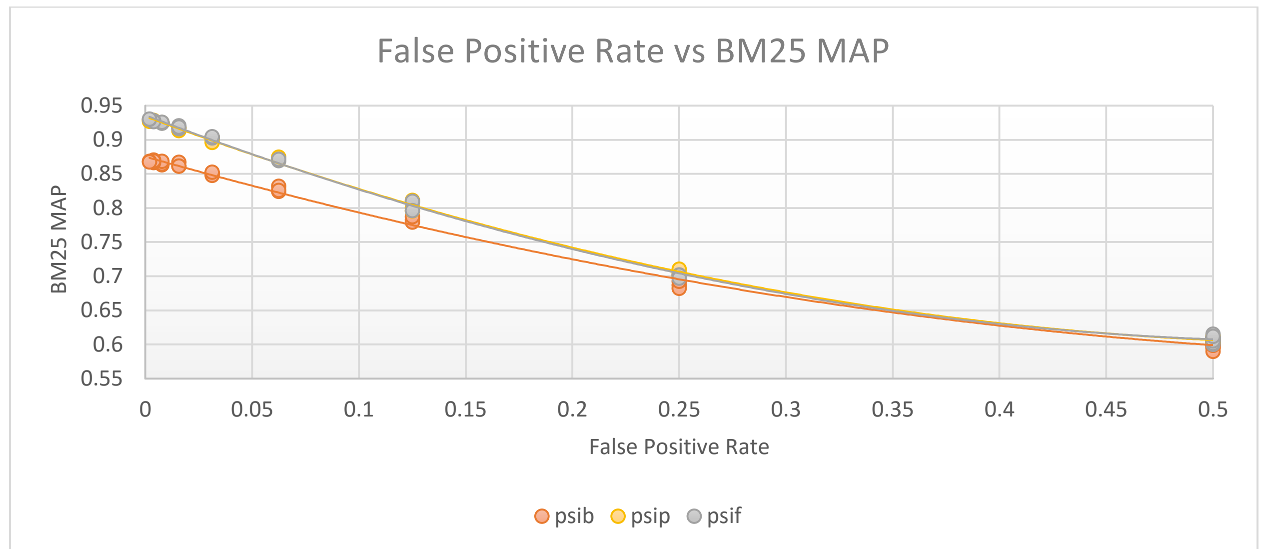


Figure 30 False Positive Rate vs BM25 Mean Average Precision

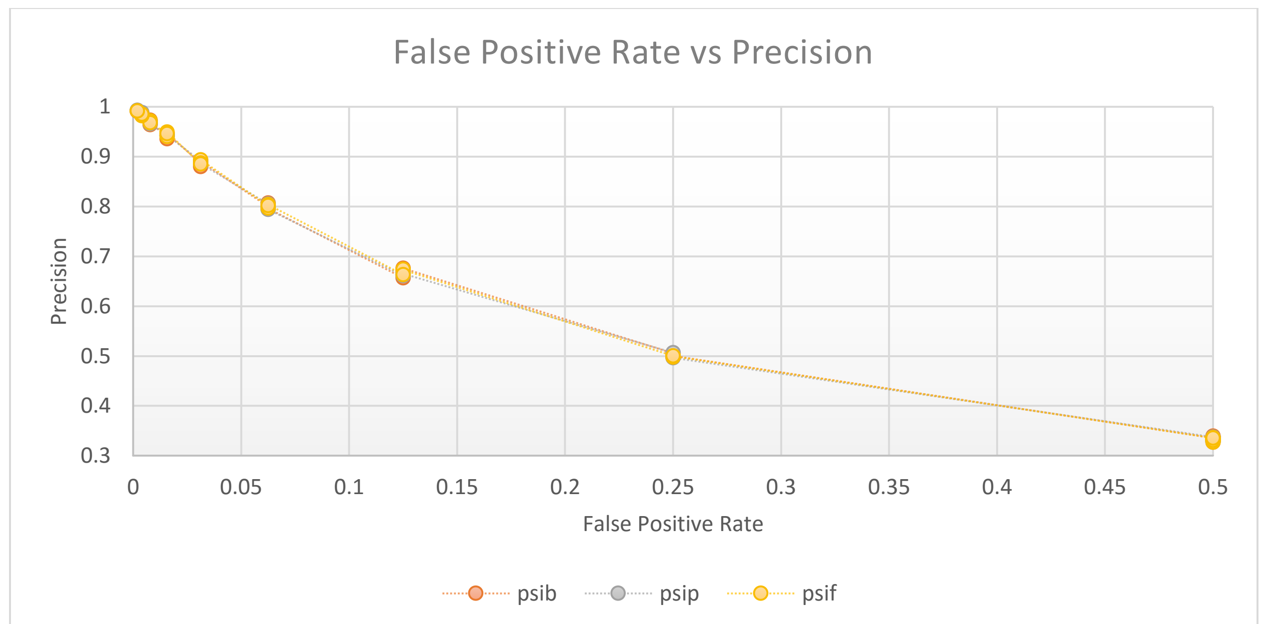


Figure 31 False Positive Rate vs Precision

[39] The probability that a search term appears in a document is ~0.25. In practice, BM25 may do better than our experiments suggest since we did not include particularly rare terms in the query set.

Obfuscations vs BM25

EXPERIMENT DETAILS	
INPUT	Obfuscations (per query)
OUTPUT	BM25 MAP, BM25 lag
CONSTANTS	1 secret, 256 location uncertainty
	0.001 false positive rate
	12 pages, 1000 documents (documents/corpus)
	1 term/query, 1 or 2 words/term *or* 6 terms/query, 6 words/term
TESTBED	machine B

Figure 32 Experiment #9

In Figure 33, we plot obfuscations versus BM25 MAP on a rather large class of query—6 terms/query, 6 words/term. PSIP and PSIF perform very close to 100% and each additional obfuscation per query only reduces BM25's MAP score by 0.005%. PSIB and BSIB also do well and remain largely unaffected by the obfuscated terms as well.

Figure 34 reveals that every obfuscated term added to the query increases the BM25 lag time by 0.02 milliseconds compared to PSI-based indexes, which increase at a rate of 0.014 milliseconds per obfuscated term. Note that these lag times cannot be directly compared with the lag time reported in other BM25 experiments, as this experiment was conducted by a different machine. However, it does not seem unreasonably slow. Also, note that every one of the queries is slow compared to smaller, more typical queries performed in other experiments.

Finally, Figure 35 shows BM25 MAP on a more modest set of queries consisting of 1 term/query and 1 or 2 words/term, which results in an across the board reduction in the BM25 MAP score. As discussed elsewhere, BM25 generally does better on queries that are more complicated. Unfortunately, each additional obfuscated term injected into the query also has a larger negative impact on the BM25 MAP score, i.e., -0.008 versus -0.003.

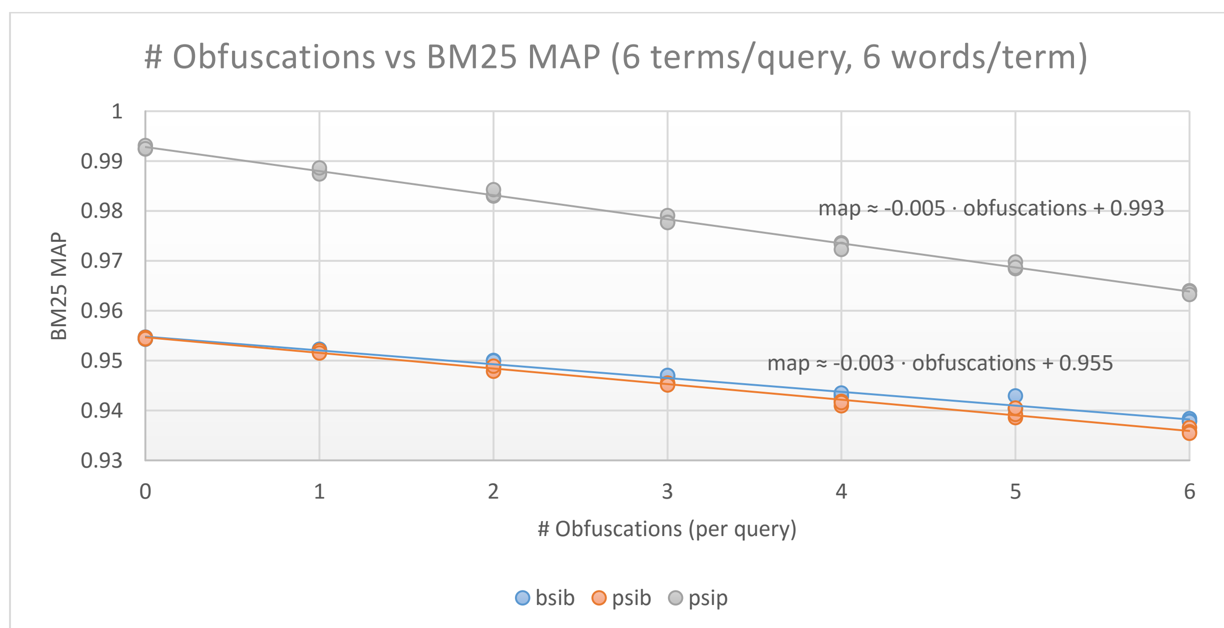


Figure 33 Obfuscations/Query vs BM25 MAP with 6 Terms/Query, 6 Words/Term

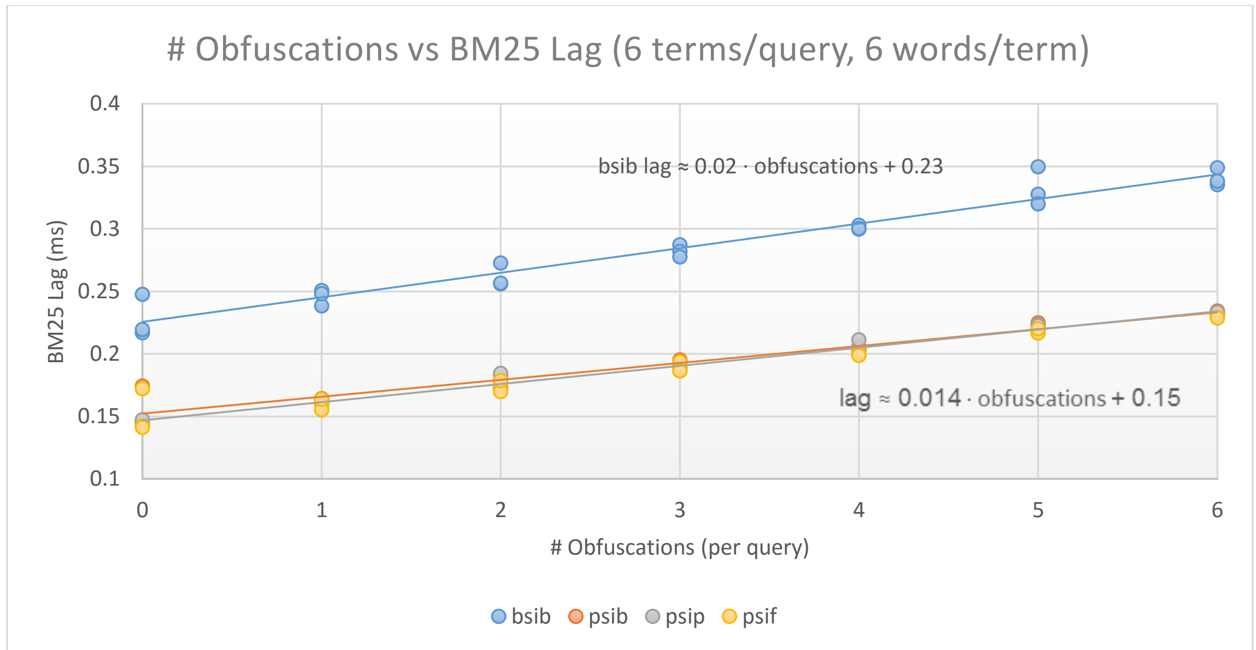


Figure 34 Obfuscations/Query vs BM25 Lag Time with 6 Terms/Query, 6 Words/Term

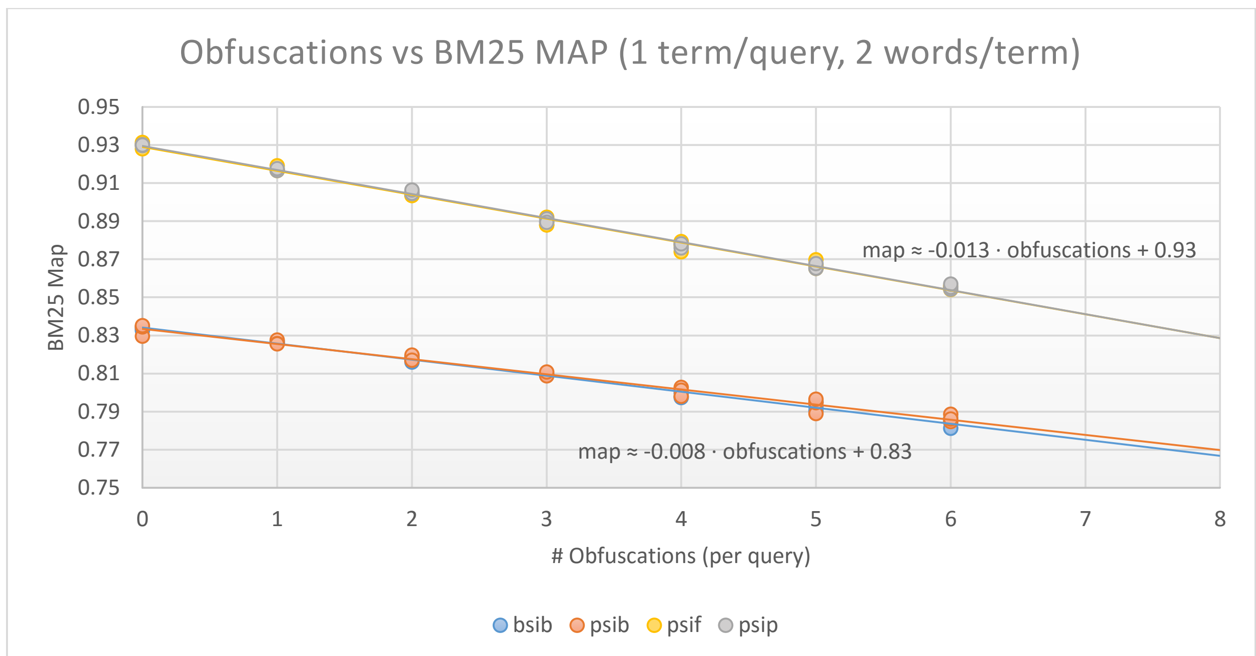


Figure 35 Obfuscations/Query vs BM25 MAP with 1 Term/Query, 2 Words/Term

*Obfuscations vs MinDist**

EXPERIMENT DETAILS	
INPUT	# Obfuscations (per query)
OUTPUT	MinDist* MAP
CONSTANTS	1 secret, 256 location uncertainty
	12 pages, 1000 documents (documents/corpus)
	0.001 false positive rate
	6 terms/query, 1 or 2 words/term
TESTBED	machine B

Figure 36 Experiment #10

Earlier attack simulations demonstrated the effectiveness of obfuscations in mitigating attacks. Judging by Figure 37, increasing obfuscations have almost no effect on MinDist MAP scores. This is certainly welcome news—we can exploit obfuscations without incurring much, if any, loss in MinDist* accuracy.

Figure 38 shows a linear relationship between obfuscations and MinDist* lag time. This makes sense; it is essentially the same increase in lag time expected from any additional query terms—obfuscated terms or otherwise.

Note that PSIP is pulling ahead in a majority of the benchmarks measuring lag time or mean average precision. This was expected; subsequent experiments will expand on why this is happening.

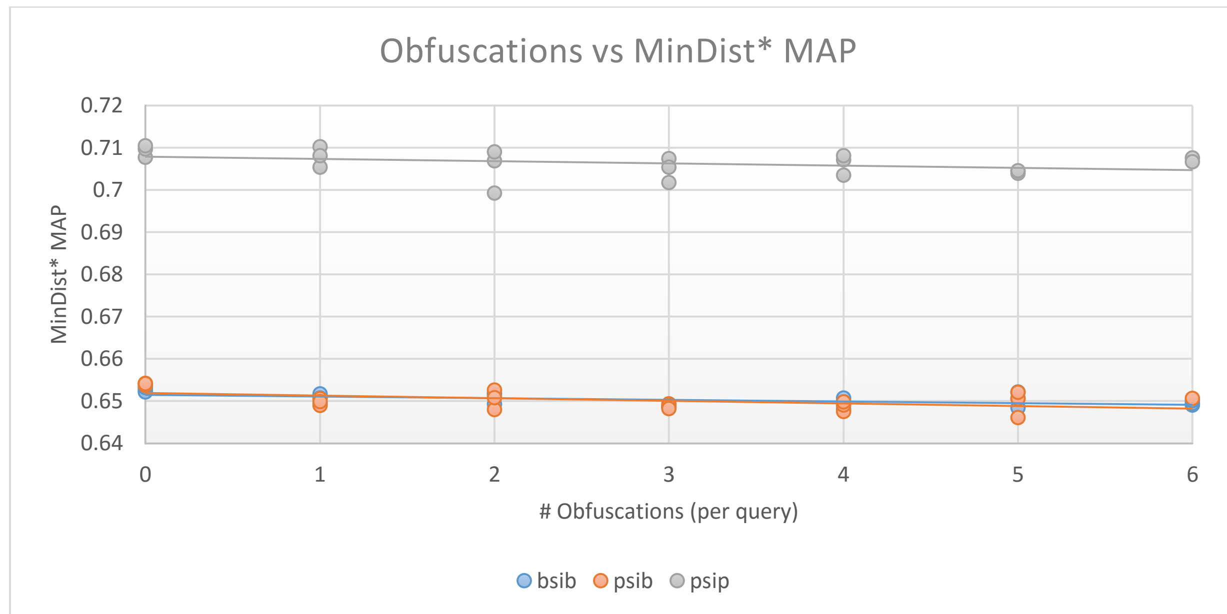


Figure 37 Obfuscations/Query vs MinDist Mean Average Precision*

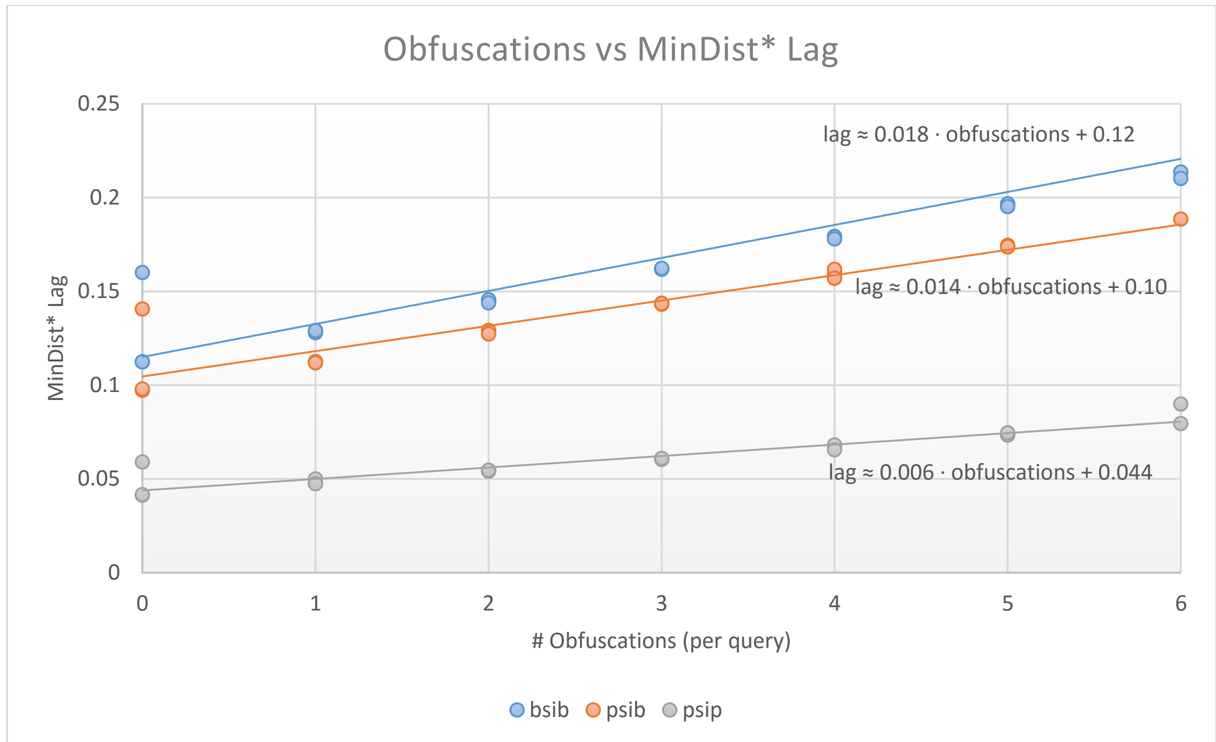


Figure 38 Obfuscations/Query vs MinDist Lag Time*

Pages vs Secure Index Size

EXPERIMENT DETAILS	
INPUT	pages (~250 words/page)
OUTPUT	secure index size (bytes)
CONSTANTS	1 secret, 256 location uncertainty
	0.001 false positive rate
	1000 documents (documents/corpus)
TESTBED	machine A

Figure 39 Experiment #11

As discussed elsewhere, PSIB is optimized for smaller documents. For PSIP and BSIB, every page is approximately 1700 and 1500 bytes respectively; their secure index sizes are linearly dependent upon their page counts.

