3-5-2020

Balancing Security, Performance and Deployability in Encrypted Search

David Joel Pouliot
Portland State University

Balancing Security, Performance and Deployability in Encrypted Search

by

David Joel Pouliot

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Computer Science

Dissertation Committee:
Charles Wright, Chair
Wu-chang Feng
David Maier
Wu-chi Feng

Portland State University
2020

Abstract

Encryption is an important tool for protecting data, especially data stored in the cloud. However, standard encryption techniques prevent efficient search. Searchable encryption attempts to solve this issue, protecting the data while still providing search functionality. Retaining the ability to search comes at a cost of security, performance and/or utility.

An important practical aspect of utility is compatibility with legacy systems. Unfortunately, the efficient searchable encryption constructions that are compatible with these systems have been proven vulnerable to attack, even against weaker adversary models.

The goal of this work is to address this security problem inherent with efficient, legacy compatible constructions. First, we present attacks on previous constructions that are compatible with legacy systems, demonstrating their vulnerability. Then we present two new searchable encryption constructions. The first, weakly randomized encryption, provides superior security to prior "easily deployable" constructions, while providing similar ease of deployment and query performance nearly identical to unencrypted databases. The second construction, EDDiES, provides much stronger security at the expense of a slight regression on performance.

These constructions show that it is possible to achieve a better balance of security and performance with the utility constraints that come with deployment in legacy systems.

Contents

List of Tables

List of Figures

List of Algorithms

List of Theorems

1 Introduction

Many organizations today are moving to the cloud, shipping their critical data to servers over which they have little control. Utilizing the cloud comes with numerous benefits, including high availability, easy data access, reduced infrastructure costs, faster ramp up time for development, and ability to access data from anywhere.

This ability to access data from anywhere is almost a universal requirement for many companies and individuals. In house enterprise servers also provide this ability to access data from anywhere. This feature comes with added security risk. Any server that allows data access from anywhere is now subject to attacks from anywhere.

It is not surprising then that data breaches are becoming more common and gaining more attention as businesses of all sizes become increasingly reliant on digital data and cloud computing. As companies store sensitive data on enterprise databases and cloud servers, breaching a company's data is a matter of gaining access to restricted networks.

Using cloud services also comes with a new threat to data privacy, the cloud provider itself. The server has full access to the users data. They may only be curious, but they might also abuse this data access.

The most common way of securing this data is to encrypt it before storing the data on the cloud or enterprise servers. This provides end to end security from the time it leaves the user. In many instances, companies do not encrypt their data. They rely on access controls to protect it. The reason for not encrypting the data is that standard encryption mechanisms also prevent the server from performing any useful computation on the clients behalf. One of the most desirable operations on encrypted data is search. Encrypting the data would not stop all breaches, but it would require the attacker to compromise the account or password of specific user(s) to gain access to the data.

The research community has attempted to address this need with searchable encryption. The origins of this research start in the year 2000. But even with almost 20 years of research in this field, we are still far from the ideal *practical* solution. All the proposed constructions so far have varying compromises in security, performance and utility. The most secure systems include extensive performance and utility limitations.

In an ideal setting, encrypted data should not leak even 1 bit of information about the encrypted plaintext. Anyone without the key should not be able to distinguish between the ciphertext and random bits of the same length. This is the informal security definition for encryption. With encrypted search, in order to achieve sublinear performance, relaxing this standard is necessary.

Each efficient encrypted search construction has a leakage profile. With some security definitions, this is formally defined. With other security definitions, the leakage is implicit and determined by either a limitation of the attacker's power in the definition or an assumption of the data that is encrypted. The consequences of this leakage vary. Some types of leakage are easily attacked, while others are more resistant.

Utility can refer to many features, ease of deployment, multiple user support and types of queries supported. Most encrypted search systems support only basic equality and range queries. Some have added support for boolean queries, prefix queries and others.

One of the most important practical aspects of utility is compatibility with legacy systems. Without this compatibility, the time and money required to deploy searchable encryption is too great. If the costs of data breaches increase enough, that may change. But until the incentives or economics change regarding data security, this type of utility is a practical necessity. Unfortunately, the systems that offer the best performance and compatibility have severe security issues.

1.1 Research Problem

The research problem addressed in this dissertation is, "Is it possible to produce searchable encryption techniques that provide acceptable levels of performance and security, with a hard constraint of being compatible with existing systems?" This dissertation focuses on a specific class of existing systems, legacy Database Management Systems (DBMS). **Contributions**. We provide the following contributions towards the development of searchable encryption that is compatible with legacy systems:

- In Chapter 3 we analyze recent attempts to provide searchable encryption to legacy systems on the web and Android platform. The constraints of these systems placed significant restrictions on the type of searchable encryption that could be implemented. We demonstrate that these constructions are vulnerable to attacks.

- In Chapter 4 we present a new construction: weakly randomized encryption (WRE). WRE utilizes various techniques to add enough randomness to deterministic encryption, resulting in a system that is significantly more resistant to inference attacks, while still retaining high performance and ease of deployment.

- In Chapter 5, we present another new construction, EDDiES, which takes inspiration from Bloom filters. This construction provides much stronger security over WRE, but it makes some sacrifices in performance relative to WRE or unencrypted databases.

2 Background and Related Work

This chapter provides necessary background information used in the rest of the thesis. Chapter 2.1 provides background information on generic search strategies. Chapter 2.2 contains a cryptographic background. Chapters 2.3 through 2.7 provide background on encrypted search. Chapter 2.8 reviews attacks against encrypted search.

2.1 Search Background

The desire to perform searches in sub-linear time resulted in the creation of search indexes. These indexes provide data structures tailored to specific search strategies.

2.1.1 Full-Text Indexing

A standard approach for efficient, full-text sub-linear search over unencrypted documents is to utilize an inverted index. In an inverted index, each keyword contains a list of all the documents that contain the keyword. Common words are typically excluded from the index. The search complexity can be reduced to $O(|D(w)|)$ where $|D(w)|$ is the number of documents containing keyword w when utilizing inverted indexes. Table 2.1 is an example of an inverted index.

keyword	document ids
w_1	3,5
w_2	1,9,6
...	...
w_m	...

Table 2.1: Inverted Index Example

2.1.2 Bloom Filters

Bloom filters [8] are probabilistic data structures that represent sets and support membership queries. For applications that can tolerate a small false-positive rate, Bloom filters offer a space-efficient alternative to the full inverted index. Conceptually, the Bloom filter is an array or bit vector of m bits, all initially set to zero. To insert an element x into the set, we hash x with each of k hash functions and set each of the k bits $b_i = h_i(x), 1 \leq i \leq k$ to one in the Bloom filter. To check if an item z is in the set, we check if $h_i(z) = 1$ for all $1 \leq i \leq k$.

The standard Bloom filter construction described by Bloom allows anyone to check for the presence of an item in the filter. This is not desirable for indexing encrypted data; it could be leveraged by an attacker to perform a dictionary attack. Therefore searchable encryption schemes use a ether a keyed hash function or a pseudorandom function (PRF) (See Definition 2.4) to set the bits in the Bloom filter. Informally, it is not possible to tell the difference between the output of a pseudorandom function and random sequences. More formally, let $f_1, f_2, ..., f_k$ be a family of k pseudorandom functions. Let $F(w)$ be the set of bits in the Bloom filter that correspond to keyword w, i.e. $F(w) = \{f_i(w) : i \in [1, k]\}$. Only someone who has the secret symmetric key can compute $F(w)$. The use of keyed functions prevent simple brute-force dictionary attacks on the hash function. In practice, the PRF can be instantiated as a truncated message authentication code (MAC), sometimes also called a "keyed hash." For example, Mimesis Aegis [48] uses HMAC-SHA256 to set bits in a Bloom filter of size 2^{24}. Figure 2.1 illustrates the insertion into a Bloom filter using keyed hash functions.

Since the typical variable names for Bloom filters are similar to cryptographic variable names, we will use the following variables for Bloom filters:

- s, the size of the Bloom filter (instead of m)

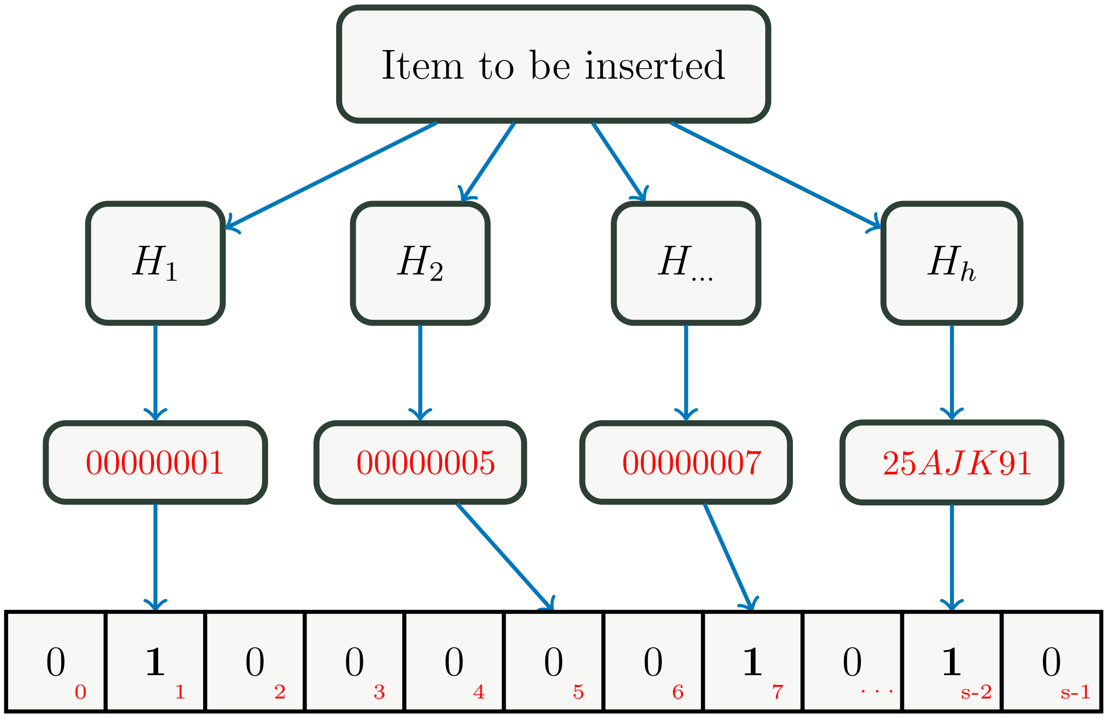


Figure 2.1: Bloom filter insertion example, h hash functions, Bloom filter size=s

- t, the number of words inserted into the Bloom filter (instead of n)
- h, the number of hash functions (instead of k)

Using these parameters, there is a formula from [13] that gives us the probability of any specific bit is set to one:

$$PR[bit_i = 1] = 1 - (1 - \frac{1}{s})^{ht} \approx 1 - e^{\frac{-ht}{s}} \tag{2.1}$$

For a false positive to occur, each bit position in a Bloom filter for a word has to be set to one. Since it requires h of these bits for the false positive, it follows that the false positive rate is

$$p^h \text{ where } p = PR[bit_i = 1] \tag{2.2}$$

2.1.3 Database Indexing

Database management systems support many different index types. They differ in their structure and query types supported.

B-Tree. One of the most common indexes is a B-Tree. It is the default index on many systems and supports efficient equality and range queries. However some data types, such as geometric and some queries (such as those with wildcards), do not work with the standard B-Tree index.

Generalized Inverted Indexes (GIN). GIN indexes allow efficient search for an item within data, such as arrays or full text. GIN also allows custom data types, provided that the appropriate access functions are implemented.

Generalized Search Tree (GIST). The Gist [37] framework utilizes a tree structure to store its data types. It is designed to index custom data types. Similar to GIN, it is extensible to these custom data types, requiring specific functions to be implemented.

2.2 Cryptography Background

This section provides some background on symmetric encryption, also known as private-key encryption. Since this work focuses on symmetric searchable encryption, it omits background work related to public-key encryption. A significant portion of this background are definitions. These definitions are a key foundation of modern cryptography, formal definitions and proofs of security.

Definition 2.1 (Symmetric Encryption)**.** *A* symmetric encryption *construction is a tuple of probabilistic polynomial-time algorithms* $(\textsc{Gen}, \textsc{Enc}, \textsc{Dec})$*:*

- $\textsc{Gen}$ *is a key generation algorithm whose input is a security parameter* 1^n *and whose output is a key* k. *This is written as* $k \leftarrow \textsc{Gen}(1^n)$ *and is a randomized algorithm. Any key* k *output satisfies* $|k| \geq n$.

- $\textsc{Enc}$ *is the encryption algorithm. It takes as inputs the key* k *and a* ***plaintext*** *message* $m \in \{0,1\}^*$ *and outputs a* ***ciphertext*** c. $\textsc{Enc}$ *may be randomized and is written as* $c \leftarrow \textsc{Enc}_k(m)$.

- $\textsc{Dec}$ *is the decryption algorithm. It takes as input a key* k *and* ***ciphertext*** c *and outputs a* ***plaintext*** *message* m *or error.* $\textsc{Dec}$ *is a deterministic algorithm and is written* $m := \textsc{Dec}_k(c)$.

For every $k, m \in \{0,1\}^*$, *it is required that* $\textsc{Dec}_k(\textsc{Enc}_k(m)) = m$.

Definition 2.2 (Negligible Function)**.** *A function* $\mu : N \rightarrow N$ *is* negligible *in* k *if for every positive polynomial* $p(\cdot)$ *and sufficiently large* $k, \mu(k) < 1/p(k)$. *Let* $poly(k)$ *and* $negl(k)$ *denote unspecified polynomial and negligible functions in* k, *respectively.*

The negligible function definition is used in numerous security definitions. Practical cryptographic constructions achieve computational security. That is, against attackers with bounded computational power, they leak a very small amount of information. This small amount of information leaked is often defined in the form of a negligible function.