However, as demonstrated Figure 40 and Figure 41, the sparse bit vector representation used in the PSIB does well for small to moderate pages, but explodes as the pages increase past a certain point (~ 100 pages). It is quadratic with respect to page count rather than linear. For small page counts, the squared component is dominated by the linear component, but for large page counts the squared component dominates.

The point of intersection between PSIB and BSIB is ~ 50 pages. This is the size of a relatively large document; for larger documents (e.g., books) with more than 50 pages, it may be advisable to segment them into smaller chunks.

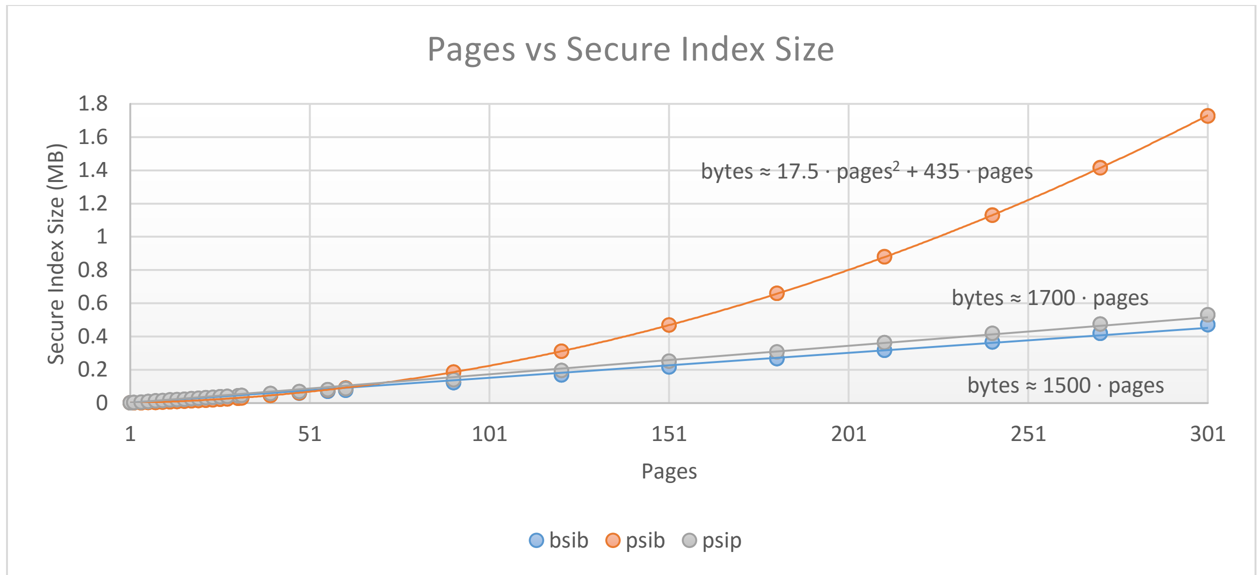


Figure 40 Page Count vs Secure Index Size

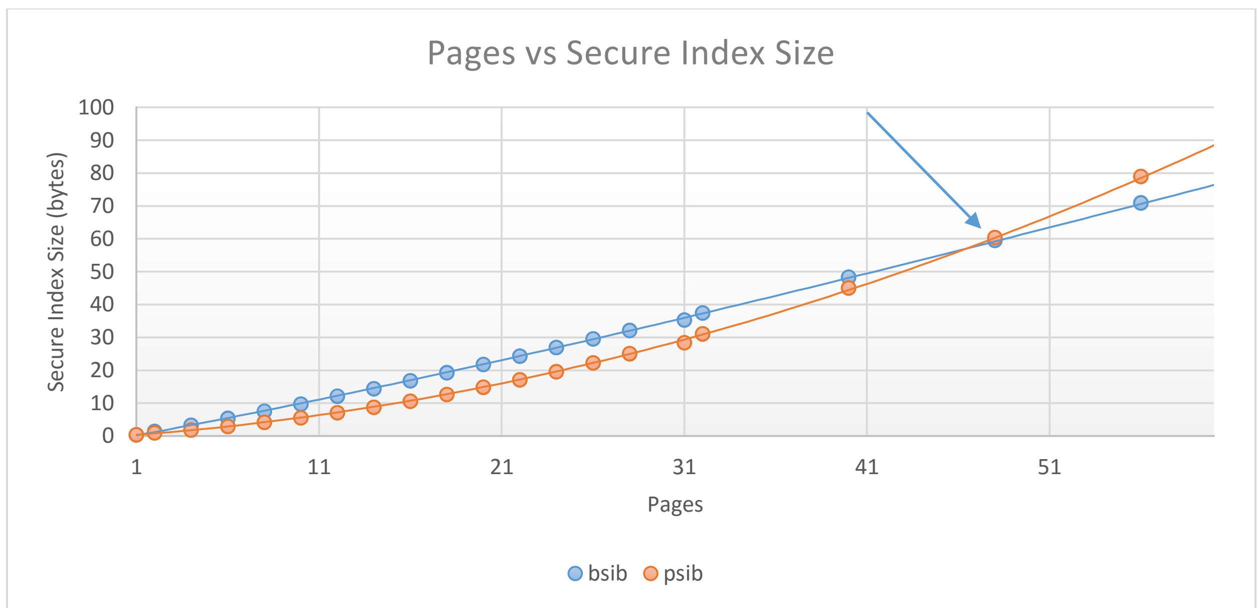


Figure 41 PSIB and BSIB Intersection for Pages/Document vs Secure Index Size

Blocks vs Secure Index Size

EXPERIMENT DETAILS	
INPUT	blocks (block segments per document)
OUTPUT	secure index size (bytes); compression ratio (ratio of secure index size to document size)
CONSTANTS	1 secret, 256 location uncertainty
	0.001 false positive rate
	1000 documents (documents/corpus)
TESTBED	machine A

Figure 42 Experiment #12

In the previous experiment, we examined how page count affected secure index size while location uncertainty was held constant at 256. This had the effect of increasing the number of blocks per

PSIB and BSIB as the page count increased. This motivates us to consider how the block count per PSIB and BSIB are affects secure index size.

In Figure 43, we see that the lines for PSIB and BSIB cross at ∼47 blocks. If the primary metric of interest is secure index size (as measured by memory allocation size), this point of intersection represents a dividing line. To the left of the line, PSIB is preferable; to the right of the line, BSIB is preferable. Note, however, that PSIB retains its significant advantage in other outputs, like query lag times.

In Figure 44, we see that for a small number of blocks, PSIB is a small fraction—a quarter—the size of the actual document. However, it grows linearly as the block count increases. On the other hand, BSIB converges (to a first approximation) to a constant factor of ∼0.75 the size of the document as the block count increases.

A high block count is ideal for MinDist* and BM25 MAP accuracy—it reduces the location uncertainty—but there is a trade-off between such accuracy and the amount of information leaked about the document.

PSIP does not represent a document as blocks; it represents a document as postings lists. Thus, location uncertainty can be adjusted to any desired value and PSIP's file size (and query lag times) will remain the same.

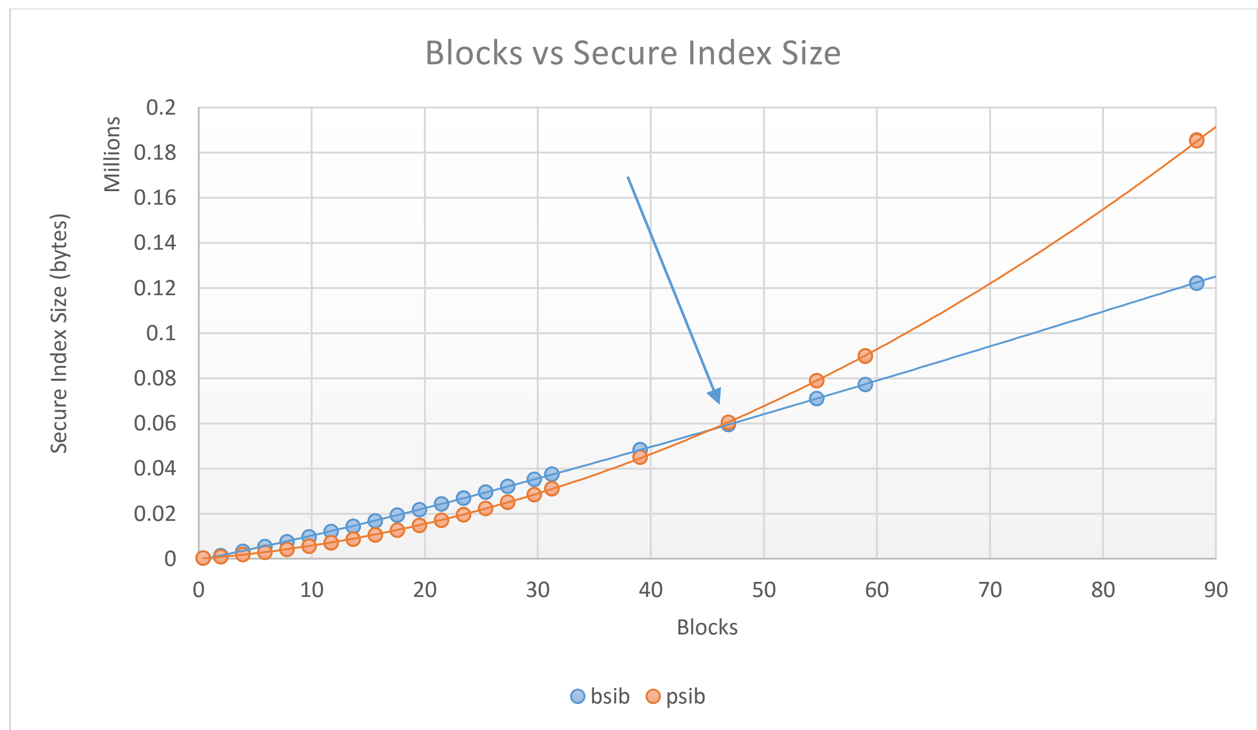


Figure 43 Blocks per PSIB/PSIB vs Secure Index Size

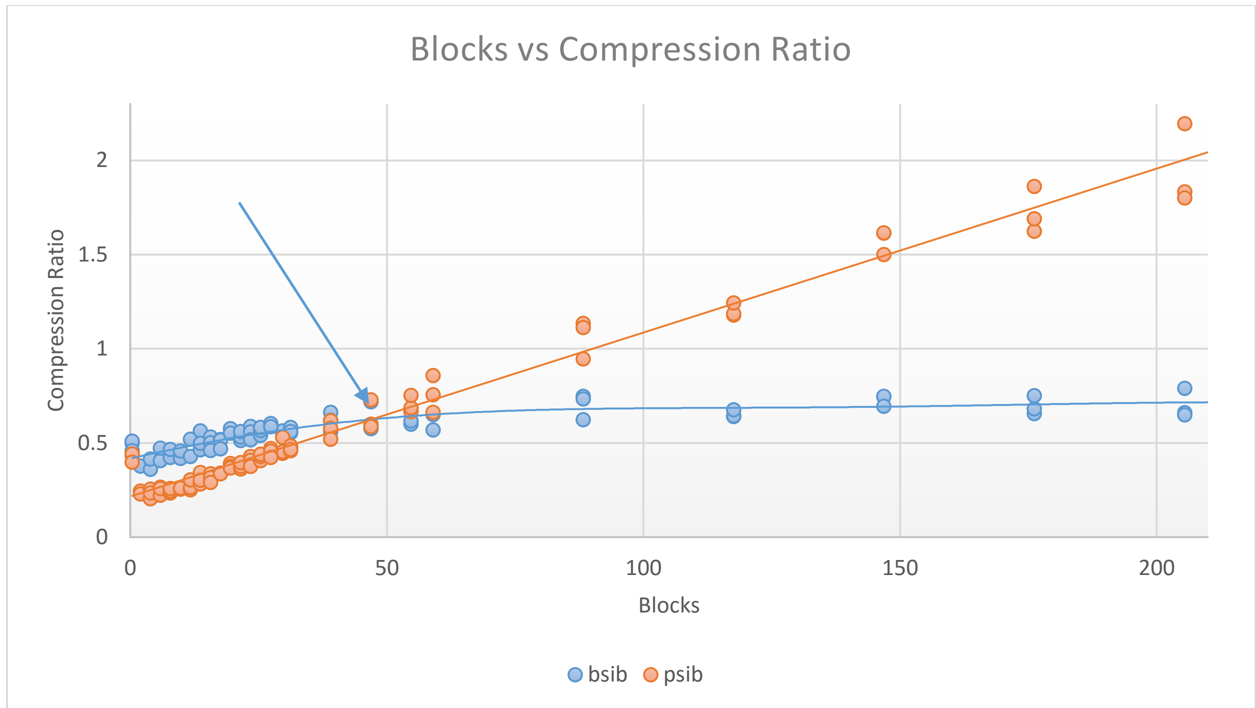


Figure 44 Blocks per PSIB/PSIB vs Compression Ratio

Documents (per corpus) vs Corpus Secure Index Size

EXPERIMENT DETAILS	
INPUT	documents (per corpus)
OUTPUT	corpus secure index size (bytes)
CONSTANTS	1 secret, 250 location uncertainty
	16 pages (per document)
	0.001 false positive rate
TESTBED	machine A

Figure 45 Experiment #13

For a reasonably large document consisting of 16 pages (4000 words, 250 words/page), we see that the average document is ~10.5 kilobytes for PSIB, ~16.8 kilobytes for PSIP, and ~23.3 kilobytes for PSIP. Note that, with that many pages and with that location uncertainty, the blocks per document is 16 for PSIB and BSIB; this is under the threshold of ~47 blocks under which PSIB is superior to BSIB. For a corpus of nearly 20,000 documents, the total corpus size is a little over 200 MB; the original corpus was 593 MB.

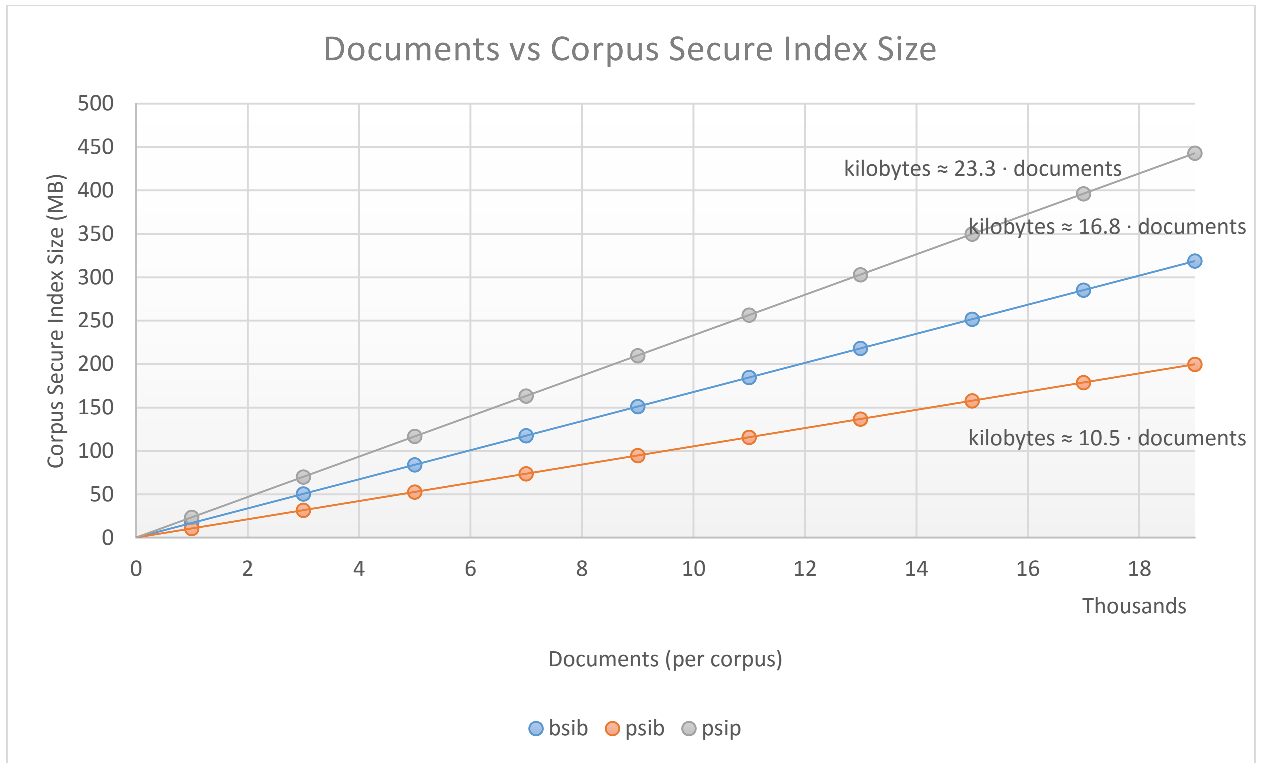


Figure 46 Documents per Corpus vs Corpus Size

Pages vs Build Time

EXPERIMENT DETAILS	
INPUT	pages (~250 words/page)
OUTPUT	build time (milliseconds)
CONSTANTS	1 secret, 256 location uncertainty, 0.001 false positive rate
	1000 documents (documents/corpus)
TESTBED	machine A

Figure 47 Experiment #14

In this experiment, we are interested in seeing how page count (∼250 words/page) affects secure index build time. The byte-size of the document is less important than its page count size. BSIB is nearly twice as slow as PSIB and PSIB, but even BSIB is only 2.4 milliseconds per page.

None of the secure indexes are unreasonably slow; even a document consisting of ∼300 pages takes only a fraction of a second to build. Indeed, the PSIB can build a ∼800 page document in only a second. And, as discussed later, there are significant performance improvements that could be easily realized, e.g., replacing unnecessary cryptographic SHA256 re-hashes with non-cryptographic hash functions.