Definition 2.3 (IND-CPA Security)**.**

CPA Indistinguishability experiment $IND-CPA_{\mathcal{A},\Pi}$

- $k \leftarrow \textsc{Gen}(1^n)$

- *Adversary* $\mathcal{A}$ *is given* 1^n *and oracle access to* $\textsc{Enc}_k(\cdot)$. $\mathcal{A}$ *outputs a pair of messages* m_0, m_1 *of equal length.*

- *A uniform bit* $b \in \{0,1\}$ *is chosen.* $\mathcal{A}$ *is given* $c \leftarrow \textsc{Enc}_k(m_b)$.
- $\mathcal{A}$ *is given oracle access* $\textsc{Enc}_k(\cdot)$.
- $\mathcal{A}$ *outputs a bit,* b'
- *If* $b = b'$*, then the output is 1 and* $\mathcal{A}$ *succeeds. A zero is output otherwise.*

An encryption scheme $\Pi = (\textsc{Gen}, \textsc{Enc}, \textsc{Dec})$ *is* indistinguishable *under chosen plaintext attack, IND-CPA if for all probabilistic polynomial time adversaries* $\mathcal{A}$ *there is a negligible function:*

$$Pr[IND - CPA_{\mathcal{A},\Pi} = 1] \leq \frac{1}{2} + negl(n)$$

IND-CPA security is a commonly used security definition and is often a requirement for most cryptographic systems. Its adversary is given access to an encryption oracle, a black box which encrypts messages that the attacker chooses using a key that is unknown to the attacker. The attacker then chooses two messages for the challenger to encrypt using the same key. The challenger chooses one of these messages randomly, encrypts it and returns it to the attacker. The attacker then has oracle access again. If the attacker cannot guess which message with probability higher than random guessing ($\frac{1}{2}$), then the system is IND-CPA secure.

Definition 2.4 (Pseudo-Random Function (PRF))**.** *Let* $F : \{0,1\}^k \times \{0,1\}^n \rightarrow \{0,1\}^m$ *be an efficient keyed function.* F *is a* pseudo-random function *if for all probabilistic polynomial time distinguishers* $\mathcal{D}$ *there is a negligible function negl such that:*

$$\left| Pr[\mathcal{D}^{F_k(\cdot)} = 1] - Pr[\mathcal{D}^{f(\cdot)} = 1] \leq negl(k) \right|$$

*where the first probability is taken over the uniform random choice of k and the randomness of $\mathcal{D}$ and the second probability it taken over the uniform choice of $f \in$ **Func**$_n$ and the randomness of $\mathcal{D}$.*

Informally, a pseudo-random function is a polynomial time function that is indistinguishable from a truly random function by any polynomial time adversary.

Definition 2.5 (Statistical Distance)**.** *The* statistical distance Δ *between two random variables X, Y over a common domain ω is defined as:*

$$\Delta(X,Y) = \frac{1}{2}\sum_{\alpha\in\omega}\Big|Pr(X=\alpha) - Pr(Y=\alpha)\Big|$$

Two random variables X, Y are said to be ϵ-*close* if the statistical distance between them is at most ϵ. Variables X, Y are called *statistically indistinguishable* if $\epsilon = negl(\alpha)$ with security parameter α.

An important application of Definition 2.5 is its use in the probability of distinguishing between two random variables or two distributions [10]. This probability is bounded by the statistical distance between the distributions.

Definition 2.6 (Distinguishing Two Distributions)**.** *Let P_0 and P_1 be probability distributions on a finite set R. Then, for every adversary $\mathcal{A}$, we have the* distinguishing advantage *of $\mathcal{A}$ between P_0 and P_1,*

$$Pr[\mathsf{Dist}_{\mathcal{A}}(P_0, P_1)] \leq \Delta(P_0, P_1)$$

Definition 2.7 (Shuffle)**.** *Let S be a set containing n distinct objects. A* Shuffle *of S is an ordered list of the objects in S. A* Shuffle *of the set $\{1, 2, ..., n\}$ is called a* Shuffle *of n.*

To shuffle a list, the set S is the indexes into the list. The shuffle of a list re-orders the indexes. Informally, this is a permutation of a list.

If S contains n distinct objects, then there are exactly $n!$ Shuffles of n.

Definition 2.8 (Pseudo-Random Shuffle (PRS)). *Let* PRS *be a deterministic polynomial time function that on input key* $k \in \{0,1\}^n$*, message* $m \in \{0,1\}^*$*, and list of messages* $[l_0, l_1, ...l_i]$ *where* $l_i \in \{0,1\}^*$*, outputs a* Shuffle *of* $[l_0, l_1, ...l_i]$*. We say* PRS *is a* Pseudo Random Shuffle *if:*

- *(Pseudorandomness:) For any probabilistic polynomial time algorithm* $\mathcal{D}$*, there is a negligible function* negl *such that*

$$\left| Pr[\mathcal{D}(\mathrm{PRS}(k, m, l))] - Pr[\mathcal{D}(\mathrm{R}(l))] \right| \leq \mathrm{negl}(n)$$

  where the first probability is taken over the uniform choice of $k \in \{0,1\}^n$, $m \in \{0,1\}^*$ *and the randomness of* $\mathcal{D}$*, and the second probability is taken over* $\mathrm{R}(l)$*, where* R *is a uniformly random shuffle algorithm.*

2.3 Origins of Searchable Encryption

Song, Wagner, and Perrig proposed the first scheme for searching on encrypted data [56]. Their construction encrypts keywords in documents by exclusive XORing them with the output of a stream cipher and a Pseudo-Random Function (PRF). A keyword is given the same PRF key across multiple documents, which facilitates the search. Searching is linear in the number and size of the documents. Being the first research product in this field, there were no standard security definitions. They proved their construction is a pseudo-random generator, proving the security of the ciphertexts, but not addressing the security when queries are issued.

Goh [29] provided the first formal security definition for searchable encryption, semantic security against adaptive chosen keyword attacks (IND-CKA). Informally, it ensures that an attacker cannot deduce a document's contents from the index. Their construction utilizes Bloom filters for indexes, where the input to the PRFs is the search term and document ID. A given keyword sets different Bloom filter address locations for different documents. As a result, the search time is linear in the number of documents.

The IND-CKA definition did not address documents of varying sizes. An updated definition by Goh and also Chang and Mitzenmacher [17] resolved the varrying sized documents. Even this latest definition was proven insufficient, as Curtmola et al. [21] shows that an insecure scheme meets their definition.

2.4 Symmetric Searchable Encryption (SSE)

Curtmola et al. [21] provided the set of security definitions for Symmetric Searchable Encryption (SSE) [15, 21, 40, 57] that are still used today. Their improved definitions allowed for varying sized documents, and protected the contents of the document and the keywords. They also added an adaptive adversarial definition where the adversary is allowed to issue a query and see the results before issuing another query.

Early Symmetric Searchable Encryption schemes assumed a relatively static document corpus and offered somewhat limited performance. Current SSE schemes can handle dynamic data [40] and offer effective performance even on very large data sets [15]. However, the security definitions for SSE require that the server must engage in a cryptographic protocol with the client to execute searches on their behalf.

A typical SSE scheme will encrypt the data using IND-CPA encryption and provide an encrypted index as well to facilitate search.

Definition 2.9 (Searchable Symmetric Encryption)**.** Symmetric Searchable Encryption (SSE) *consists of the following algorithms run by the client and server:*

$K \leftarrow \textsc{Gen}(1^k)$ *: is a probabilistic key generation algorithm that is run by the user to setup the scheme. It takes as input a security parameter k, and outputs a secret key K.*

$(I, c) \leftarrow \textsc{Enc}(K, D)$*: is a probabilistic algorithm run by the user to encrypt the document collection. It takes as input a secret key K and a document collection $D = (D_1, ..., D_n)$, and outputs a secure index I and a sequence of ciphertexts $c = (c_1, ..., c_n)$. We sometimes write this as $(I, c) \leftarrow \textsc{Enc}_K(D)$.*

$t \leftarrow \textsc{Trpdr}(K, w)$ *: is a deterministic algorithm run by the user to generate a trapdoor for a given keyword. It takes as input a secret key K and a keyword w, and outputs a trapdoor t. We sometimes write this as $t \leftarrow \textsc{Trpdr}_K(w)$.*

$X \leftarrow \textsc{Search}(I, t)$ *is a deterministic algorithm run by the server to search for the documents in D that contain a keyword w. It takes as input an encrypted index I for a data collection D and a trapdoor t and outputs a set X of (lexicographically-ordered) document identifiers.*

$D_i \leftarrow \textsc{Dec}(K, c_i)$*: is a deterministic algorithm run by the client to recover a document. It takes as input a secret key K and a ciphertext c_i, and outputs a document D_i. We sometimes write this as $D_i \leftarrow \textsc{Dec}_K(c_i)$.*

To search, the server is provided a trapdoor (one way function) to access one keyword's list of documents from the index. The trapdoor does not allow the server to see the plaintext version of the keyword, it only allows the server to decrypt the list of documents that contain the keyword. If an adversary steals the encrypted data, but does not see the queries, no real information is leaked. Most SSE schemes achieve efficiency comparable to the most efficient encrypted search solutions (e.g., deterministic encryption) but provide superior security as they only leak query search pattern and access pattern results.

The SSE constructions offer one of the best combinations of performance and security. However applying SSE to database systems has its issues, because of the requirement that clients perform all index creation and updating.

Locality. More recent SSE research deals with improving the performance of SSE. The practical performance of SSE schemes depends on the locality – the number of non-continuous locations that the server accesses for each query. Cash et al. proposed new SSE schemes with good locality [15] to allow scaling to large data sizes. Various other works [4, 16, 18, 23] provide constructions with improved locality, read efficiency and server storage.

Dynamic SSE. The early SSE schemes had inefficient ways of handling updates, forward and backward security. Updates are challenging with SSE because the act of updating an index leaks information. Forward security deals with the issue of inserting new data into the corpus after searches have been performed. Backward privacy guarantees that queries do not reveal their association to deleted documents.

Once a search is issued, the server has the trapdoor for that search and can perform the search at any time. Thus, when data is inserted or deleted, the server can see if that data matches previous searches or not. The early SSE schemes dealt with updates and achieved forward security by issuing batch updates. Each update would contain its own new set of encrypted indexes. Periodically these indexes would be combined into one.

Several works have focused on improving dynamic SSE [11, 15, 39, 40, 40, 57]. The

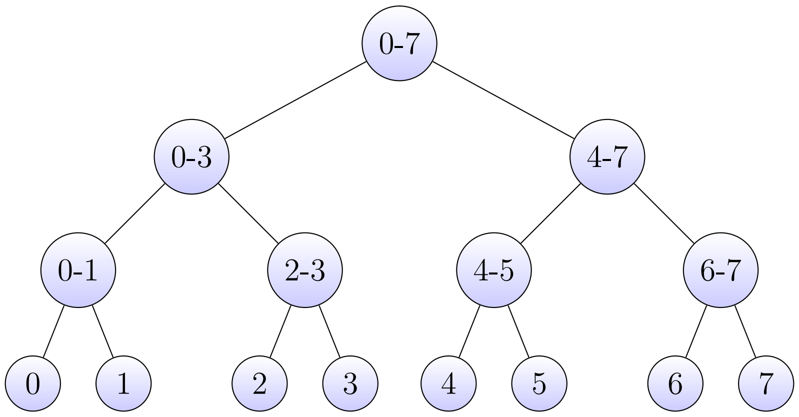


Figure 2.2: Range Query Tree Example 0-7

notion of forward and backward privacy was proposed by Stefanov et al. [57] and later formalized by Bost [11, 12]. Several works have been produced with forward privacy [11, 12, 24, 26, 28, 43, 57, 58] and backward privacy [12, 28, 58].

Range Queries. SSE constructions do not natively support range queries due to their inverted-index structure. Range queries are reduced to multiple single keyword queries. A tree structure is used to cover all the possible ranges. Figure 2.2 shows an example with range values from 0-7. Each node in the tree is an index for values in that range. If a document or record had the value 3, it would be in the index for 0-7, 0-3, 2-3 and 3.

However this structure leaks additional information to the adversary. In Figure 2.2, node (3) is in the range search for 3, 3-5, and 3-7. Also range queries have varying number of nodes, the range from 0-3 uses one node and the range 2-4 uses 2.

Demertzis et al. [22] address this additional leakage with revisions to the search strategies and tree structures. Wang et al. [62] propose a dynamic SSE construction that supports range queries.

2.5 Property Preserving Encryption (PPE)

Another type of searchable encryption evolved in a different path. The desire to encrypt databases, but still use SQL queries, led to Property Preserving Encryption (PPE). Property preserving encryption is useful because it allows comparative operations over the ciphertexts. This facilitates using database indexes on the ciphertexts with no modifications. There are a few commonly used types of Property Preserving Encryption used. Deterministic encryption [5] is one of them. Unlike randomized encryption, with deterministic encryption, each message m is mapped to a single ciphertext under a key K.

Order Preserving Encryption (OPE) [9] is a symmetric encryption scheme with the following property: If $m_1 < m_2$, then $Enc_K(m_1) < Enc_K(m_2)$. Order Preserving Encryption allows the user to perform range queries on an encrypted database. Order Revealing Encryption (ORE) typically results in ciphertexts that appear random. It uses a publicly computable function which takes two ciphertexts as input and outputs the relative ordering of the two ciphertexts.

These constructions do not follow a universal security model. Instead each construction uses a different security definition with limitations on the adversary's power, depending on what the construction leaks.

There are instances where these constructions are used outside their intended scope. For example, the security definition for deterministic encryption places a *high min-entropy* requirement on the data it is encrypting. Essentially deterministic encryption is secure only for unique data such as social security numbers. However it has been used in applications that do not meet this key security requirement [36, 53] because of the constraints on the system they were using.