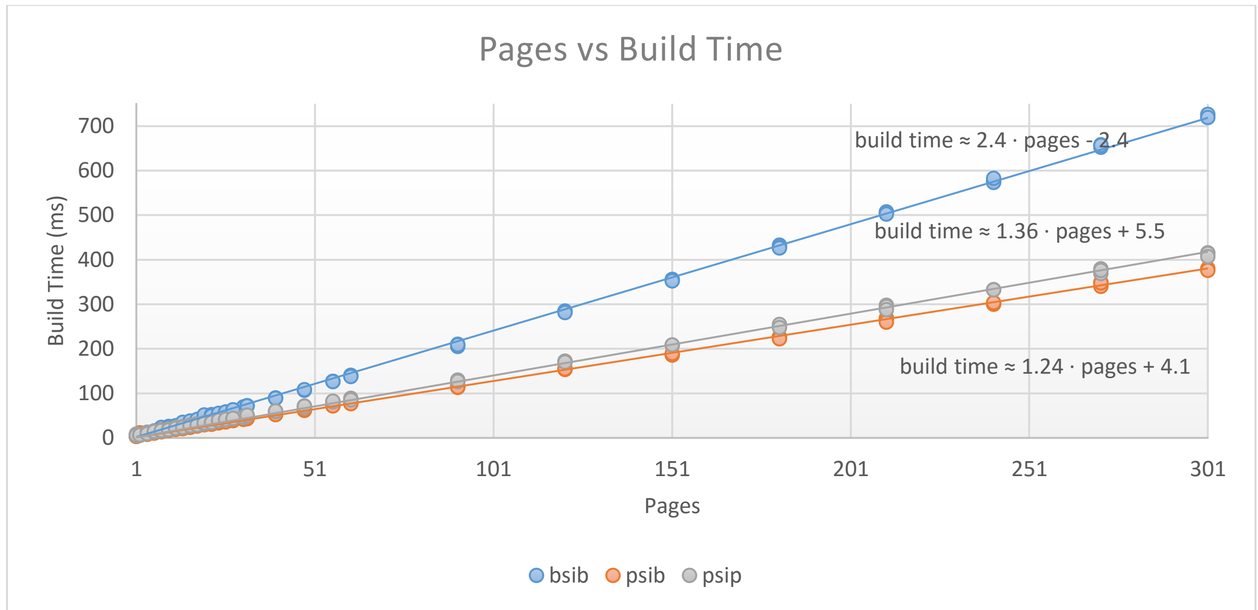


Figure 48 Pages per Document vs Build Time

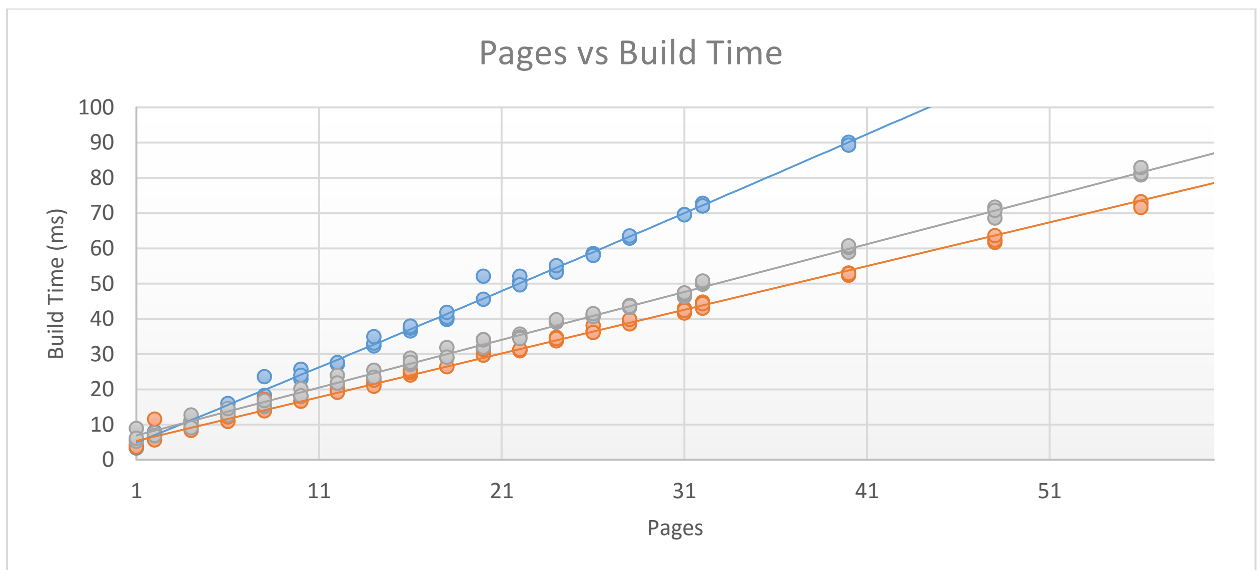


Figure 49 A Closer Look at Pages per Document vs Build Time

Documents (per corpus) vs Build Time

EXPERIMENT DETAILS	
INPUT	documents (per corpus)
OUTPUT	corpus build time (milliseconds)
CONSTANTS	1 secret
	16 pages (per document)
	250 location uncertainty
	0.001 false positive rate
TESTBED	machine A

Figure 50 Experiment #15

The time to build a corpus consisting of reasonably large 16 page documents is 24 milliseconds per document for PSIB, 27 milliseconds for PSIP, and 36 milliseconds for BSIB.

In the previous experiment, we plotted page size vs build time. The line of best fit for PSIB $\text{build time} \approx 1.24 \cdot \text{pages} + 4.1$; for PSIP, the line of best fit was build time ≈ 1.36 · pages + 5.5; finally, for BSIB, the line of best fit was build time $\approx 2.4 \cdot \text{pages} - 2.4$. Each one of these lines of best fit competently predicts the slopes in Figure 51.

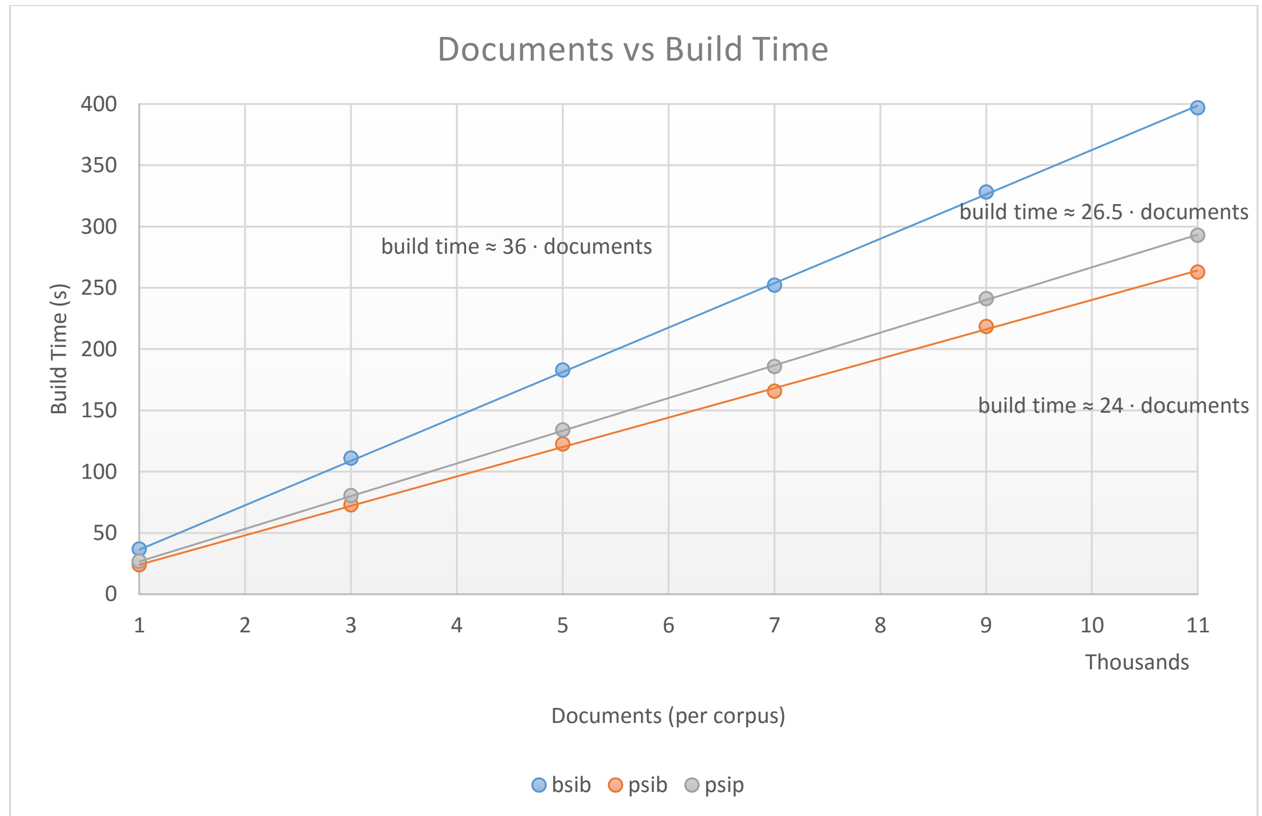


Figure 51 Documents per Corpus vs Total Build Time per Corpus

Pages vs Load Time

EXPERIMENT DETAILS	
INPUT	pages (~250 words/page)
OUTPUT	load time (milliseconds)
CONSTANTS	1 secret, 256 location uncertainty
	0.001 false positive rate
	1000 documents (documents/corpus)
TESTBED	machine A

Figure 52 Experiment #16

In this experiment, we are interested in seeing how page count affects secure index load time. Interestingly, PSIB (and PSIF) is nearly constant when representing documents from 1 page to 300 pages; 300 page documents take only 2.86 milliseconds to load raw from disk.

The other two perform less impressively. With respect to PSIP, we did not make much of an effort to optimize it. For instance, we load a term's postings list as a vector of `varints`[40], which incurs significant vector construction overhead as the number of terms in the document increases. More efficient representations of posting lists—in terms of both construction overhead and compression ration—are discussed on page 26. With respect to BSIB, as the document size increases, the overhead of de-serializing a larger number of Bloom filters may take a toll. However, the serialization seems otherwise efficient.

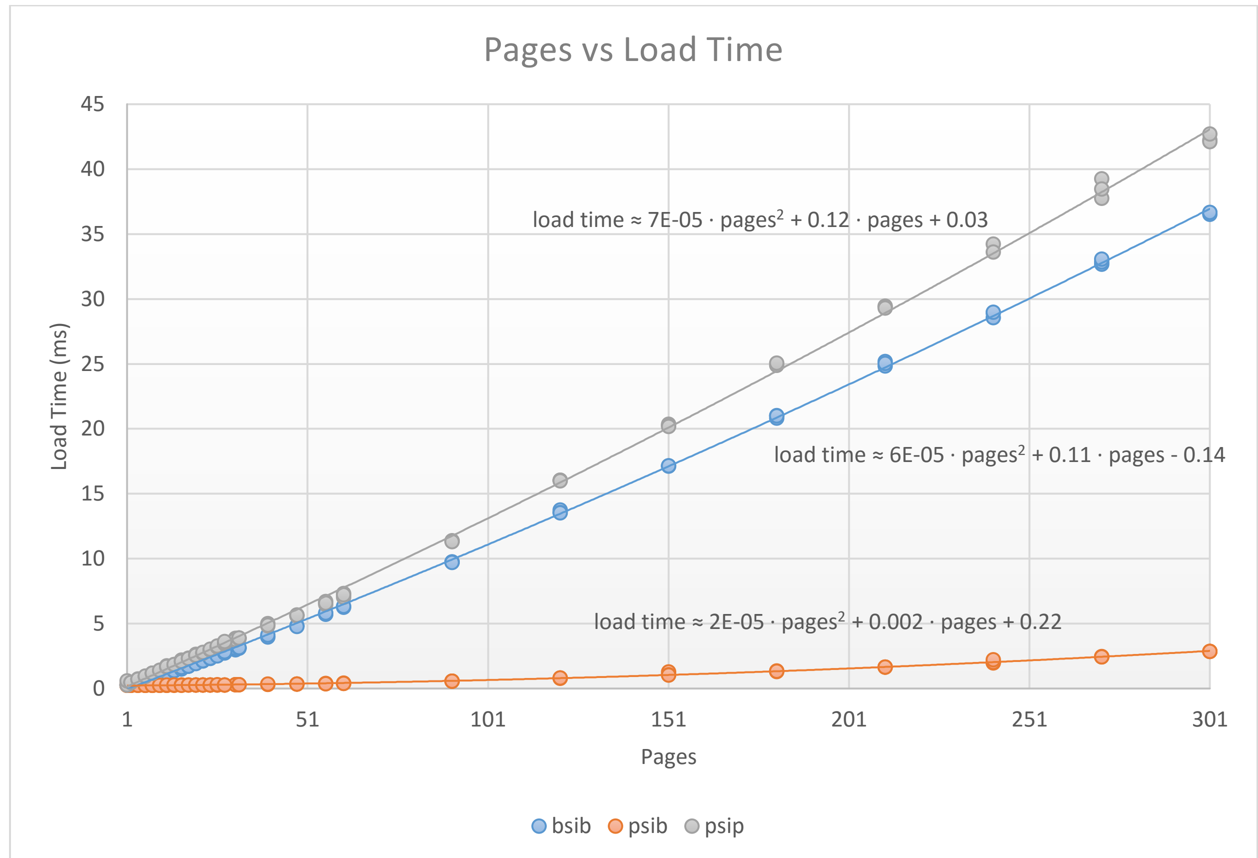


Figure 53 Pages per Secure Index vs Secure Index Load Time

Documents vs Load Time

EXPERIMENT DETAILS	
INPUT	documents (per corpus)
OUTPUT	corpus load time (milliseconds)
CONSTANTS	1 secret, 250 location uncertainty
	16 pages (per document)
	0.001 false positive rate
TESTBED	machine A

Figure 54 Experiment #17

[40] A way of storing small integers in fewer bytes. This is a slight cheat, since in general we attempted to ensure `sizeof (secure index data structure in memory)` is approximately the same as `size(secure index serialization on disk)`, but a `varint` on disk is converted into an unsigned integer once loaded into memory. Arguably, the cheat is justified since it is designed to mimic a more efficient representation.

As shown in Figure 55, the time to load a corpus consisting of medium-sized 16 page documents is 0.26 milliseconds per document for PSIB, 2.13 milliseconds for PSIP, and 1.55 milliseconds for BSIB.

In the previous experiment, we plotted page size vs load time. The line of best fit for $\text{PSIB load time} \approx 2E - 05 \cdot \text{pages}^2 + 0.002 \cdot \text{pages} + 0.22$. Plugging in $pages = 16$, we get a prediction of 0.26 milliseconds, which accurately matches the actual slope in Figure 55. The same is approximately the same for the other two secure indexes as well.

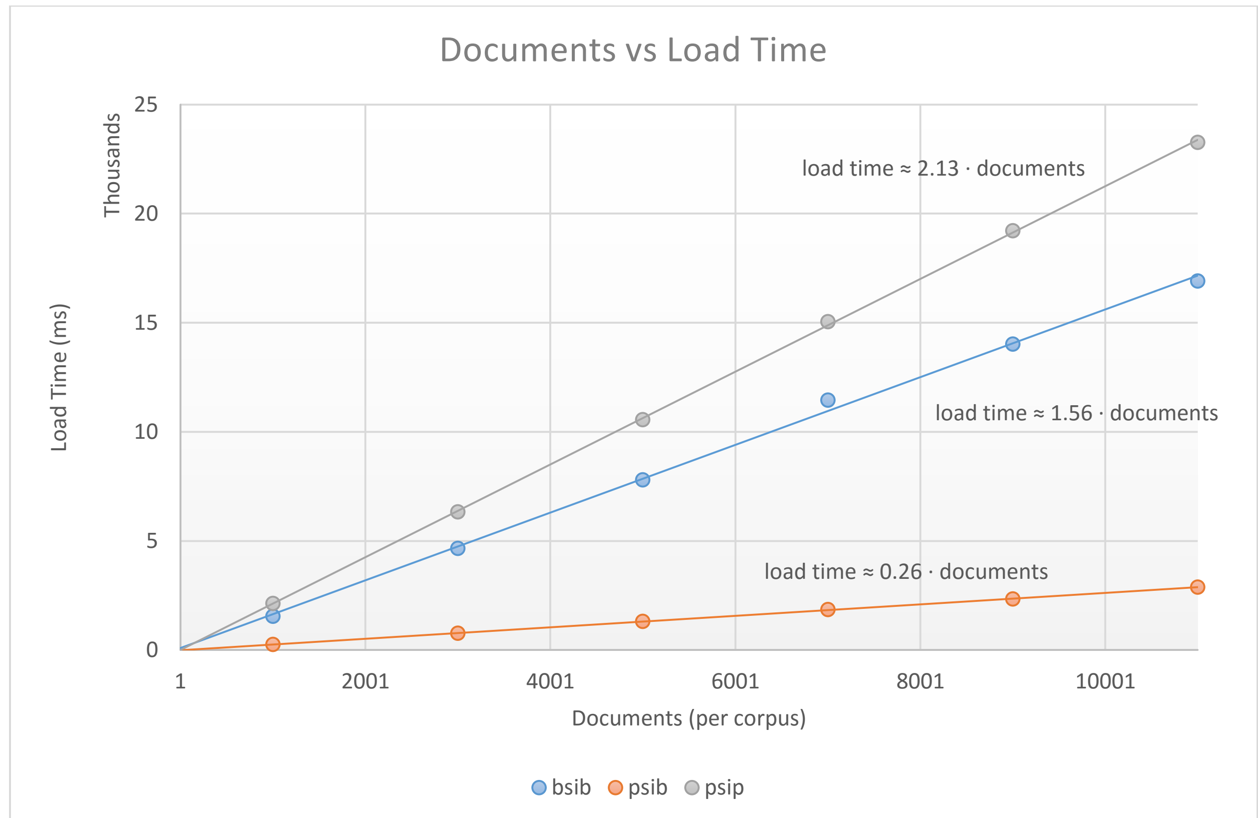


Figure 55 Documents per Corpus vs Corpus Load Time

Pages vs MinDist Lag Time*

EXPERIMENT DETAILS	
INPUT	pages (~250 words/page)
OUTPUT	MinDist* lag time (milliseconds)
CONSTANTS	1 secret, 0 obfuscations, 256 location uncertainty
	0.001 false positive rate
	2 terms/query, 1 or 2 words/term
TESTBED	machine A

Figure 56 Experiment #18

In this experiment, we are interested in seeing how page count affects MinDist* lag time. BSIB is unique among the secure indexes in that MinDist* lag time is linearly dependent upon page count; every additional page incurs ~ 0.0004 milliseconds, as shown in Figure 57.

For large documents (more specifically, for documents with a large number of block segments), BSIB performs poorly on this measure. This is the expected outcome. For a fixed location uncertainty, as

the page count increases the document must be segmented into more blocks and therefore, because every block is assigned a Bloom filter, more Bloom filters must be queried (i.e., more hash functions must be evaluated). Since each Bloom filter hash function evaluation requires a constant amount of time, all query lag times—MinDist* included—are dependent upon the number of hash functions that must be evaluated per document.

For a ~ 300 page document, the lag time is nearly 0.14 milliseconds. If the corpus consists of a million such documents, this operation would require nearly 140 seconds to complete. This is certainly impractical.