2.6 Linear and Superlinear Searchable Encryption

This encrypted search group is called linear and superlinear searchable encryption due to the linear or greater search time in the size of the data set. These include systems built with Fully Homomorphic Encryption; Secure Multiparty Computation; and Oblivious Ram (ORAM). These constructions often provide superior security to SSE and often provide special functionality – but they trade off performance for this advantage.

Fully Homomorphic Encryption (FHE). Creating a searchable encryption scheme utilizing Fully Homomorphic Encryption (FHE) [27] would likely result in a highly secure searchable encryption scheme. Fully Homomorphic Encryption allows operations to be performed on ciphertexts. For example, if there are two plaintext messages, m_1 and m_2, and encryption algorithm E_k and decryption algorithm D_k, ciphertexts $c_1 = E_k(m_1)$, $c_2 = E_k(m_2)$, $m_1 + m_2 = D_k(c_1 + c_2)$, then it is homomorphic under addition. However, a searchable encryption system built solely from FHE will require linear search time.

Secure Multiparty Computation (MPC). Secure Multiparty Computation [63] considers the problem of different parties computing a joint function of their separate, private inputs without revealing any extra information about these inputs other than what is leaked from the result of the computation. With Secure Multiparty Computation, a number of users each have private data $d_1, d_2, ...d_n$ and can compute the value of a public function $F(d_1, d_2, ..., d_n)$ while keeping their inputs secret from the other users. A searchable encryption system built from SMC would have minuscule leakage, achieving a high level of security. However the performance is linear in the size of the database, which is not practical for most uses.

Two recent projects built database systems with MPC [35] and the Jana [3] system from Galois.

Oblivious Ram (ORAM). Oblivious Ram [30] is designed to hide access patterns. Specifically, for any two sequences y, y' of equal length, access patterns $A(y)$ and $A(y')$ are computationally indistinguishable. Access pattern refers to the information that is implied by the query results. For example, one query can return a document x, while the other query could return x and another 10 documents. The size of the return results implies that the predicate used in the first query is more restrictive than that in the second query. Implementing this technique in a searchable encryption construction would provide protection against an active adversary. Like FHE and SMC, ORAM suffers from performance issues that prevent it from being practical on all but very small databases.

2.7 Trusted Third Party Searchable Encryption

This group we call trusted third party searchable encryption, because their constructions require a trusted third party, often a separate index server. However in many instances, the purpose of the third party is to gain some other privacy attribute, such as protecting the identity of the client performing the search. Systems such as Secure Anonymous Database Search [54] and Blind Seer [50] fall into this category.

Secure Anonymous Database Search (SADS). The Secure Anonymous Database Search (SADS) [54] construction requires a separate semi-trusted Index Server to facilitate search. The Index Server (IS) uses a Bloom filter per document built from the encryptions of all words of the document. The encryptions are deterministic, which is the reason for the separate Index Server (IS) to facilitate search. Without this trusted IS server, the index structure would leak.

The SADS construction was improved by Pappas et al. [51], increasing performance and adding range query functionality. But it still requires a trusted index server.

Blind Seer. The Blind Seer [50] project also utilizes a separate index server. This index server stores the Bloom filters in a search tree. Each leaf holds a Bloom filter from a database record. The Bloom filter contains all of the keywords from this record. Each node in the tree stores a Bloom filter that contains all of the keywords of its children.

The search tree for this method is encrypted. To perform searches, the tree is traversed using a secure computation between the client and the index server. This traversal with secure computation requires communication between the client and server for each node traversed in the tree.

2.8 Attacks on Searchable Encryption

All known searchable encryption constructions that are efficient leak some information. This leakage profile varies on the construction, but each type of leakage is vulnerable to attacks.

Encrypted Search Security Models. The security models for encrypted search come in two primary categories. The first is the offline or passive adversarial model. In this model, the attacker receives a snapshot of the encrypted data and nothing else. The second model is the online or persistent adversarial model. With this model we assume the attacker has a presence in the server and not only can see the encrypted data, but also observes the encrypted queries, return results and updates to the encrypted data. Figure 2.3 illustrates the two models.

It is important to distinguish the two models because the security definitions for an encrypted search system will be under one of these two models in most cases. A

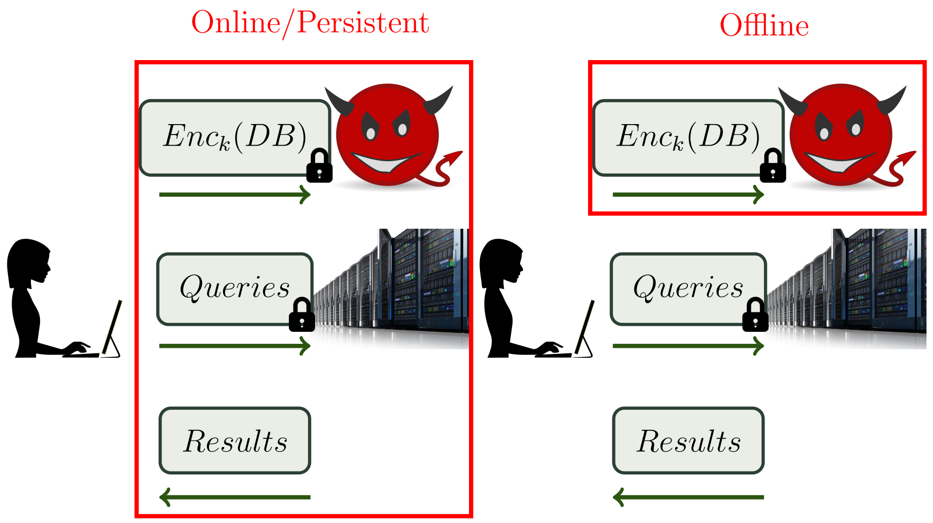


Figure 2.3: Adversarial Models

system may have a proof of security under the passive adversarial model, but be insecure under the persistent model.

Inference Attacks on Encrypted Data. In an inference attack, the adversary uses some outside "auxiliary" information to exploit leakage from a cryptographic construction in order to infer the value of some hidden data. The original statistical inference attack was Al-Kindi's frequency analysis, first proposed in the 9^{th} century AD. It was developed to break classical cryptographic schemes such as substitution ciphers, and it is still widely used to illustrate the weakness of these schemes in introductory cryptography courses. More recent inference attacks have also targeted efficient schemes for storing and searching records in relational databases [49, 55].

Interestingly, a new analysis by Lacharité and Paterson [46] proves that frequency analysis is the maximum likelihood estimator for deterministic encryption.

Islam, Kuzu, and Kantarcioglu (IKK) [38] presented the first inference attack against symmetric searchable encryption. They noted that over time, as more searches are performed in an SSE, the server can see which keywords tend to occur together in the same documents. In the IKK attack, the adversary observes the frequency of co-occurrence for each pair of words in some corpus of training data, then uses this information to map keywords in the encrypted corpus back to plaintext words. IKK prove that the adversary's task is an NP-complete combinatorial optimization problem, but they also demonstrate that simulated annealing can be used to recover most of the top few hundred keywords in under 14 hours.

Unfortunately the IKK experiments use the same data for the target and for the adversary's auxiliary information—this implicitly assumes that the adversary has perfect knowledge of the word co-occurrence frequencies. Cash, Grubbs, Perry, and Ristenpart (CGPR) [14] attempted to reproduce the IKK experiments, and their results show that the accuracy of the IKK attack degrades quickly as the adversary's knowledge of the encrypted corpus decreases. CGPR present new, simpler attacks that outperform the IKK attack but also require knowledge of a large fraction of the target data.

Frequency Analysis and ℓ_p-Optimization. The application of frequency analysis to attack PPE is straightforward. Given an auxiliary corpus of plaintext messages, a target corpus of encrypted messages and their tags, the adversary simply counts the number of times that each keyword appears in the auxiliary data and the number of times each tag appears in the target data. He sorts both the list of keywords $\mathbf{w} = \{\mathbf{w}_1, ..., \mathbf{w}_n\}$ and the list of tags $\mathbf{t} = \{\mathbf{t}_1, ..., \mathbf{t}_n\}$ by their frequency. So, for example, $\mathbf{w}_1$ is the most common keyword and $\mathbf{w}_2$ is the second-most common, and so forth. Finally, the adversary concludes that the i^{th} most common tag corresponds to the i^{th} most common keyword; that is, $\mathbf{t}_i \equiv \mathbf{w}_i$, for all $i \in [1, n]$.

Naveed, Kamara, and Wright [49] use frequency information in an inference attack. Rather than simply matching up plaintexts to ciphertexts in order of decreasing frequency, they pose the problem as a *linear sum assignment problem* to find the optimal matching that minimizes the total difference in frequencies. For $p \geq 2$, they found that ℓ_p-optimization produced results that are identical to frequency analysis; these results strongly suggests that the two approaches may in fact be equivalent.

Both of these previous attacks rely solely on the frequencies of individual plaintexts and ciphertexts. Although they were highly effective against categorical data in encrypted databases, our experimental results in Chapter 4.1.1 show that they are less accurate against natural language data, where the frequencies distribution of plaintext keywords are much noisier and the keyword domains a great deal larger.

Multi-Column Inference Attack. Many previous attacks against encrypted databases attack one column at a time, relying solely on the ciphertext frequencies in a column. Another attack technique utilizes the relationships between columns. In a database, it is likely that the contents of one column are related in some way to the contents of another. For example, in the US census database from 1994, the income column is highly correlated to the eduction column. Once an adversary recovers plaintext data from the income column, they would have a higher chance of guessing the eduction column correctly.

The multi-column inference attack exploits the data relationships between columns in a database. Bindschaedler et al. [6] calls this a *multinomial attack*, since they model the vector of plaintexts as sampled according to the multinomial distribution.

Using the relationships between columns, they show the technique is more accurate than the single column attacks, especially on columns where the data is close to uniformly distributed.

Access Pattern Attacks. In the past few years, research has been published attacking the access pattern leakage that SSE and other systems have. Using just this leakage, attackers attempt to reconstruct the database or search terms. Kellaris, Kollios, Nissim, and ONeill (KKNO) [42] presented the initial research for this type of attack. Their attack required $\mathcal{O}(N^4 logN)$ range queries to fully reconstruct a database where N is the number of possible values in the database. For databases where every possible value is in the data, they improved the bound to $\mathcal{O}(N^2 logN)$. However these values assume that all the queries follow a uniform distribution on the ranges. This performance was further improved by Lacharit, Minaud, and Paterson [47] to $\mathcal{O}(NlogN)$ under the same assumptions.

The practicality of these attacks due to the high number of queries required led to ϵ-Approximate Database Reconstruction [47]. Instead of fully reconstructing the database, the attacker's goal is to find the value of every record up to an error of ϵ. This attack is still very useful to an adversary for small values of ϵ. They require only $\mathcal{O}(Nlog\ \epsilon^{-1})$ queries, but still require the assumption that every possible value is in the database.

Two recent works [34, 45] demonstrated that this type of attack is possible without the restriction of uniform queries. Further improvements by Grubs et al. [34] improved the ϵ attack to only a function of ϵ, $\mathcal{O}(\epsilon^{-4} log\ \epsilon^{-1})$.

These more recent works show that practical attacks are possible using only access pattern leakage. The most obvious defense against this attack is to use ORAM. However the performance characteristics of ORAM are not practical for many applications. A new and open field of research is how to stop these access pattern attacks efficiently.

File Injection Attacks. In file injection attacks, the attacker sends files that it chooses to the client who encrypts and uploads them to the server. SSE constructions without forward privacy will reveal which queries match that documents. Zhang et al. [65] shows an attacker can learn a very high fraction of the keywords searched by the client with file injection attacks.

3 Attacking PPE With Weighted Graph Matching

In this chapter, we demonstrate the vulnerability of Property Preserving Encryption (PPE) constructions to inference attacks. Prior to this work along with similar research [14, 49], consequences of the leakage inherent with PPE constructions was not fully known.

The desire to add encryption to existing applications without losing functionality such as the ability to perform searches has resulted in a few resourceful projects. In order to add end-to-end encryption to legacy applications while retaining the convenience of full-text search, ShadowCrypt [36] and Mimesis Aegis [48] use Property Preserving Encryption (PPE). The Mimesis project coined a new cryptographic technique called “efficiently deployable efficiently searchable encryption” (EDESE) that allows a standard full-text search system to perform searches on encrypted data. However their construction is actually an instance of PPE, since they use a form of deterministic encryption. Compared to other recent techniques for searching on encrypted data, these PPE-EDESE schemes leak a great deal of statistical information about the encrypted messages and the keywords they contain.

To support end-to-end encryption in applications that do not provide encryption support, both ShadowCrypt and Mimesis insert themselves between the user interface and the application. ShadowCrypt targets web applications, taking advantage of the Shadow Dom, built to keep components on a web page isolated from each other. Mimesis creates an Android layer between the application layer and user layer. Both systems encrypt user input before the application receives any data, and conversely decrypt data received from the application before presenting it to the user.

Using unmodified applications and user interfaces creates constraints on the types of encryption used, especially if search functionality is required. To satisfy these constraints and achieve backwards compatibility with legacy systems, PPE encryption constructions are used. These PPE constructions often have an aggressive leakage profile.

To enable a legacy service or application to search on encrypted messages, Mimesis and ShadowCrypt attach to each message a list of identifiers—here we call them "search tags"—that correspond to the set of keywords in the message. Each tag t is computed as a pseudorandom function of a keyword w, using a secret key k. When the user initiates a search for a keyword w, they intercept the user's input and replace the plaintext query with one or more search tags $t = F_k(w)$ before returning control back to the legacy app. On the back end, the legacy application can use any standard full-text indexing technique to keep track of the list of messages that contain each tag. Given a search request for tag t, the legacy app's back end consults its index and returns the encrypted documents to the front end. When the front end goes to display the search results, Mimesis and ShadowCrypt again intercept the user interface to decrypt the messages and display the plaintext to the user.