The PSI-based secure indexes, to a first approximation, take only a small constant amount of time with respect to page count. However, the constant—while small—will have scalability issue as corpus size grows to many thousands of documents. For instance, a corpus consisting of a million documents would require nearly ~20 seconds to complete. Even PSIP, the fastest secure index on this benchmark, would require ~7 seconds to complete.

None of them is below the one-second mark for the million-document example; one-second response times are often considered the maximum delay a typical user will tolerate, and the network latency time is not even being factored into this measure. Of course, secure indexes do not represent the typical use case—the services provided by secure indexes are not without cost—simply evaluating cryptographic hashes is computationally demanding, and we perform at least one of those per query term per document. Moreover, the slowed response times are far better than the alternative of downloading the entire corpus, decrypting the documents, and then conducting local searches on them.

There are a couple of immediately obvious ways to improve the computational efficiency of query operations like MinDist*. First, each secure index in the database re-hashes the hidden query's unigram and bigram terms with SHA256. While the re-hashing operation is desirable to ensure that a cryptographic hash of a term in one secure index looks nothing like the cryptographic hash of the same term in any other secure index, using SHA256 to perform the re-hashing is overkill; after all, the hidden query itself has already been transformed using SHA256.

According to our benchmarks, each evaluation of SHA256 takes ~0.0024 milliseconds on *machine A*. This is significant; a single SHA256 hash consumes one-third the total time taken, on average, to complete a PSIP MinDist* query (per secure index) consisting of two terms per query and one or two words per term. In fact, ~ 0.0024 milliseconds are required for each trapdoor in the query. While the secure indexes short-circuit processing queries where appropriate (in this experiment consisting of two terms per query, they can at most avoid processing one term per query per document), it is clear that significant savings could be realized by using an orders-of-magnitude faster non-cryptographic hash function without any loss in confidentiality.

Another way to speed up query processing is through parallel programming techniques. Each query can be independently queried so this is an *embarrassingly parallel* problem and performance scales linearly with core count. Given N cores, PSIP MinDist* query lag time would be $\sim 0.0078/N$ milliseconds. It could complete the MinDist* query operation against a million documents in less than a second given $N = 8$ cores.

BSIB would require around $N = 140$ cores to get under the one-second mark on a million documents consisting of ~ 300 pages per document. Using Bloom filters in the ways that have been

previously proposed does not seem to be practical at scale. When we include some optimizations, like replacing the slow cryptographic SHA256 re-hash with a fast hash and incorporate memorization or caching, it may still work at scale in practice.

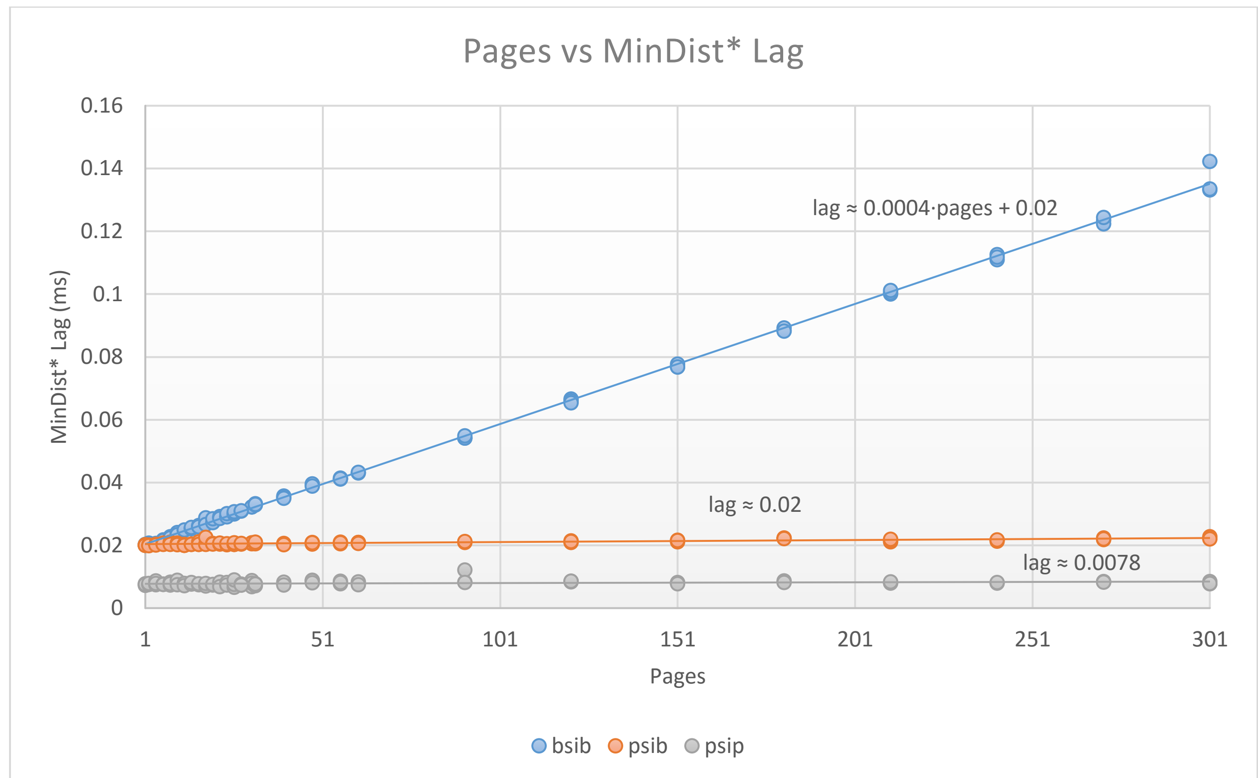


Figure 57 Pages per Secure Index vs MinDist Lag Time*

Location Uncertainty vs Location Error

EXPERIMENT DETAILS	
INPUT	location uncertainty
OUTPUT	average absolute location error
TESTBED	machine A

Figure 58 Experiment #19

MinDist* (see page 31) relies on a secure index's approximate location information to calculate minimum pairwise distances for query terms in the given document (or, in the case of PSIM, the approximate minimum pairwise distance information is directly encoded into it). The less uncertain the location is, all things else being equal, the more accurate MinDist* output will be (when compared to rank-ordered output of the canonical index with perfect information).

However, if the location information is too precise, a hypothetical adversary will have more success at inferring the contents of the document. Thus, the word positions must be uncertain—e.g., only reporting that a word falls within some range (block), as PSIB and BSIB do, scrambling the positions in some random way, as PSIP does, or directly encoding the minimum pairwise distances, as PSIM does.

In this experiment, we are interested in observing the expected location error for block-based secure indexes (PSIB and BSIB) and scrambled postings (PSIP)[41]—$centered_uniform$ and $centered_triangular$. Figure 59 clearly shows that the block-based secure indexes result in the greatest loss of accuracy. This was expected since terms are assigned uncertainty ranges—block ranges—that are not centered on their true positions. The other two (either of which may be used by PSI0P), $centered_uniform$ and $centered_triangular$, are named after the PDFs they sample their position offsets from: $centered_uniform$ uniformly samples an integral offset from $[-r, r]$, where $2r$ is equal to the location uncertainty, and $centered_triangular$ samples from the triangular distribution with a mean and mode equal to the true position and a base also of length $2r$. The triangular distribution has the least variance of the three, so naturally it has the least amount of error of the tree.[42]

These results are significant in light of our analysis on page 37, which suggests that block-based approaches as represented by PSIB and BSIB are significantly easier for a hypothetical adversary to compromise compared to centered approaches as represented by PSIP.

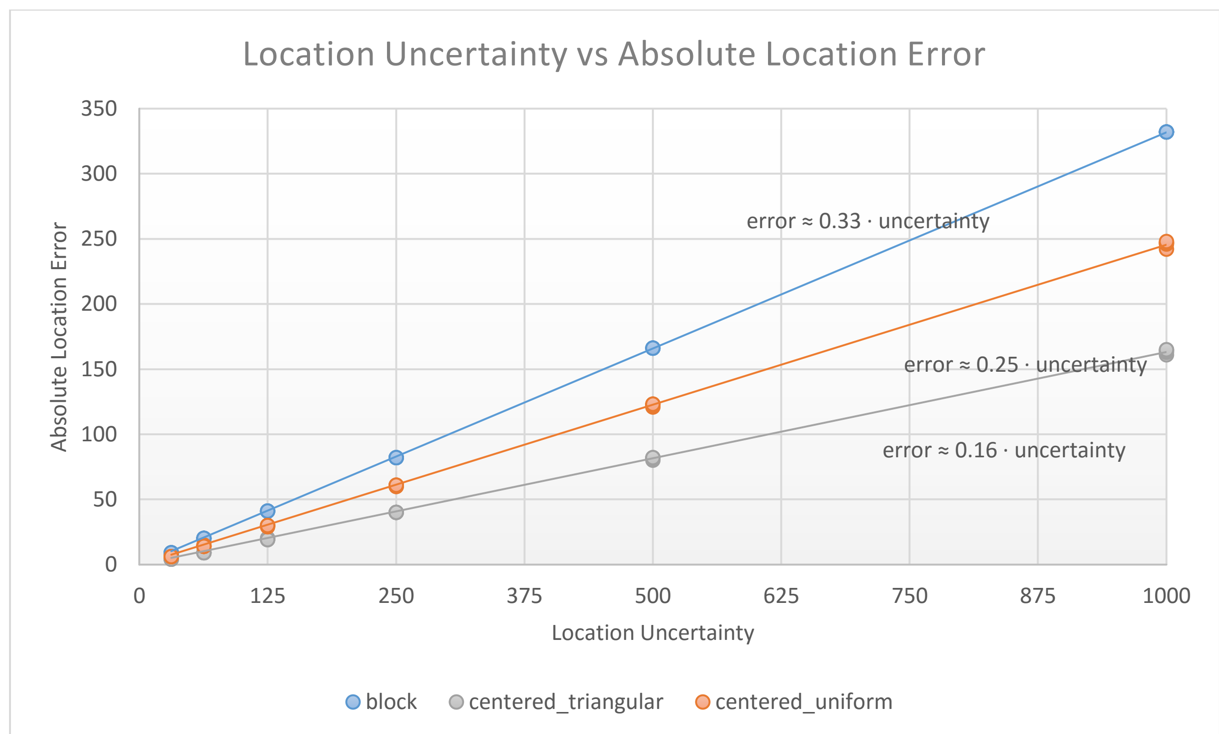


Figure 59 Location Uncertainty vs Absolute Location Error

[41] Note that PSIM is not simulated in this experiment since it can preserve perfect minimum pairwise distance information without the corresponding loss of confidentiality.
[42] PSIP may use any appropriate PDF, e.g., it could use a normal distribution with more or less variance to trade accuracy for leakage.

Location Uncertainty vs MinDist MAP*

EXPERIMENT DETAILS	
INPUT	location uncertainty
OUTPUT	MinDist* MAP
CONSTANTS	1 secret, 0 obfuscations
	1000 documents (per corpus)
	0.001 false positive rate
	3 terms/query, 1 or 2 words/term
TESTBED	machine A

Figure 60 Experiment #20

As shown in Figure 61 and Figure 62, as location uncertainty converges to 0, all of the secure indexes converge to the same MinDist* MAP, which is approximately a score of 0.95. However, as location uncertainty increases, PSIP quickly diverges from the other two.

Moreover, PSIB and BSIB do not scale well to large numbers of blocks (recall that location uncertainty is inversely proportional to the number of blocks in PSIB and BSIB). If the chosen parameters for MinDist* flatten out its curve, i.e., it is made to be less sensitive to small changes in location uncertainty (or the documents are reasonably small), PSIB and BSIB are competitive choices for MinDist*. Otherwise they are not well suited to it compared to PSIP. However, note that they may effectively enable other search criteria like Boolean proximity search.

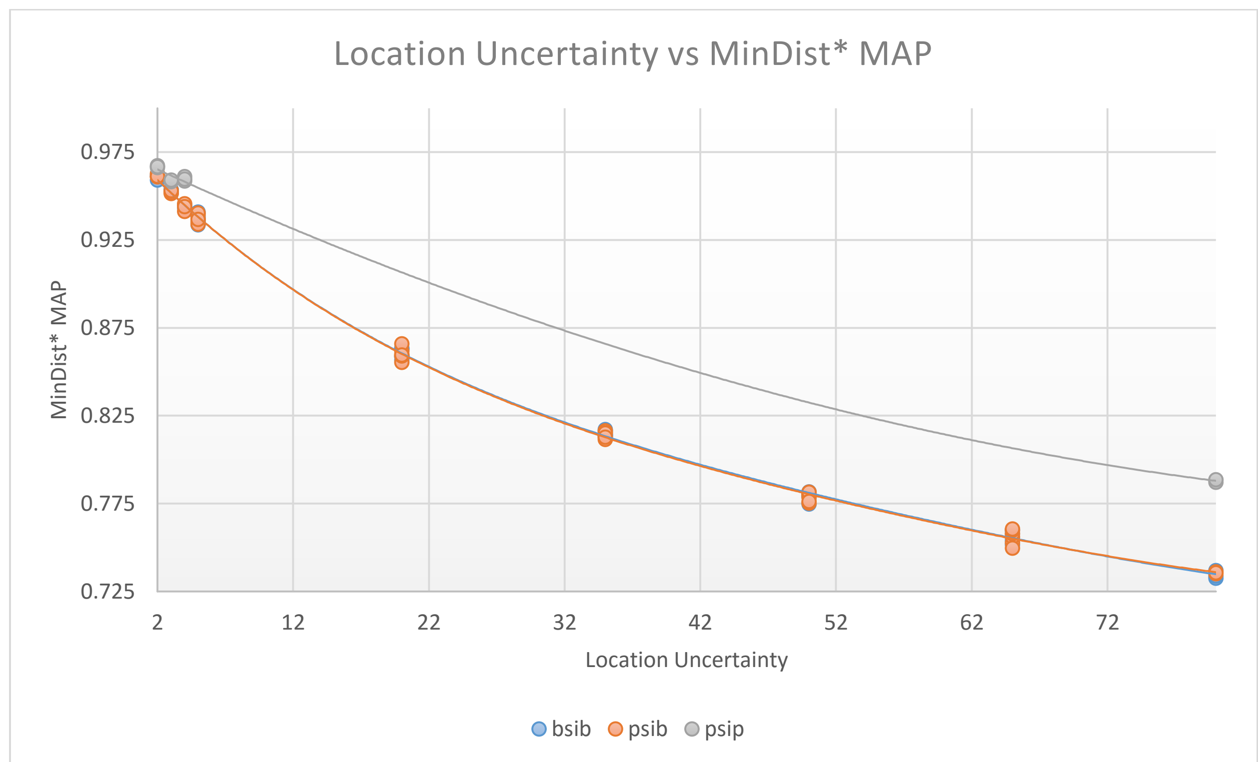


Figure 61 Location Uncertainty vs MinDist Mean Average Precision*

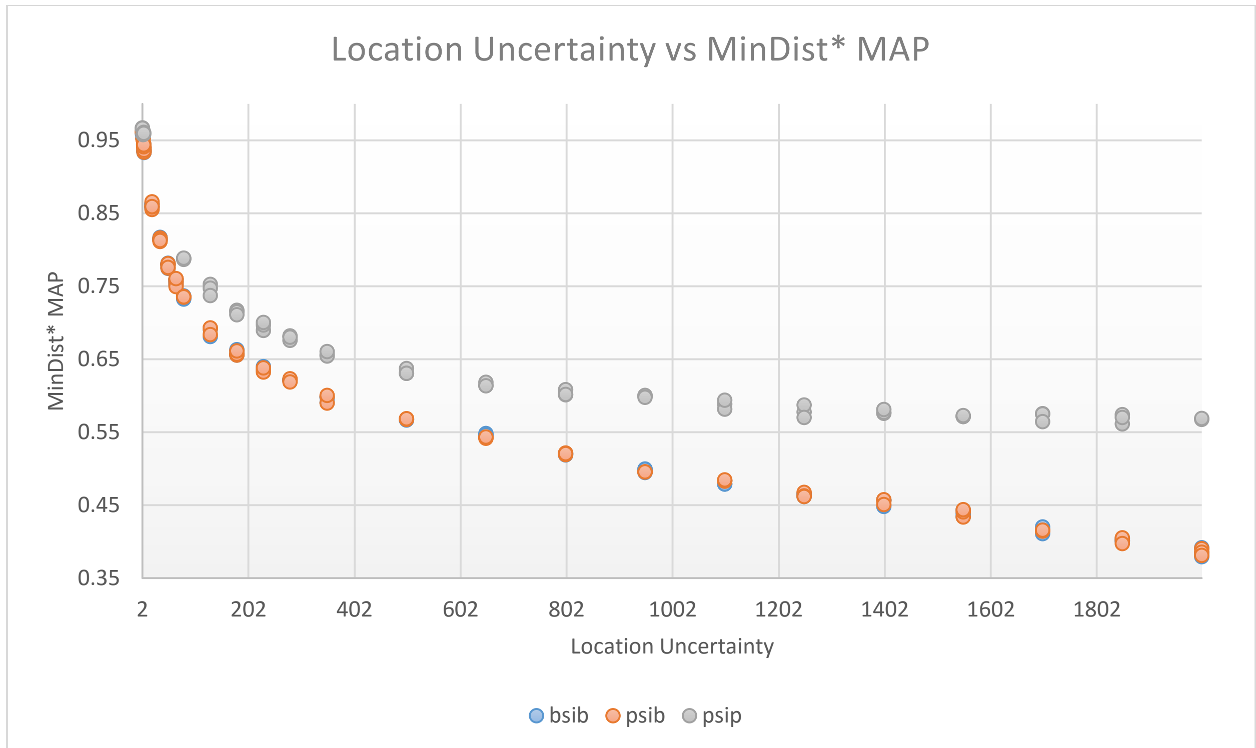


Figure 62 A Closer Look at Location Uncertainty vs MinDist MAP*

Pages vs BM25 Lag Time

EXPERIMENT DETAILS	
INPUT	pages (~250 words/page)
OUTPUT	BM25 lag time (milliseconds)
CONSTANTS	1 secret, 0 obfuscations, 256 location uncertainty
	0.001 false positive rate
	2 terms/query, 1 or 2 words/term
TESTBED	machine A

Figure 63 Experiment #21

In this experiment, we are interested in seeing how page count affects BM25 query lag time. Pages vs BM25 lag time shows a similar pattern to pages vs MinDist* lag time. However, note that BM25 is slower. The reason for this is related to a technicality in the implementation of the BM25 algorithm. Specifically, the implementation queries secure indexes twice for each query term; once for calculating the number of documents which contain a given query term, and once for calculating the frequency of the given query term per document.

The implementation of the BM25 algorithm can be streamlined to only require a single query per document. Moreover, many of these computations can be memoized. In practice, this can be expected to save a significant amount of work.

As shown by Figure 64 and Figure 65, BM25 lag time for PSIB and PSIP are, to a first approximation, independent of page count.

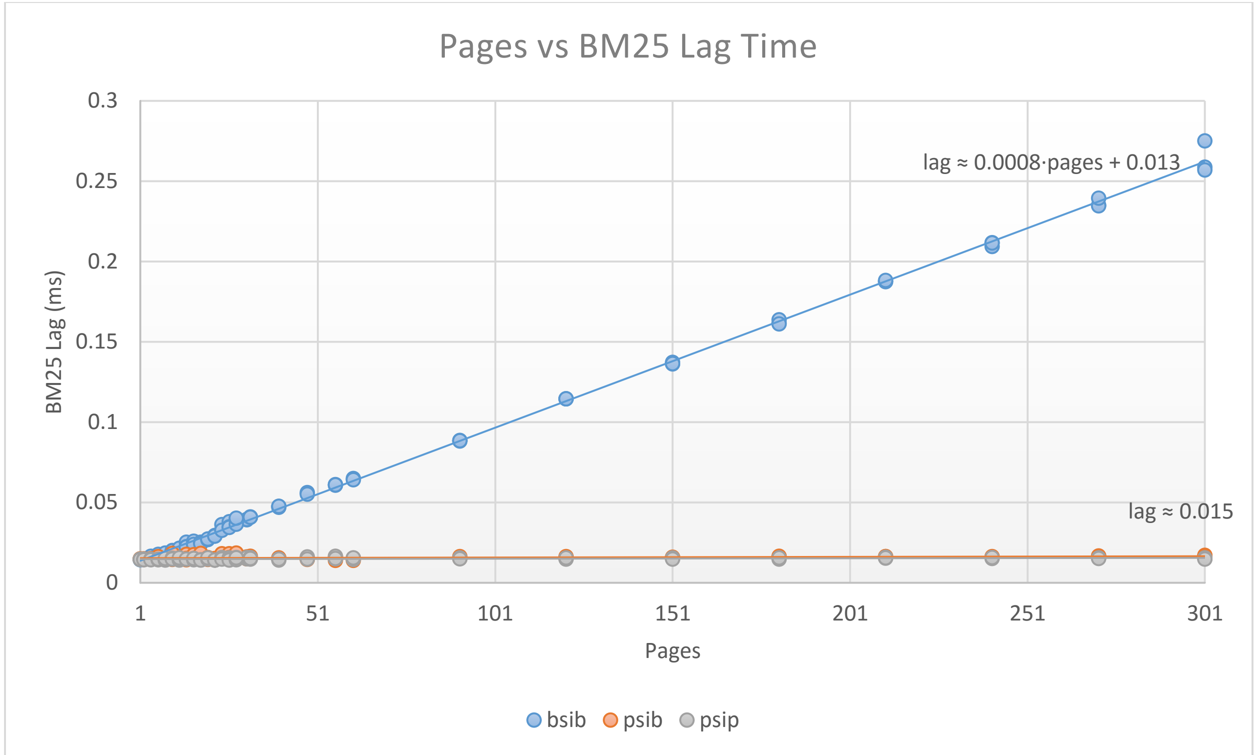


Figure 64 Pages per Secure Index vs BM25 Lag Time

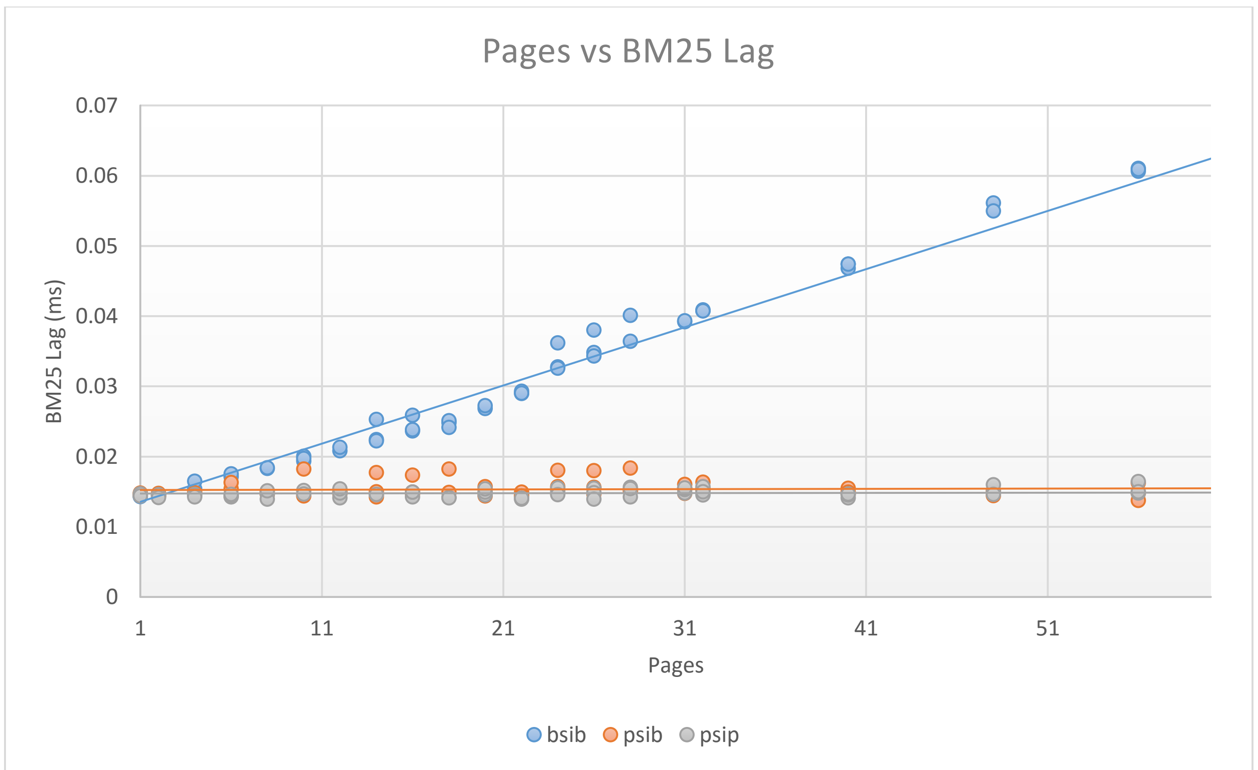


Figure 65 A Closer Look at Pages per Secure Index vs BM25 Lag Time

Location Uncertainty vs BM25 MAP

EXPERIMENT DETAILS	
INPUT	location uncertainty
OUTPUT	BM25 MAP
CONSTANTS	1 secret, 0 obfuscations, 0.001 false positive rate 3 terms/query (Figure 67), 2 terms/query (Figure 68) 1 or 2 words/term 16 pages (per document), 1000 documents (per corpus)
TESTBED	machine A

Figure 66 Experiment #22

In this experiment, we are interested in seeing how location uncertainty effects BM25 MAP. In Figure 67 and Figure 68, PSIP and PSIF track each other perfectly, as do PSIB and BSIB. PSIP and PSIF preserve, if desired, perfect frequency information for unigrams and bigrams (although false positives on negative examples are still possible), and this is independent of location uncertainty in PSIP (and PSIF does not store location information).

PSIB and BSIB approximate a term's frequency by counting how many of the blocks it appears in. The more blocks (the lesser the location uncertainty), the more accurately it approximates the true frequency. Indeed, in Figure 67, we see that as location uncertainty converges to 2, all of the secure indexes converge to the same BM25 score.

In addition, notice that compared to Figure 67, Figure 68 is less accurate for a given location uncertainty for all of the secure index types. The reason for this has to do with the fact that Figure 67 has more terms per query. It seems to be the case that the more terms per query the less sensitive BM25 is to frequency approximation errors.

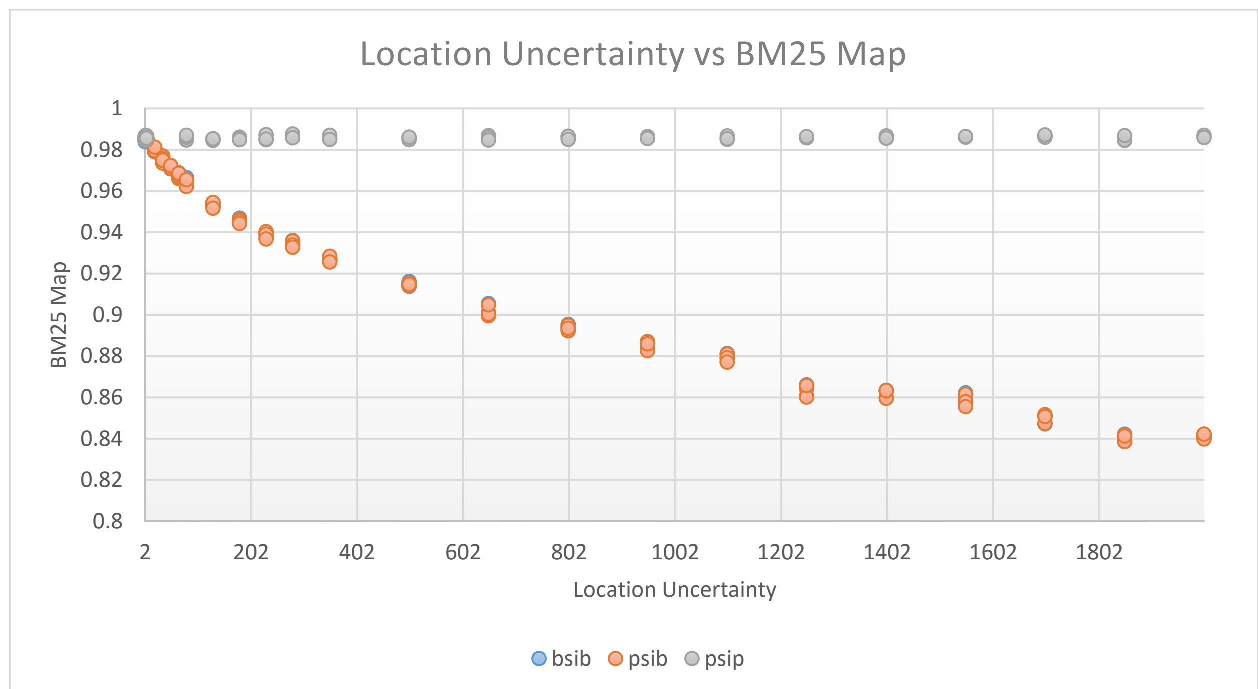


Figure 67 Location Uncertainty vs BM25 MAP with 2 Words/Term, 3 Terms/Query

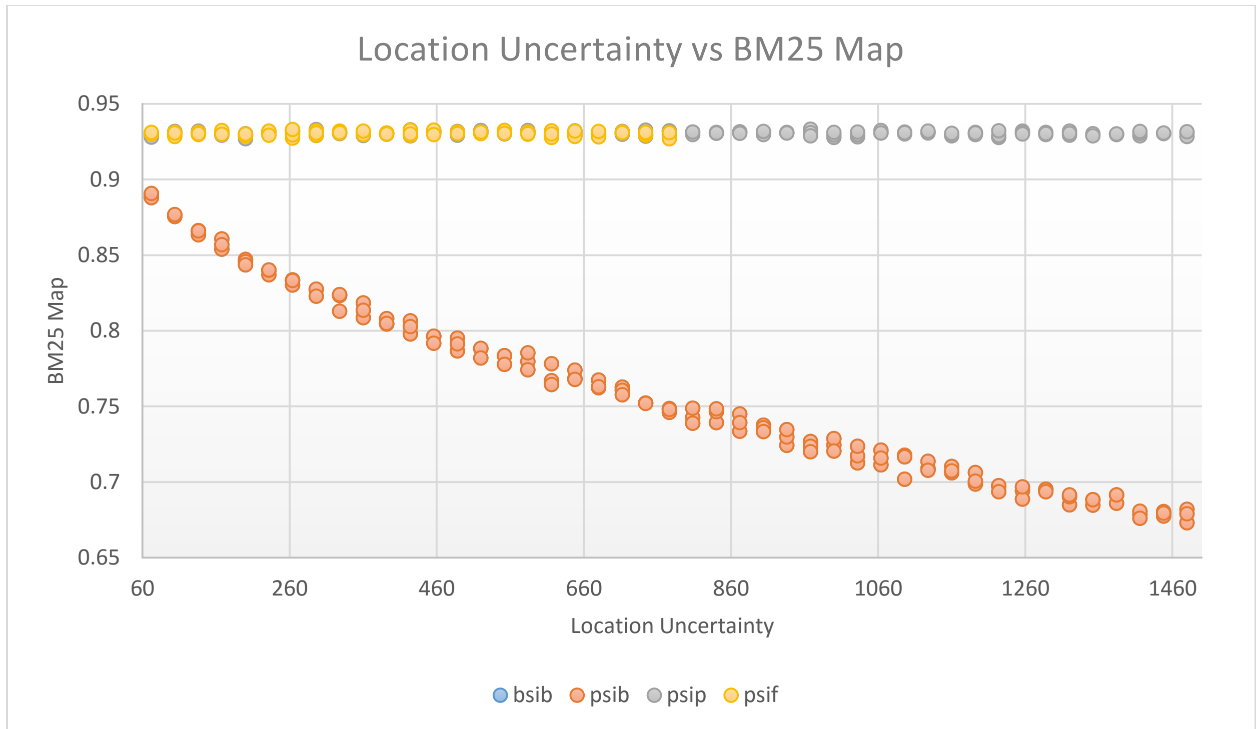


Figure 68 Location Uncertainty vs BM25 MAP with 2 Words/Term, 2 Terms/Query

Pages vs Boolean Lag Time

EXPERIMENT DETAILS	
INPUT	pages (~250 words/page)
OUTPUT	Boolean query lag time (milliseconds)
CONSTANTS	1 secret, 0 obfuscations
	256 location uncertainty
	0.001 false positive rate
	2 terms/query, 1 or 2 words/term
TESTBED	machine A

Figure 69 Experiment #23

As with every other output related to lag time, BSIB performs comparatively poorly—as expected. Note that Boolean queries do not rank documents; a document is either relevant to the query (in this case, for a document to be relevant it must contain all of the terms in the query—a Boolean AND operation) or it is non-relevant. This is the quickest kind of query.

Indeed, the time required to perform the excessive SHA256 re-hashing operation takes up the most significant portion of time. PSIB, PSIP, and PSIF (PSIF is not shown but it tracks PSIP) can complete this operation in approximately 0.003 milliseconds, at least 0.0024 milliseconds of which is consumed by computing unnecessary SHA256 re-hashes. In other words, this is a ~500 nanosecond operation. This should allow a corpus of two million secure indexes to be searched by queries of this form in a second (not including the round-trip network delay).

With some of the other proposed performance enhancers, like caching and parallel computing, the simple Boolean query operation is extremely scalable. For documents of a typical length, e.g., less than 50 pages, even BSIB performs acceptably.

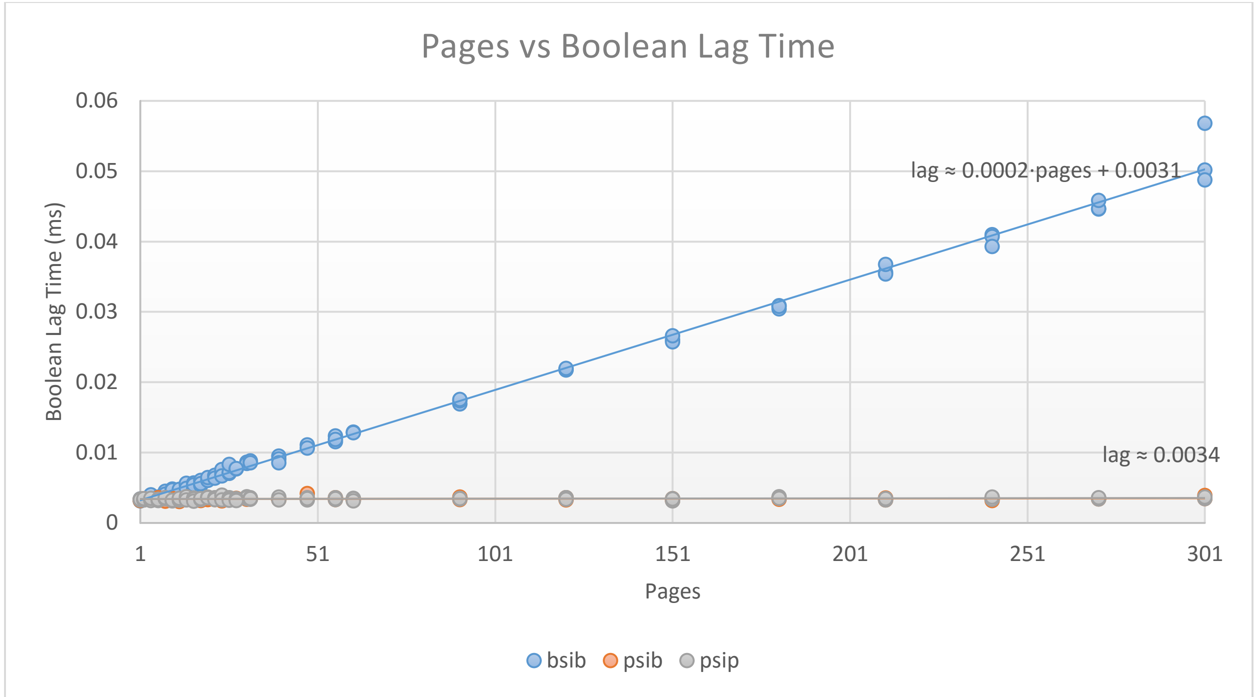


Figure 70 Pages per Secure Index vs Boolean Search Lag Time

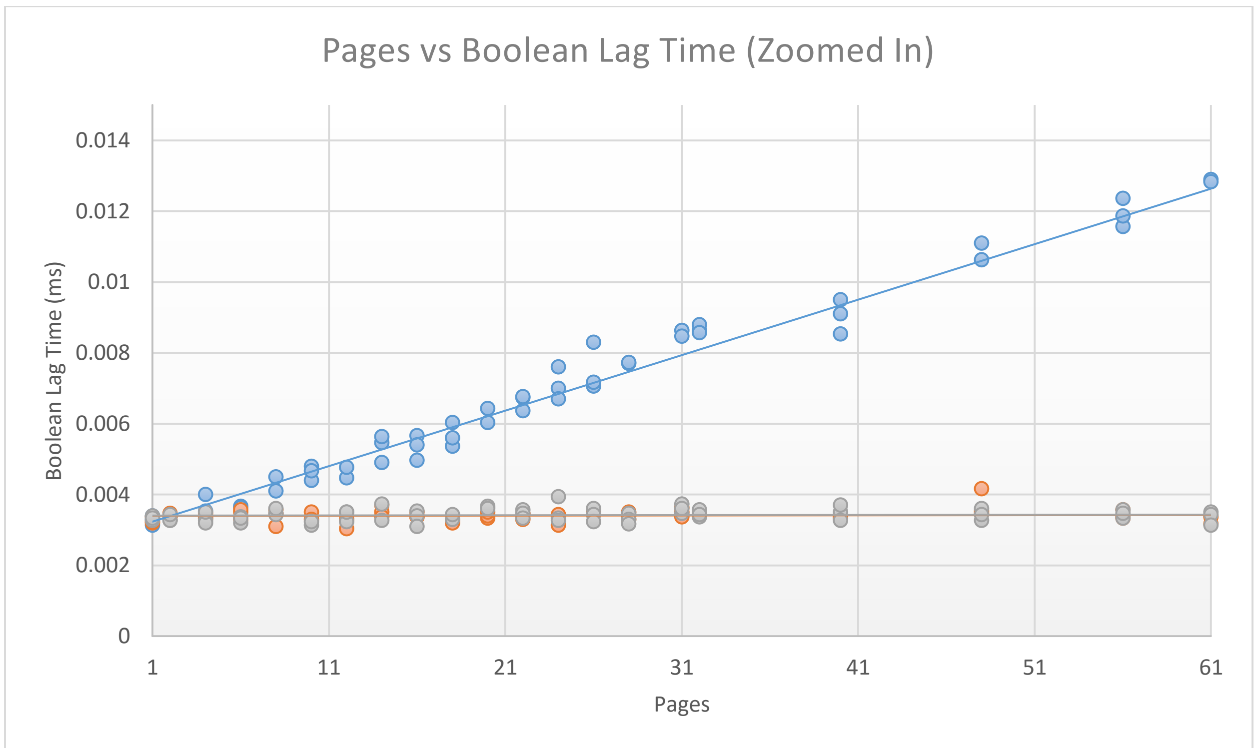


Figure 71 A Closer Look at Pages per Secure Index vs Boolean Search Lag Time

Secure Index Poisoning: Junk Terms vs Compression Ratio, BM25 MAP

EXPERIMENT DETAILS	
INPUT	Junk term percentage (proportion of fake terms in secure index)
OUTPUT	BM25 MAP, compression ratio
CONSTANTS	16 pages, 1000 documents (per corpus)
	1 secret, 0 obfuscations
	256 location uncertainty
	0.001 false positive rate
	2 terms/query (only relevant for BM25 MAP)
	1 or 2 words/term (only relevant for BM25 MAP)
TESTBED	machine A

Figure 72 Experiment #24

As expected, BM25 is, to a first approximation, independent of junk term percentage (see page 40). Also as expected, the compression ratio does depend on junk term percentage. PSIB, PSIP, and PSIF grow non-linearly as the junk percentage increases, but at worst (when there are 50% junk terms) they are only approximately double the original size. We also see that PSIB and BSIB eventually cross over at around the 40% mark. Before 40%, PSIB has the advantage.