The practical benefits of this are clearly compelling. It allows users to immediately begin encrypting their communications, without changing providers or losing familiar application user interfaces. This approach also has much lower startup costs compared to other techniques for searching encrypted data.

However there are consequences for using PPE-EDESE as an encryption primitive. These constructions leak information that is very useful to an attacker, such as the relative frequency of each tag in the corpus and the frequency with which tags occurs together in the same message.

In contrast, with a conventional SSE scheme, the adversary is only allowed to learn the relationship between a tag and a document when the user performs a search for corresponding keyword.

This additional leakage provides the opportunity for statistical inference attacks whereby the adversary uses known word frequencies to match the observed tags back to the keywords that they represent.

A key insight underlying our approach is that the ShadowCrypt adversary's task reduces to well-known combinatorial optimization problems based on graph matching: *weighted graph matching* and *labeled graph matching*. Although these graph matching problems are in NP, there exist several efficient solvers that can find good approximate solutions in polynomial time. Another key observation of our work is that constructions such as Mimesis that use Bloom filters for encrypted search must be careful in how they configure the Bloom filter's parameters. If the filter parameters are chosen carelessly, or with only efficiency in mind, then the adversary can use an additional pre-processing step to apply the graph matching attacks against the bits in the Bloom filter.

Using real email and chat data, we show in Chapter 3.2.1 how these solvers can be used to efficiently and accurately recover the list of keywords for messages encrypted with Mimesis and ShadowCrypt. For example, for several users in the Enron email corpus, the attack can recover more than 900 of the top 1000 most common keywords. In a corpus of chat messages from the Ubuntu Linux project, it recovers more than half of the top 500 keywords. Recovering so many of the top keywords would enable the adversary to perform a variety of interesting analyses on the encrypted documents, such as grouping similar documents together in clusters or identifying the *sentiment* (positive-negative, happy-sad-angry) expressed in each message.

Our threat model is conservative, in that we give the adversary access to only the ciphertext messages and the search tags. Unlike the standard adversary model for

SSE, our adversary cannot issue queries and has no control over the plaintext data.

To mitigate against our attacks, we demonstrate and evaluate a new strategy based on careful tuning of the Bloom filter parameters to reduce the information leaked by the tags. Experimental results show that an efficient choice of parameters is sufficient to break our current attack, while better protection is possible at the cost of increased space overhead. However, we caution that our defense does not eliminate the information leakage entirely. Given a sufficient amount of data, it is still possible that a clever adversary might be able to reverse-engineer the Bloom filter.

Beyond the immediate impact to efficiently searchable encryption, our results here may have implications for the security of other encrypted search systems that use Bloom filters, such as SADS [51, 54] and BlindSeer [25, 50], and for systems that perform symmetric searchable encryption over natural language documents [15, 21, 29].

3.1 Graph Matching

Weighted Graph Matching.

The weighted graph matching (WGM) problem is a well known combinatorial optimization problem that has been studied for nearly 30 years [20]. Given two edge-weighted graphs G and H with n nodes each, the problem is to find the permutation that re-labels the nodes in H so that the permuted graph most closely resembles G. More formally, let $A_G = [g_{ij}]$ and $A_H = [h_{ij}]$ be the adjacency matrices of G and H, respectively. Here, $g_{ij} \geq 0$ gives the weight of the edge connecting nodes i and j in G, and $h_{ij} \geq 0$ gives the weight in H. Further, let X be an $n \times n$ permutation matrix, and let $A'_H = XA_HX^T$ be the adjacency matrix for the permuted version of H, with edge weights h'_{ij}. The goal of the optimization problem is then to find the permutation matrix that minimizes the matrix distance between A_G and A'_H. For example, using the Euclidean distance as our matrix distance, the problem can be stated as

$$\begin{aligned}
\textbf{minimize} \quad & ||A_G - XA_HX^T||_2 \; = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n}(g_{ij} - h'_{ij})^2} \\
\textbf{subject to} \quad & \sum_{i=1}^{n} X_{ij} = 1, \qquad 1 \leq j \leq n \\
& \sum_{j=1}^{n} X_{ij} = 1, \qquad 1 \leq i \leq n \\
& X_{ij} \in \{0, 1\}, \qquad 1 \leq i, j \leq n.
\end{aligned}$$

The WGM problem is in NP. There exist many algorithms for efficiently finding approximate solutions, including an influential 1988 paper by Umeyama [59] that uses eigendecomposition of the adjacency matrices to find a nearly-optimal solution in $O(n^3)$ time. Umeyama's algorithm works especially well when the two input graphs are nearly perfectly isomorphic. The PATH algorithm [64] is more robust, using an adaptive path-following strategy; it also runs in $O(n^3)$ time, but with a larger constant factor than Umeyama's algorithm. A powerful linear programming (LP) technique from Almohamad and Duffuua [1] has complexity $O(n^7)$, so we do not consider it for use in practical inference attacks.

Labeled Graph Matching. Labeled graph matching (LGM) is a further generalization of WGM. Whereas in WGM the similarity of two graphs is computed as a function of their edge weights, in LGM the nodes may also have weights. The best matching is the one that simultaneously minimizes the difference in edge weights while maximizing the similarity of the node weights. For example, if C_{ij} gives the similarity of node i's weight in G with node j's weight in H, and $\mathbb{P}$ is the set of all permutation matrices, then the permutation that maximizes the similarity of the node weights is

$$\max_{X \in \mathbb{P}} \operatorname{tr}(C^T X) = \max_{X \in \mathbb{P}} \sum_{i=1}^{n} \sum_{j=1}^{n} C_{ij} X_{ij}.$$

A natural way to include both the edge weights and the node weights in a single objective function is with a simple linear combination where the parameter α designates the weight given to the terms of the linear combination. The full optimization problem can then be stated as

$$\begin{array}{rll} \textbf{minimize} & (1-\alpha)\ ||A_G - XA_HX^T||_2 - \alpha\ \operatorname{tr}(C^T X) & \\ \textbf{subject to} & \displaystyle\sum_{i=1}^{n} X_{ij} = 1, & 1 \le j \le n \\ & \displaystyle\sum_{j=1}^{n} X_{ij} = 1, & 1 \le i \le n \\ & X_{ij} \in \{0,1\}, & 1 \le i,j \le n. \end{array}$$

Both the Umeyama algorithm and PATH can be easily adapted to solve the labeled graph matching problem. The GraphM software package [64] includes efficient implementations of these and other algorithms.

3.2 Attacks on ShadowCrypt

The idea of using word co-occurrence frequencies for inference attacks against symmetric searchable encryption was first proposed by Islam, Kuzu, and Kantarcioglu [38]. Here we improve on this attack strategy for EDESE and formalize it as an instance of the graph matching problems described above. We also compare our attack to previous inference attacks, including classical frequency analysis and the ℓ_p-optimization technique from outlined in Chapter 2.8 from Naveed et al. [49].