According to these results, we may choose to significantly poison the secure indexes by adding fake terms to mitigate the attacks (see page 37) with very little loss in accuracy and a moderate cost to size.

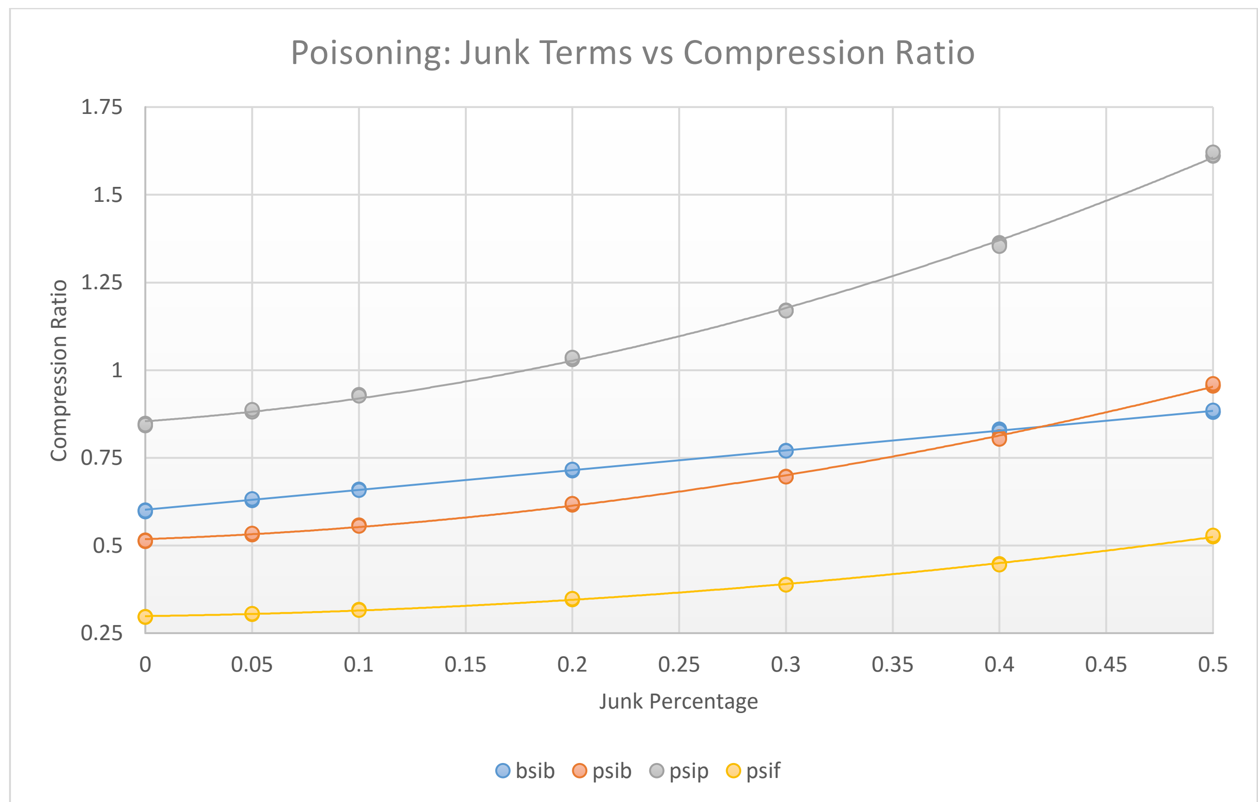


Figure 73 Percentage of Junk Terms vs Secure Index Compression Ratio

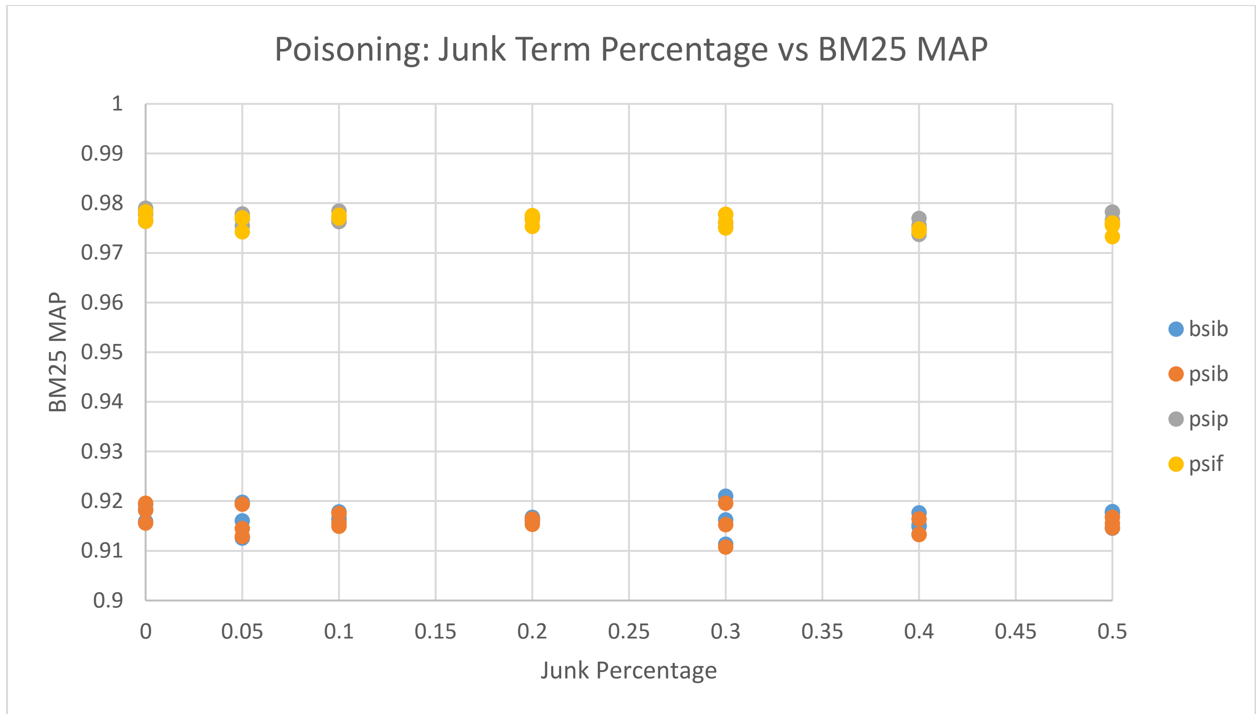


Figure 74 Percentage of Junk Terms vs BM25 MAP

Secure Index Poisoning: Frequency Percent Error vs Compression Ratio

EXPERIMENT DETAILS	
INPUT	Relative frequency percent error and junk percentage (same value)
OUTPUT	BM25 MAP
CONSTANTS	16 pages, 1000 documents (per corpus)
	1 secret, 0 obfuscations
	256 location uncertainty
	0.001 false positive rate
	1 or 2 words/term
TESTBED	machine A

Figure 75 Experiment #25

The frequency error for PSIB and BSIB is implicitly dependent on location uncertainty and cannot be controlled explicitly. Thus, we do not include their outputs in this experiment.

In this experiment, we decided to change both the relative frequency error and the junk term percentage simultaneously for PSIP and PSIF. For PSIF, approximate frequency $\approx$ true frequency $\pm$ $UNIF(0,\text{relative frequecy error}) \times$ true frequency, but for PSIP approximate frequency $\approx$ true frequency $+ UNIF(0,\text{relative frequency error}) \times$ true frequency. This explains why PSIP performs consistently better than PSIF, as it has a smaller relative error range. This was done due to time constraints; ideally, PSIP would have the same relative frequency error formula as PSIF, and it is expected once that was implemented in PSIP it would have the same response as PSIF on this experiment.

As shown in Figure 76, they both perform admirably. Even with a relative frequency error and junk percentage of 50%, they both do better than 90% when using 2 terms/query. When we decrease the terms per query to 1, they perform expectedly worse—indeed, at 50% error, PSIF reaches nearly

80% mean average precision. We may choose to significantly poison these secure indexes to mitigate the adversary's attack with only a modest loss in accuracy.

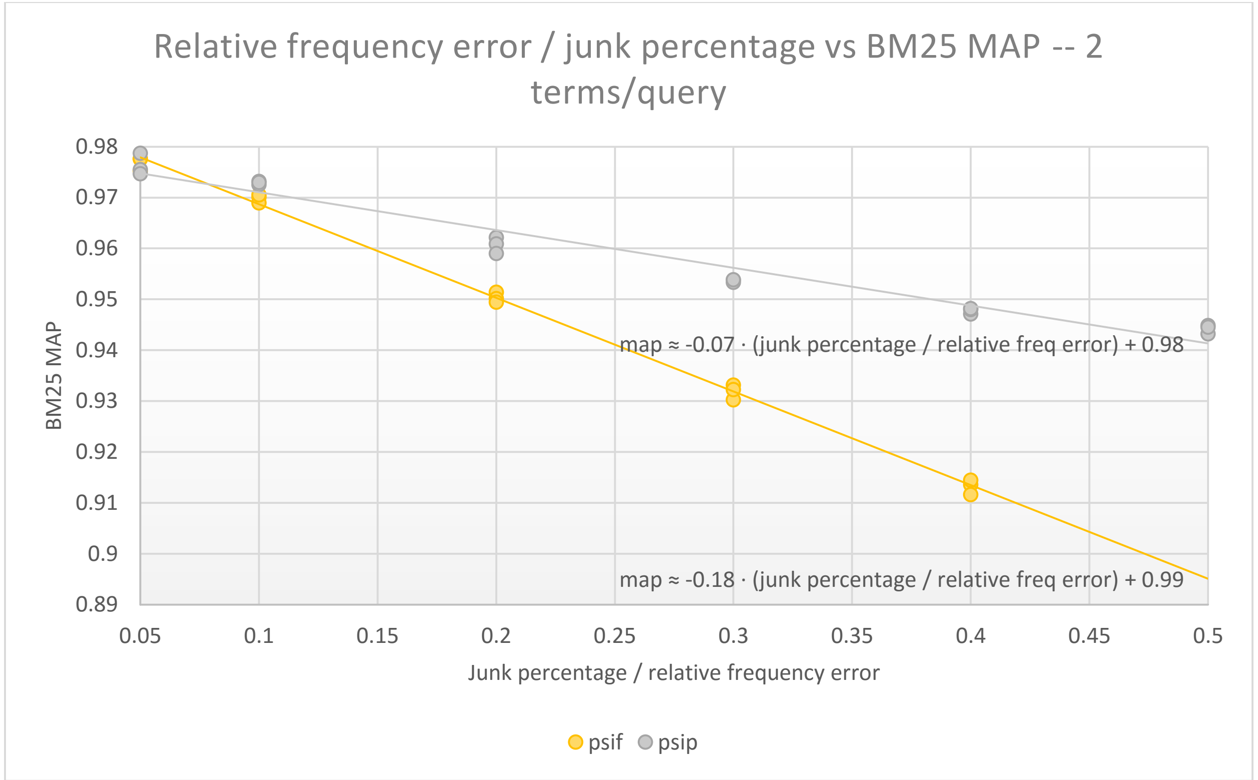


Figure 76 Relative Frequency Error vs BM25 MAP with 2 Terms/Query

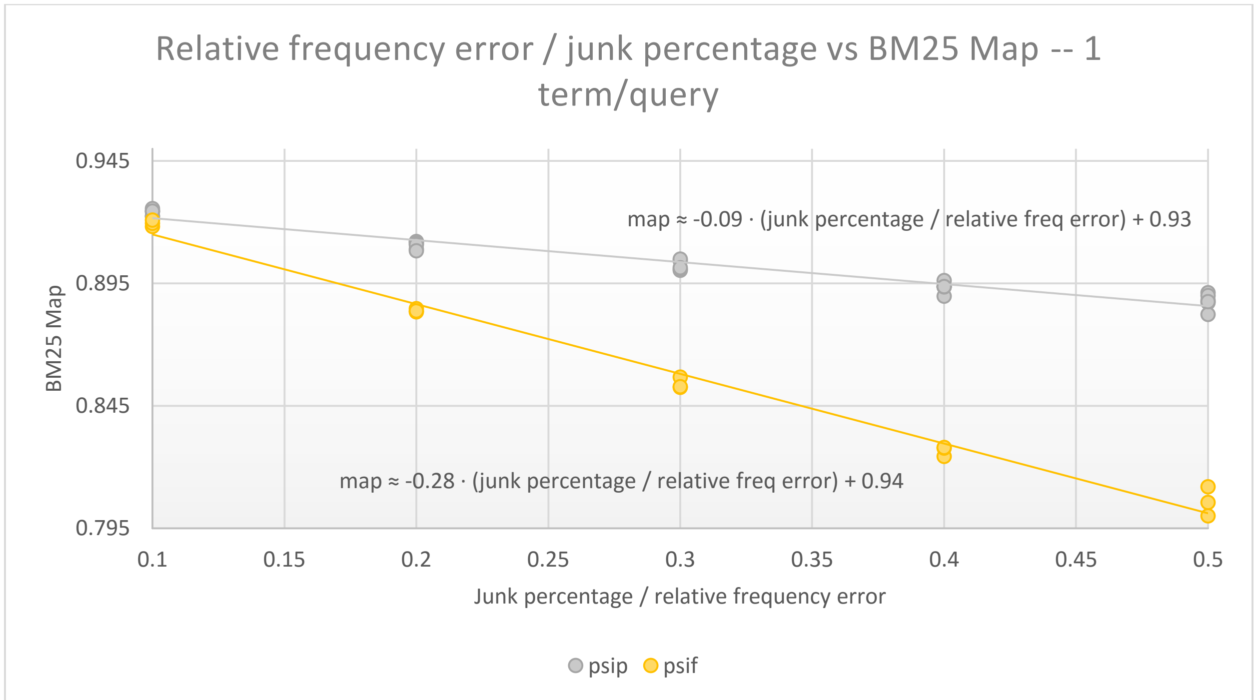


Figure 77 Relative Frequency Error vs BM25 MAP with 1 Term/Query

Compression Ratio (Secure Index Size to Document Size) vs MinDist MAP*

EXPERIMENT DETAILS	
INPUT	compression ratio (ratio of secure index size to document size)
OUTPUT	MinDist* MAP
CONSTANTS	1 secret, 0 obfuscations 0.001 false positive rate 3 terms/query, 1 or 2 words/term 1000 documents (per corpus)
TESTBED	machine A

Figure 78 Experiment #26

In this experiment, we are interested in seeing how the ratio of the secure index size to the original document size affects MinDist* MAP accuracy.

PSIP has the highest unconditional MinDist* MAP score. However, PSIB and BSIB have much better compression ratios. In fact, PSIP has a nearly constant ratio (across a range of document sizes)—the only way to change its size is by changing the false positive rate or by poisoning it. Depending on how the poisoning is done, it can be made smaller (e.g., replacing multiple positions with a single mean position in a postings list) or larger (adding positions in a postings list or adding fake terms). This is both a positive and a negative, as PSIP is nearly (in these examples) always the same fraction of the original document's size but is consistently the highest performer on MAP accuracy and lag time.

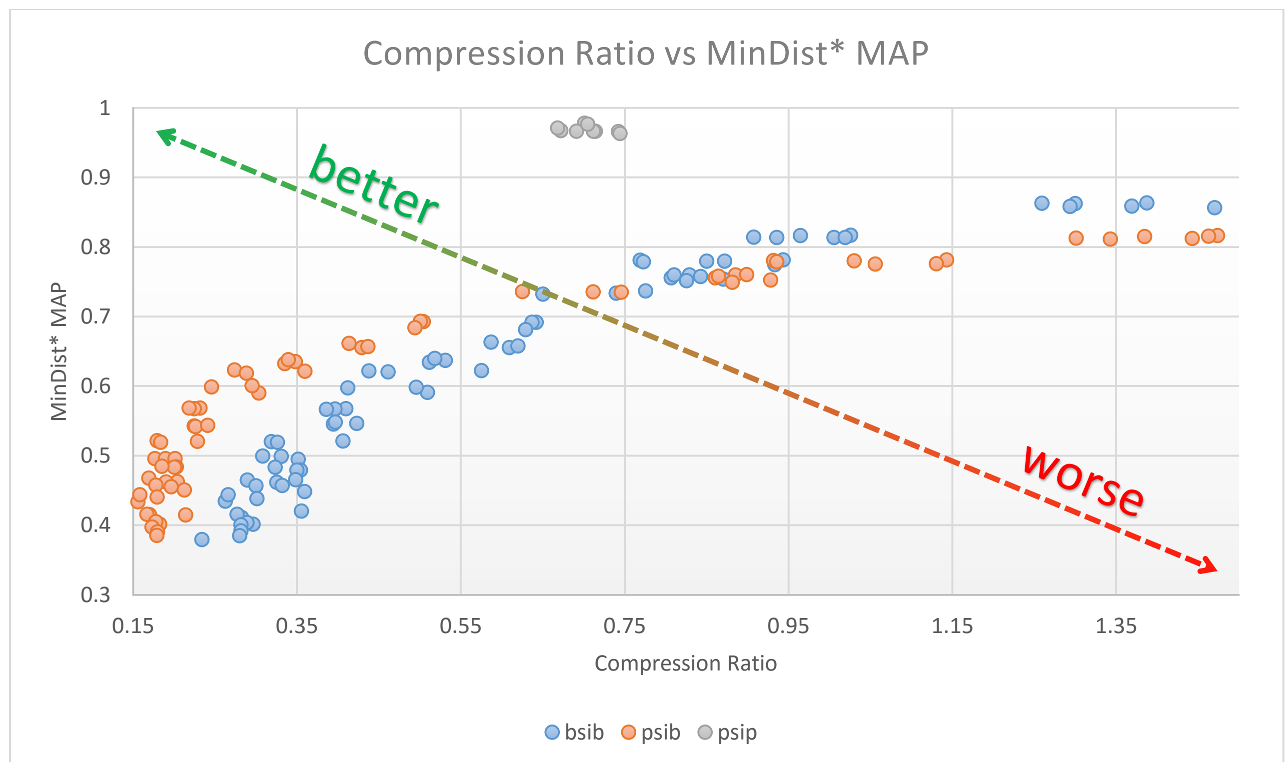


Figure 79 Compression Ratio vs MinDist MAP*

Words/Term vs Precision and Recall

EXPERIMENT DETAILS	
INPUT	words/term
OUTPUT	precision and recall
CONSTANTS	1 secret, 0 obfuscations
	0.001 false positive rate
	1 term/query
	16 pages, 500 documents (per corpus)
TESTBED	machine A

Figure 80 Experiment #27

For a term to be a *false positive*, one of the following conditions must be true (see page 23):

(1) If the query term is a keyword, then its unigram is not in the document but due to the false positive rate of the secure index it is a positive hit.
(2) If the query term is a phrase, and all of its bigrams are present in the document, then they are in the wrong order. This is a false positive caused by the *biword* model.
(3) If the query term is a phrase, and not all of its bigrams are present in the document, then all of these non-present bigrams must be false positives.

Theoretically, a *false negative* can only occur if:

(4) The query term is an n-gram, $n > 2$, and the secure index is a PSIB or a BSIB. If this is the case, a phrase may exist in the original document, but due to the way PSIB and BSIB filter out false positives, they may also filter out true positives on occasion since we only count a phrase as a hit if all of its bigrams are in a single block.

In this test, we are interested in observing how precision and recall behave in practice in light of the above points. Each query consists of one term, from one to six words per term. As can be seen in Figure 81, as expected recall decreases the as the words/term increases for PSIB and BSIB. Additionally, precision increases when words/term increases. Point (3), above, explains why. It is less probable to get k false positives than $k-1$ false positives. Mathematically:

$$P[\text{false positive}|\text{n-gram}] \sim \varepsilon^k, k < n$$

where ε is the false positive rate on bigrams (and unigrams) and k is the number of bigrams in the n-gram which are non-existent in the n-gram phrase with a maximum of $n-1$ bigrams.

However, unexpectedly we do see some false negatives when the term consists of only one or two words—a single unigram or a single bigram. Theoretically, this should not happen. It is extremely minor, i.e., recall for this situation is over 99.95% accurate but we were expecting 100%. This warrants further investigation.

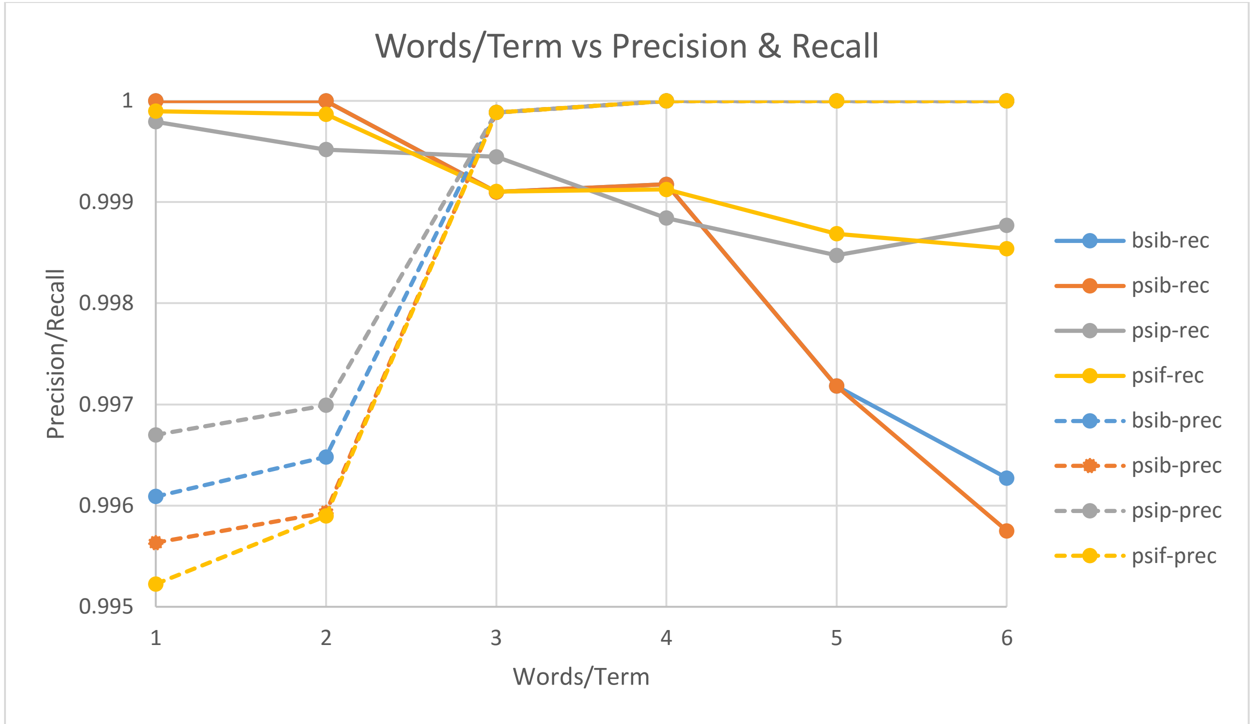


Figure 81 Words per Term vs Precision and Recall

CHAPTER V
EXTENSIONS

SET-THEORETIC QUERIES

Set-theoretic queries are a simple extension of Boolean queries. Let the results (list of references) returned from Boolean search be sets, then the intersection (AND), union (OR), and complement (NOT) of them may be taken, e.g., $\text{result set for } q_1 \text{ AND result set for } q_2$.

All that a secure index needs to do in order to support set-theoretic query operators is to implement the $\text{contains(ht: HiddenTerm): Boolean}$ interface. Thus, for each atomic query in a compound query, it will use this interface to retrieve the query's result set and then apply set-theoretic operations on the result sets to implement AND, OR, and NOT.

For instance, to implement $\text{NOT}(q)$, find all documents in the corpus that are relevant to the atomic query q and then take the complement of this result set, i.e., only include a document (from the corpus) in the complement set if it is not in the result set.

Furthermore, all other set-theoretic operators, like set difference, can be expressed in terms of AND, OR, and NOT[43], e.g., $\text{difference}(A, B) = A \text{ AND NOT}(B)$. Note that to support arbitrary set-theoretic operations, the language should be defined by a recursive grammar. This will allow for applying operators to nested results. For a company like Google, this is not necessarily a practical option since they must support millions of queries per second on billions of documents, but it may be a practical option for an *Encrypted Search* cloud server.

Set-theoretic query grammar. An example of a recursive grammar, expressed in BNF notation, for set-theoretic queries is:

<table>
<tr><td><query></td><td>::=</td><td><atomic_query> | (<query>) |</td></tr>
<tr><td></td><td>::=</td><td>NOT <query> |</td></tr>
<tr><td></td><td>::=</td><td><query> <binary_op> <query></td></tr>
<tr><td><atomic_query></td><td>::=</td><td><term> | <term> <atomic_query></td></tr>
<tr><td><term></td><td>::=</td><td><exact_phrase> | <keyword></td></tr>
<tr><td><binary_op></td><td>::=</td><td>AND | OR | ...</td></tr>
<tr><td><exact_phrase></td><td>::=</td><td>"<keywords>"</td></tr>
<tr><td><keywords></td><td>::=</td><td><keyword> | <keyword> <keywords></td></tr>
<tr><td><keyword></td><td>::=</td><td><alphanumeric><keyword> | <alphanumeric></td></tr>
<tr><td><alphanumeric></td><td>::=</td><td>a|b|...|z|0|1|...|9</td></tr>
</table>

Table 7 BNF Set-Theoretic Query Grammar

FUZZY SET-THEORETIC SEARCH

Fuzzy set-theoretic queries are an extension of classical (crisp) set-theoretic queries. Instead of operating on Boolean values, they operate on degree of membership values; a degree of membership value represents the degree (in the range $[0, 1]$) to which something is a member of a

[43] Technically, only AND and NOT (or OR and NOT) are needed, but for computational efficiency both AND and OR should be efficiently supported such that they may be short-circuited as early as possible. Indeed, this may justify implementing additional operators without reducing them to combinations of AND, OR, and NOT.

set. At one extreme, a value equal to 1 is equivalent to true—e.g., a document is completely relevant to a query. At the other extreme, a value equal to 0 is equivalent to false—e.g., a document is completely irrelevant to a query.

Fuzzy operator equivalents to NOT, OR, and AND are:

$$\mathrm{NOT}(q) = 1 - q$$

$$\mathrm{AND}(q_1, q_2) = \mathrm{MIN}(q_1, q_2)$$

$$\mathrm{OR}(q_1, q_2) = \mathrm{MAX}(q_1, q_2)$$

Fuzzy set-theoretic queries may use the normalized output of any scoring algorithm or heuristic that does not simply output a binary score[44]. The non-binary output from BM25 and MinDist* are obvious candidates for this. However, before they may be used, their output must be normalized such that the maximum value is 1 and the minimum value is 0.

For instance, unnormalized MinDist* is $MinDistScore(Q) = \ln\left(\alpha + \gamma \cdot \exp\left\{-\frac{\beta s}{|Q'|^{\theta}}\right\}\right)$. The minimum value for $MinDistScore$ is $\lim_{s\to\infty} MinDistScore(Q) = \ln(\alpha)$ and the maximum value is $MinDistScore(Q; s = 0) = \ln(\alpha + \gamma)$. Thus, normalized MinDistScore is $MinDistNorm(d, Q) = \frac{MinDistScore(d,Q)-\ln(\alpha)}{\ln(\alpha+\gamma)-\ln(\alpha)} = \frac{MinDistScore(d,Q)-\ln(\alpha)}{\ln\left(\frac{\alpha+\gamma}{\alpha}\right)}$. $MinDistNorm$ may be used in fuzzy set theoretic queries. For example:

$$\text{fuzzy degree of membership for doc } d = (MinDistNorm(d, Q_1)\ OR\ MinDistNorm)\ \mathrm{AND}\ \mathrm{NOT}\big(\mathrm{MinDistNorm}(d, Q_2)\big),$$

where Q_1, Q_2, and Q_3 can be any query.

A similar normalization can be done for $BM25Score$. Moreover, it may be reasonable to allow the metrics to be specifically requested by the user, e.g., let $MinDistNorm(d, Q) \equiv close(d, Q)$, $BM25Norm(d, Q) \equiv term_importance(d, Q)$, and the linear combination $\alpha_1 \cdot MinDistScore(d, Q) + \alpha_2 \cdot BM25Score(d, Q) \equiv importance(d, Q)$. Thus, for example:

$$\text{fuzzy degree of membership for doc d} = \Big(\mathrm{close}(d, Q_1)\ \mathrm{OR}\ \mathrm{NOT}\big(\mathrm{importance}(d, Q_2)\big)\Big)\ \mathrm{OR}\ \mathrm{term_importance}(d, Q_3)$$

Typically, some defuzzification procedure is used to produce some actual result from the degree of membership values (like a steering direction in a fuzzy control system). In the context of information retrieval, one option is to define an operator, $true(q) = q > K$, where q represents the degree to which the confidential document is relevant to the fuzzy query. Thus, if $q > K$, the corresponding document is considered to be relevant and will be returned in the result set. However, defuzzification seems unnecessary and even undesirable; rather, q can serve as the document's score for rank-ordering. No defuzzification is warranted.

We can also apply hedges to degree of membership values. Hedges modify the degree of membership value in a way that conforms to common intuition, e.g., a proposition can be *somewhat*

---

[44] Fuzzy set-theoretic queries reduce to classical set-theoretic queries if the scoring algorithm only outputs binary scores.

true but may not be *very* true. Indeed, *somewhat* and *very* are two examples of hedge functions. Let $\text{very}(q) = q^2$ and $\text{somewhat}(q) = q^{\frac{1}{2}}$.

On the one hand, observe that $\text{very}(q)$ is lower than q if $q \in (0,1)$ and $\text{very}(q) = q$ if $q = 0$ or $q = 1$. Also, note that $\frac{d}{dq}\text{very}(q) \propto q$, and thus the marginal value of $\text{very}(q)$ increases as q goes from 0 to 1, i.e., as q approaches 1, $\text{very}(q)$ increases at a faster and faster rate and converges to $\text{identity}(q)$ at $q = 1$. On the other hand, observe that $\text{somewhat}(q)$ is higher than q if $q \in (0,1)$ and $\text{somewhat}(q) = q$ if $q = 0$ or $q = 1$. Also, note that $\frac{d}{dq}\text{somewhat}(q) \propto q^{-\frac{1}{2}}$, and thus the marginal value of $\text{somewhat}(q)$ decreases as q goes from 0 to 1, i.e., as q approaches 1, $\text{somewhat}(q)$ increases at a slower and slower rate and converges to $\text{identity}(q)$ at $q = 1$. In other words, you quickly reach a state of diminishing returns, e.g., a "medium" q is not much less *true* than a "large" q.

An example of a fuzzy query using these two hedges is:

$$\text{fuzzy degree of membership for doc } d \\ = \text{very}\Big(\text{close}(d, Q_1) \text{ OR somewhat}\big(\text{close}(d, Q_2)\big)\Big) \text{ AND importance}(d, Q_3)$$

Note that fuzzy set-theoretic queries may also be used as a query language to enable query expansion, like expanding a term to include synonymous terms.

Fuzzy set-theoretic query grammar. The set theoretic grammar described in Table 7 for set theoretic grammars can be extended to support fuzzy set-theoretic constructs.

```
        <query>  ::=  <atomic_query> |
                 ::=  <unary_op> <query>
                 ::=  (<query>)
                 ::=  <query> <binary_op> <query>
 <atomic_query>  ::=  <term> | <term> <atomic_query>
         <term>  ::=  <exact_phrase> | <keyword>
     <unary_op>  ::=  NOT | <hedge> | <search_type>
  <search_type>  ::=  close | importance | term_importance | ...
        <hedge>  ::=  somewhat | very | ...
    <binary_op>  ::=  AND | OR | ...
 <exact_phrase>  ::=  "<keywords>"
     <keywords>  ::=  <keyword> | <keyword> <keywords>
      <keyword>  ::=  <alphanumeric><keyword> | <alphanumeric>
 <alphanumeric>  ::=  a|b|...|z|0|1|...|9
```

Table 8 BNF Fuzzy Set-Theoretic Grammar

BOOLEAN PROXIMITY SEARCHING

In our experiments, we use MinDist* as a way to rank-order documents according to distance metric that takes as input a sum of minimum pairwise distances. However, another perhaps more useful—and far more straightforward—way to use the proximity information in secure indexes is to require that all of the terms in a query be within a minimum proximity of each other.

An algorithm to enable this functionality has already been essentially implemented for the MinDist* scoring function. It is less complicated (both conceptually and computationally) than the MinDist*.

Furthermore, MinDist* or BM25 can be used in tandem with Boolean proximity requirements, e.g., rank-order only those documents which contain all the terms in the query within a minimum proximity of each other.

Caching Results

Caching previously calculated results could result in significant savings. For example, whenever a term in a Boolean search is mapped to a set of documents, store the mapping in a cache so that subsequent Boolean searches involving the term may be serviced in near constant time.

We had initially included an LRU cache to memoize computations like the above, but we decided not to use them for the experiments for more predictable query lag times. In a practical implementation, a cache would be used to avoid doing unnecessary work. Since queries tend to be heavily biased towards a small subset of terms, this could result in significant savings.

Note that the CSP may technically do this without user permission. While it is a concern that a CSP may secretly collect such statistics, there may be little that can be done about it (with the exception of Oblivious RAM-like techniques).

CHAPTER VI
FUTURE WORK

Simulating an Adversary with Secure Index Access

On page 37 we provided a theoretical treatment on a hypothetical adversary who exploits the approximate and uncertain information in secure indexes to compromise their contents, e.g., reconstructing some fraction of their contents. We also included several experiments to analyze the effect of strategies intended to mitigate the risks posed by the adversary, e.g. secure index poisoning. However, due to time constraints, we did not implement a simulation of this adversary as we did for the query privacy adversary.

Mitigating Access pattern Leaks

As described on page 7, information leaks can take many other forms. To mitigate such information leaks, in general we can look to Oblivious RAM [18] for inspiration. Oblivious RAM may naively be thought of in the following way: to prevent meaningful statistics from being gathered about a user's activities, whenever an action—a read or write—is performed, include other randomly chosen actions (e.g., fake queries) to obscure the user's actual interests or activities.

Multiple secure indexes per document. For each document, construct multiple secure indexes in which each one will look different because they will each use a different document reference[45] and, more importantly, they will each use different salts for their searchable terms (trapdoors).

Consider the following. Let a document A read "$Hello\ world!$". Let us represent the confidential document A with $N = 3$ salts, resulting in three different secure indexes A_1, A_2, and A_3. Then, let A_i's trapdoors be $\{$"$hello|salt_i$", "$world|salt_i$", "$hello|world|salt_i$"$\}$. Thus, each time you perform a search, sample from the set of salts so that different variations of the secure indexes may be targeted.

Not only will this support query privacy (in the same way having multiple secrets do), but it will also cause the same query to return a set of logically equivalent results (with slight variations due to false positives) in $N = 3$ different ways. This increases the size of the secure index database by a factor of N and increases lag time by up to a maximum factor of N.

[45] Simply encrypt (using an invertible encryption scheme) the document reference with different salts.

Fake secure indexes. An extension of multiple secure indexes per confidential document is the automatic inclusion of fake secure indexes. These may be automatically generated from some given language model (e.g., trigram language model). The intent behind including fake secure indexes is to make it more difficult for an adversary to determine which documents retrieved in response to a query are of actual interest. The user who submitted the search will be able to instantly filter out the fake results, e.g., fake secure indexes will have some identifier in their reference string that is only discernable after decryption.

Fake queries. Instead of transmitting a single query for each actual query a user is interested in, a fake query rate parameter causes an appropriate number of additional fake queries to be generated[46]. Fake queries are strictly intended to obfuscate the user's actual queries of interest, just as query obfuscation is intended to obfuscate the actual terms of interest in a single query.

Fake queries may consist of terms chosen from a distribution that is likely to result in a plausible distribution of hits (i.e., relevant to an appropriately large set of documents), e.g., sample the terms in fake queries from a Zipf distribution. In this way, it may be impossible for an adversary to separate fake queries from real queries. Of course, this comes at the cost of increased resource consumption, e.g., the server must process more queries per legitimate query, thus increasing processing and network transmission requirements.

SEMANTIC SEARCH

On page 14, semantic searching was discussed. In the context of *Encrypted Search*, this can be implemented using standard natural language processing techniques, but it must generally be done in a pre-computed way due to the way trapdoors are constructed, i.e., some form of exact string matching must be performed on cryptographic hash values.

Consider the following. If a user's information need is represented by the query "carnivore hunting prey," one may assume she is also interested synonymous concepts or even more specific concepts, like "dog chasing cats" and "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 accurately determined, e.g., "carnivore" maps to "carnivore-1" (word sense 1 of carnivore in a dictionary). Using an ontology (like *WorldNet*), it can be determined that "carnivore-1" is a concept which includes more specific concepts like "dog-1" and "lion-2". Following this, we can expand[47] the query “carnivore hunting prey” into a set of queries that better represents the information need:

{“carnivore-1 hunting-3 prey-4”, “dog-1 chasing-1 cat-1”, “dog-1 chasing-1 feline-2”, “lion-2 hunting-3 antelope-1”, …} Subsequently, this query set must be converted into a (potentially prioritized) list of hidden queries, in which each hidden query consists of a list of cryptographic trapdoors.

In addition, during the preprocessing stage of secure index construction, similar techniques may be used to insert pre-computed cryptographic hashes such that exact string matches will occur for

[46] This may take the form of the user's client sending N fake queries per real query, where N is a discrete random variable, or it may consist of something else entirely, like a fake query bot providing a plausible flow of fake queries independently of real queries.

[47] A form of query expansion.

documents relevant to a given hidden query. For example, instead of just storing a *biword* model of a sentence that reads, “dog chasing cat,” store a word-sense disambiguated *biword* model of “dog-1 chasing-1 cat-1”. Furthermore, again using the same or similar techniques to semantic query expansion described previously, also insert synonymous or more general concepts, e.g., a *biword* model of "carnivore hunting prey”.

It is interesting to note that most of the work can be done in either the query expansion engine, or in the secure index representation. That is, we can generate an on-the-fly set of queries from a single query through query expansion techniques and leave the secure index alone, or we can front-load most of the work in the secure index and leave the queries alone.

There are well-known trade-offs to either approach, but in the context of *Encrypted Search*, there are additional considerations. Namely, if using query expansion, each query may expand to multiple hidden queries, all being related in some way. This information can be exploited by the adversary (see page 34), e.g., a more sophisticated probability model may use this information to mount more effective maximum likelihood attacks. In any future work, we could implement standard semantic search techniques, and explore these trade-offs with the aim of not only quantifying and mitigating information leakage.

TOPIC SEARCH (CLASSIFICATION)

Using Bayes rule, and an assumption of independent, identically distributed unigrams[48], we have the following Naïve Bayes simplification:

$$P[topic|doc] \approx \frac{P[topic]P[unigrams(doc) \mid topic]}{P[unigrams(doc)]} \approx \frac{P[topic]\prod_{u \in unigrams(doc)} P[u|topic]}{\prod_{u \in unigrams(doc)} P[u]}$$

Since the denominator is a constant given a document as evidence, it can be ignored when one is only interested in determining what the most likely topic is. Thus, the most likely $topic$ given doc is:

$$\text{most likely topic given doc} \approx \underset{topic}{\operatorname{argmax}} \left\{ P[topic] \sum_{u \in unigrams(doc)} P[u|topic] \right\}$$

For example, one topic may be *medical science*. Operationally, sample unigrams from a medical science corpus to estimate $P[u|medical\ science]$ and apply the $argmax$ formula. If $P[topic]$ cannot be estimated, a uniform distribution may be assumed (i.e., remove it from the $argmax$ formula).

This approach to topic search is founded upon a rigorous mathematical formalism, and it demonstrably works in other information retrieval contexts, but *Encrypted Search* may pose new challenges to it.

---

[48] If the secure index contains bigrams, Markov chains of order $m \geq 2$ may be used to model the true underlying distribution for a given topic with greater accuracy.

LETTER N-GRAMS AND WORD N-GRAMS

In our secure indexes, word unigrams and bigrams are atomic, indivisible units in the secure indexes. However, this is not necessarily the case. One could go in either direction, inserting larger units (e.g., trigrams) or smaller units (e.g., letter n-grams).

If larger word n-grams (e.g., trigrams) are used, then more secure (i.e., no need to deconstruct a trigram phrase into two bigrams which may leak co-occurrence information) and more exact searching on larger phrases will be the result.

If letter n-grams are used, then consider the string "hello world". If storing letter trigrams, then the following transformation takes place, where * denotes whitespace.

$$list[\text{“}hello\ world\text{”}] \rightarrow set\{\text{"}hel\text{"}, \text{"}ell\text{"}, \text{"}llo\text{"}, \text{"}lo*\text{"}, \text{"}o*w\text{"}, \text{"}*wo\text{"}, \text{"}wor\text{"}, \text{"}orl\text{"}, \text{"}rld\text{"}\}$$

To search for the word “hello”, check for the existence of "hel", "ell", and "llo" in the set. If all three letter trigrams exist, that word is said to exist. As in the *biword* model, false positives are possible. In addition, with letter n-grams partial word matches are automatically possible. For instance, if the user wishes to find any words matching “ello”, then simply check for the existence of “ell” and “llo”. In fact, any substring that is three characters or larger can be matched.

LEARNING OPTIMAL PARAMETERS

Encrypted Search benefits from an embarrassing amount of data from which to learn optimal parameters for any given scoring function. To learn optimal parameters, we need a training set of documents D_{set} and a query set Q_{set}. We construct a set of secure indexes SI_{set} for D_{set}, and then rank-order both SI_{set} and D_{set} according to each $q \in Q_{set}$. Then, we calculate the mean average precision (MAP) for the rank-ordered output from SI_{set} using the rank-ordered output from D_{set} as the $true$, canonical output.

For instance, $MinDistScore$ is a proximity scoring function has the following form:

$$MinDistScore(d, Q) = \ln\left(\alpha + \gamma \cdot \exp\left\{-\frac{\beta s}{|Q'|^{\theta}}\right\}\right),$$

where α, γ, β, and θ are tunable parameters. Thus, an example of an objective function to optimize is:

$$\operatorname*{argmax}_{\alpha,\gamma,\beta,\theta} \text{MeanAveragePrecision}(\text{RankOrder}(SI_{set}, Q_{set}), \text{RankOrder}(D_{set}, Q_{set}))$$

Since training data is abundant (it can be automatically generated, as we have done for our experiments), the tunable parameters for each scoring function should be independently optimized using a supervised learning algorithm that maximizes the mean average precision over the specified argmax parameters.

Note that this particular example is relatively straightforward, but more sophisticated techniques may be used to, for instance, avoid over-fitting on the training sets. Also, note that each secure index has many free parameters, e.g., location uncertainty, false positive rate, and so forth. These may be included as tunable parameters in the *argmax* optimization as well.

CHAPTER VII
CONCLUSIONS

Our research contributes to *Encrypted Search* in a few different ways. We designed and implemented several different new types of secure indexes—PSIB, PSIP, PSIF, and PSIM—based on a probabilistic set we call the Perfect Hash Filter.

On various metrics, we compared PSIB, PSIP, and PSIP to each other and to BSIB, a previously proposed secure index based on the popular Bloom filter probabilistic set. We compared them with respect to lag time, compression ratio, build time, load time, precision, recall, BM25, and MinDist*. On most benchmarks, the PSI-based secure indexes compared favorably to BSIB, especially with respect to lag time. Moreover, given the flexibility of the Perfect Hash Filter, we were able to use more sophisticated representations, like PSIP, which performed in some cases significantly better than both PSIB and BSIB in terms of accuracy and significantly better than BSIB in terms of lag time.

Moreover, in PSIP location uncertainty is independent of every other observed output except MinDist*—e.g., location uncertainty is independent of compression ratio, build time, BM25, etc. This flexibility makes it possible to choose a location uncertainty independent of concerns over these other outputs, and thus the choice of a location uncertainty becomes exclusively a trade-off between location accuracy (e.g., MinDist* accuracy) and confidentiality—i.e., confidentiality and location accuracy are inversely proportional. However, it performed generally worse on the compression ratio metric, although suggestions for ways to significantly improve this outcome were discussed.

We also explored the use of these standard information retrieval scoring techniques while paying close attention to confidentiality concerns. In general, we discovered that query privacy is greatly improved by using obfuscated queries and multiple secrets. Obfuscations had an insignificant impact on BM25 and MinDist*, and thus Encrypted Search is free to use obfuscations without significantly degrading relevancy of search results. However, obfuscations did significantly negatively affect Boolean search, e.g., if a user submits a search to find documents containing all of the terms in a query, and the query has obfuscations (fake terms), then very few documents (depending on the false positive rate) will both have the terms of interest to the user, and the obfuscated terms. Secrets, however, had no impact on the quality of search results, but they do degrade the compression ratio.

Experimentally, higher rates of obfuscation did not necessarily improve query privacy. Indeed, there was a global optimum such that confidentiality becomes progressively worse as you move away from it in either direction. We speculate that this was due to the distribution of obfuscation terms (uniformly sampled) being significantly different than the distribution of real terms (Zipf). If we chose a distribution for obfuscations that more accurately resembled the distribution of real terms, we believe there would be no such sweet spot; the higher the obfuscation rate, the better.

Including multiple secrets for each searchable atomic term (i.e., multiple trapdoors) also had a huge impact on query privacy. In this case, the more secrets there are, the stronger the confidentiality is (with respect to a simulated adversary using maximum likelihood attacks). However, there comes a point of diminishing returns in which the advantage of including one more secret is very unlikely to outweigh its cost in other respects, namely compression ratio.

Analytically, we also discovered that the location uncertainty should be quite large to preserve document confidentiality against an adversary, who has access to the raw contents of the secure index, employing *jig-saw*-like attacks. However, increasing the location uncertainty has a significant negative impact on MinDist* MAP accuracy[49]. In response to this insight, we designed and implemented the PSIM secure index, and suggested using it as a way to make any of the other secure indexes more sensitive to the MinDist* metric without revealing too much about the confidential document (i.e., the adversary would not have as much success with the proposed *jig-saw*-like attacks against PSIM).

While we did not perform any simulations of an adversary trying to compromise the confidentiality of secure indexes using such techniques, we did provide a detailed theoretical treatment to motivate experiments on secure index poisoning and high false positive rates.

Increasing the false positive rate had little impact on compression ratio and only a small impact on BM25 and MinDist* MAP scores. However, increasing the false positive rate had a huge impact on precision, as expected. This expectation was one of the motivations for exploring the use of standard degree of relevancy scoring techniques for *Encrypted Search*—these scoring techniques seem to work well despite the presence of approximation errors and false positives when using secure indexes.

Also, secure index poisoning, especially in the form of adding fake terms, encouragingly had little impact on BM25 MAP, MinDist* MAP, and precision, and only a modest impact on compression ratio. An *Encrypted Search* user is relatively free to poison a secure index to whatever desired level and still be justified in expecting good search performance in terms of accuracy and speed.

---

[49] On top of that, PSIB and BSIB do not scale well to large numbers of blocks—which is the ratio of word count to location uncertainty—and thus it may not even be a reasonable option to use small location uncertainties (unless the documents are reasonably small). PSIP, as just discussed, does not possess this problem.

REFERENCES

[1] D. X. Song, D. Wagner and A. Perri, "Practical Techniques for Searches on Encrypted Data," *Proceedings of the 2000 IEEE Symposium on Security and Privacy,* no. Dawn Xiaodong Song, David Wagner, and Adrian Perrig, pp. 44-55, 2000.

[2] G. Navarro, E. S. de Moura, M. Neubert, N. Ziviani and R. Baeza-Yates, "Adding Compression to Block Addressing Inverted Indexes," *Information Retrieval,* vol. 3, no. 1, pp. 49-77, 2007.

[3] G. G. Langdon, "Huffman codes," 2000.

[4] M. Mowbray, S. Pearson and Y. Shen, "Enhancing Privacy in Cloud Computing via Policy-Based Obfuscation," *Journal of Supercomputing,* vol. 61, p. 267–291, 2012.

[5] C. T. a. D. L. Christian Collberg, "A Taxonomy of Obfuscating Transformations," 1997.

[6] D. Hofheinz, J. Malone-lee and M. Stam, "Obfuscation for cryptographic purposes," *TCC,* pp. 214-232., 2007.

[7] P. Golle, J. Staddon and B. Waters, "Secure Conjunctive Keyword Search over Encrypted Data," *Applied Cryptography and Network Security, Lecture Notes in Computer Science,* vol. 3089, pp. 31-45, 2004.

[8] Z. Wei, Z. Dan-Feng, G. Feng and L. Guo-Hua, "On Indexing and Information Disclosure Measure for Efficient Cryptograph Query," *Proceedings of the World Scientific and Engineering Academy and Society International Conference on Computers,* pp. 476-480, 2009.

[9] C. Dong, G. Russello and N. Dulay, "Shared and Searchable Encrypted Data for Untrusted Servers," *Data and Applications Security XXII, Lecture Notes in Computer Science,* vol. 5094, pp. 127-143, 2008.

[10] M. R. Asghar, G. Russello, B. Crispo and M. Ion, "Supporting Complex Queries and Access Policies for Multi-User Encrypted Databases," *Proceedings of the ACM Workshop on Cloud Computing Security Workshop,* pp. 77-88, 2013.

[11] J. Li, Q. Wang, C. Wang, N. Cao, K. Ren and W. Lou, "Fuzzy Keyword Search over Encrypted Data in Cloud Computing," in *Proceedings of IEEE INFOCOM*, 2010.

[12] M. Celikik and H. Bast, "Efficient Fuzzy Search in Large Text Collections," *ACM Transactions on Information Systems,* vol. 31, no. 2, pp. 1-59, May 2013.

[13] D. Boneh, G. D. Crescenzo, R. Ostrovsky and G. Persiano. , "Public-key encryption with keyword search," *Proceedings of Eurocrypt, Lecture Nodes in Computer Science,* May 2004.

[14] W. Diffie and M. E. Hellman, "New directions in cryptography," 1976.

[15] M. Naor and M. Yung, "Universal One-Way hash functions and their cryptographic applications," pp. 33-43, 1989.

[16] E.-J. Goh, "Secure Indexes," *Trust, Privacy, and Security in Digital Business, Lecture Notes in Computer Science,* vol. 3592, pp. 128-140, 2005.

[17] A. Swaminathan, Y. Mao, G.-M. Su, H. Gou, A. Varna, S. He, M. Wu and D. Oard, "Confidentiality-Preserving Rank-Ordered Search”".

[18] B. P. a. T. Reinman, "Oblivious RAM revisited".

[19] S. Artzi, C. Newport and D. Schultz, "Encrypted keyword search in a distributed storage system".

[20] A. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey," *Internet Mathematics,* vol. 1, no. 4, pp. 485-509, 2002.

[21] N. Cao, C. Wang, M. Li, K. Ren and W. Lou, "Privacy-Preserving Multi-keyword Ranked Search over Encrypted Cloud Data," *Proceedings of IEEE INFOCOM,* April 2011.

[22] Y.-c. Chang and M. Mitzenmacher, "Privacy preserving keyword searches on remote encrypted data," *Lecture Notes in Computer Science,* vol. 3531, pp. 442-455, 2005.

[23] Q. Liu, G. Wang and J. Wu, "An Efficient Privacy Preserving Keyword Search Scheme in Cloud Computing," *Proceedings of International Conference on Computational Science and Engineering,* vol. 2, pp. 715-720, August 2009.

[24] H. Cao, D. Jiang, J. Pei, E. Chen and H. Li, "Towards Context-Aware Search by Learning a Very Large Variable Length Hidden Markov Model from Search Logs," *Proceedings of the International Conference on World Wide Web,* pp. 191-200, 2009.

[25] F. Giunchiglia, U. Kharkevich and I. Zaihrayeu, "Concept Search: Semantics Enabled Syntactic Search," *Proceedings of CEUR Workshop,* 2008.

[26] R. A. Baeza-yates, "Text retrieval: Theory and practice," *In 12th IFIP World Computer Congress,* vol. I, pp. 465-476, 1992.

[27] C. Buckley and G. Salton, "Term-weighting approaches in automatic text retrieval," *INFORMATION PROCESSING AND MANAGEMENT,* vol. 24, pp. 513-523, 1988.

[28] H. Cao, D. H. Hu, D. Shen, D. Jiang, J.-T. Sun, E. Chen and Q. Yang, "Context-Aware Query Classification," *Proceedings of the International ACM SIGIR Conference on Research and Development in Information Retrieval,* pp. 3-10, 2009.

[29] P. Indyk and R. Motwani, "Approximate nearest neighbors: Towards removing the curse of dimensionality," pp. 604-613, 1998.

[30] Y. Hua, B. Xiao, B. Veeravalli and D. Feng, "Locality-Sensitive Bloom Filter for Approximate Membership Query," *IEEE Transcations on Computers,* vol. 61, no. 6, pp. 817-830, 2012.

[31] R. Krovetz, "Viewing morphology as an inference process," pp. 191-202, 1993.

[32] R. Brinkman, P. Hartel, W. Jonker and C. Bösch, "Conjunctive Wildcard Search over Encrypted Data," *Proceedings of the VLDB International Conference on Secure Data Management,* pp. 114-127, 2011.

[33] Y. Tang, D. Gu, N. Ding and H. Lu, "Phrase Search over Encrypted Data with Symmetric Encryption Scheme," *2012 32nd International Conference on Distributed Computing Systems Workshops (ICDCSW),* pp. 471-480, 2012.

[34] C. D. Manning, P. Raghavan and H. Schütze, in *An Introduction to Information Retrieval*, Cambridge University Press, 2009, p. 233.

[35] R. Schenkel, A. Broschart, S. Hwang and M. Theobald, "Efficient Text Proximity Search," *Lecture Notes in Computer Science,* pp. 287-299, 2007.

[36] S. Tarkoma, C. E. Rothenberg and E. Lagerspetz, "Theory and Practice of Bloom Filters for Distributed Systems," *IEEE Communications Surveys & Tutorials,* 2012.

[37] Z. Kissel and J. Wang, "Verifiable Symmetric Searchable Encryption for Multiple Groups of Users," *Proceedings of SAM 2013,* 2013.

[38] D. Belazzougui, F. Botelho and M. Dietzfelbinger, "Hash, displace, and compress," in *Algorithms - ESA 2009*, vol. 5757, Springer, 2009, pp. 682-693.

[39] C. D. Manning, P. Raghavan and H. Schütze, in *An Introduction to Information Retrieval*, Cambridge University Press, 2009, p. 121.

[40] T. Tao and C. Zhai, "An Exploration of Proximity Measures in Information Retrieval," in *SIGIR '07 Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval Pages 295-302*, 2007.

[41] C. Gentry, "Fully homomorphic encryption using ideal lattices," *Proc. STOC,* pp. 169-178, 2009.

[42] A. Kumar, J. J. Xu, J. Wang and L. Li, "Space-Code bloom filter for efficient traffic flow measurement," *Proceedings of the 2003 ACM SIGCOMM conference on Internet measurement,* pp. 167-172, 2003.

[43] Q. Liu, G. Wang and J. Wub, "Secure and Privacy Preserving Keyword Searching for Cloud Storage Services," *Journal of Network and Computer Applications,* vol. 35, no. 3, pp. 927-933, May 2012.

[44] S. Pearson, Y. Shen and M. Mowbray, "A Privacy Manager for Cloud Computing," *Cloud Computing, Lecture Notes in Computer Science,* vol. 5931, pp. 90-106, 2009.

[45] Y. Lu and G. Tsudik, "Enhancing Data Privacy in the Cloud," *Trust Management V, IFIP Advances in Information and Communication Technology,* vol. 358, pp. 117-132, 2011.

[46] R. Latif, H. Abbas, S. Assar and Q. Ali, "Cloud Computing Risk Assessment: A Systematic Literature Review," *Future Information Technology, Lecture Notes in Electrical Engineering,* vol. 276, pp. 285-295, 2014.

[47] S. Mehrotra, C. Li, B. Iyer and H. Hacigümüş, "Executing SQL over Encrypted Data in the Database-Service-Provider Model," *Proceedings of the ACM SIGMOD International Conference on Management of Data,* pp. 216-227, 2002.

[48] B. Zhu, B. Zhu and K. Ren, "PEKSrand: Providing Predicate Privacy in Public-Key Encryption with Keyword Search," *Proceedings of IEEE International Conference on Communications,* pp. 1-6, 2011.

[49] R. Curtmola, J. Garay, S. Kamara and R. Ostrovsky, "Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions," *Proceedings of the ACM Conference on Computer and Communications Security,* pp. 79-88, 2006.

[50] B. Xiang, D. Jiang, J. Pei, X. Sun, E. Chen and H. Li, "Context-Aware Ranking in Web Search," *Proceeding of the International ACM SIGIR Conference on Research and Development in Information Retrieval,* pp. 451-458, 2010.

[51] Y. Shen and J. Yan, "Sparse Hidden-Dynamics Conditional Random Fields for User Intent Understanding," *Proceedings of the International Conference on World Wide Web,* no. Shuicheng, Lei Ji, Ning Liu, Zheng Chen, pp. 7-16, 2011.

[52] J. Chen, H. Guo, W. Wu and W. Wang, "iMecho: a Context-Aware Desktop Search System," Proceedings of the International ACM SIGIR conference on Research and development in Information Retrieval," pp. 1269-1270, 2011.