3.2.1 Graph Matching Attacks

We now give the polynomial-time reduction of the inference attack on EDESE to the graph matching problems, WGM and LGM. Given a plaintext corpus for use as the adversary's auxiliary information and an EDESE-encrypted corpus as his target, the attacker first removes the most common "stop" words (e.g. *a, the, and, of*, ...) from the auxiliary data, because the victim system almost certainly stripped them from the target data before generating the tags. Then the adversary selects the top n most common remaining keywords $\mathbf{w} = \{\mathbf{w}_1, ..., \mathbf{w}_n\}$ from the auxiliary data and the top n most common tags $\mathbf{t} = \{\mathbf{t}_1, ..., \mathbf{t}_n\}$ from the target data. They create two graphs, G and H, to represent the auxiliary and target data, respectively, as follows. For each $i, j \in [1..n]$, they set the weight of the edge g_{ij} in G to be the probability, over the auxiliary corpus, that keywords $\mathbf{w}_i$ and $\mathbf{w}_j$ occur in the same document. Similarly, they set the weights h_{ij} in H to be the probability, over the target data, that tags $\mathbf{t}_i$ and $\mathbf{t}_j$ are attached to the same encrypted document. This technique is sufficient to reduce the attack to the WGM problem.

To yield an instance of the LGM problem, the adversary must create a similarity matrix C for the nodes. Intuitively, each cell C_{ij} in the matrix should give the similarity of the frequency of word $\mathbf{w}_i$ compared to the frequency of tag $\mathbf{t}_j$. There are many ways to capture this similarity. For example, we might set the node weights similar to Naveed et al. [49] to minimize the overall difference in frequencies. Here, we opt instead for a slightly different approach based on the method of maximum likelihood. Let g_i be the fraction of auxiliary documents that contain word $\mathbf{w}_i$ and h_j be the frequency of tag $\mathbf{t}_j$ in the target data. Let D be the number of documents in the target. Let $k_j = h_j \cdot D$. Then the adversary sets the similarity C_{ij} as the likelihood that word $\mathbf{w}_i$ appears in k_j out of D documents. The permutation matrix X that maximizes the objective function is therefore the maximum likelihood solution.

$$\begin{aligned}
D &= \text{Number of Documents} \\
g_i &= \text{Frequency of Word } \mathbf{w}_i \\
h_j &= \text{Frequency of Tag } \mathbf{t}_j \\
k_j &= h_j \cdot D \\
C_{ij} &= \text{Binom}(k_j, D, g_i) \\
C_{ij} &= \binom{D}{k_j} (g_i)^{k_j} (1 - g_i)^{D-k_j}
\end{aligned}$$

The adversary solves the graph matching problem to find the optimal permutation matrix X' that most closely maps H to G. They then apply the same permutation to the list of tags to obtain the permuted list $\mathbf{t}' = X'\mathbf{t}$. Finally, the adversary concludes that each tag $\mathbf{t}'_j$ in the encrypted corpus represents keyword $\mathbf{w}_j$ from the auxiliary data, for all $j \in [1, n]$.

3.3 Empirical Evaluation

ShadowCrypt and Mimesis aim to support email and other messaging applications, including Gmail, Twitter, WhatsApp, and others. To evaluate the practical impact of our attacks, we use two data sets of real email and chat messages. The Enron email corpus [44] includes real emails from the mailboxes of 150 employees of Enron Corporation, received between 2000 and 2002. It was originally made public as part of the federal government's investigation into the company's collapse, and it has since been used in several studies on the practicality of searchable encryption schemes [40] and the effectiveness of inference attacks [14, 38]. The Ubuntu Chat Corpus [60] is composed of archived chat logs from Ubuntu's Internet Relay Chat technical support channels. This corpus comes from the logs between July 2004 and October 2012. Table 3.1 summarizes the data sets that we use to evaluate our attacks on real data.

Corpus	**Type**	**Date**	**Messages**	**Mean Keywords per Message**
Enron	Email	2000–2002	517446	101
Ubuntu	IRC chat	2004–2012	26360715	6.57

Table 3.1: Email and Chat Corpora

3.3.1 Initial Experiments

For a direct comparison with prior work that tests and trains on the same data [38] or that gives the adversary access to the plaintext corpus [14], we performed a small initial experiment with the Enron corpus. We note that normally, testing and training on the same data is considered exceptionally bad practice. However, for tools such as ShadowCrypt and Mimesis Aegis, there is one real scenario where this might give an appropriate model for the adversary's capabilities. Suppose a user has a large corpus of messages stored on a service such as Gmail, and they decide to encrypt all of their old messages using EDESE. At the moment when they finish uploading the encrypted messages, the server has perfect knowledge of both the old plaintext corpus and its new EDESE tags. Since the server already has the old plaintext corpus, the point of performing the attack at this stage is to learn information about the tags. If the server can match tags to keywords at that point in time, it can recover the keywords in each new encrypted message almost for free.

In this initial experiment, we divide each Enron user's mails randomly into a training set and a testing set. Here we ignore the testing set, and we use the training set as both the adversary's auxiliary information and the target data. We use the training set to construct an adjacency matrix as described in Chapter 3.2.1 and we use this matrix as both A_G and A_H. We use the open source `graphm` tool with the Umeyama and PATH algorithms to find the permutation that most closely matches A_G to A_H.

In this easy attack scenario, the weighted graph matching attack performs extremely well. The Umeyama WGM algorithm achieves perfect 100% accuracy for every user in the corpus, as does simple frequency analysis. This is to be expected, as both algorithms are optimized for the case where the auxiliary and target data have very few differences. The Umeyama algorithm matches each pair of graphs in under 40 seconds. The PATH algorithm is designed to handle greater variation in the graphs, so it runs roughly two orders of magnitude slower than Umeyama. Its accuracy on this experiment is also somewhat reduced compared to the naive algorithms. Figure 3.1 shows the complementary cumulative distribution function (CCDF) of the PATH WGM algorithm's accuracy across all 150 users in the Enron data set.

A point at position (x, y) on the graph means that the attack correctly matched at least x% of the keywords for y% of the users in the corpus. The attack recovers about 95% of the keywords for more than 90% of the users, with some slight degradation in accuracy, as we expand the attack to target a larger number of keywords.

Previous attacks on standard SSE require *a priori* knowledge of both target corpus and some number of the queries. Islam, Kuzu, and Kantarcioglu report that their simulated annealing attack could analyze up to 150 queries and 2500 keywords in under 14 hours, and it could recover more than 80% of the queries. That is, the IKK attack can recover between 120 and 150 of the top keywords, as long as they are used in a query by some user. Given a similar experimental setup with 10% of queries known *a priori*, Cash et al.'s *count attack* recovers 100% of the queries. The accuracy of our attack is similar to the related work, even when we have no known queries. But because EDESE gives us access to the tags for *all* the keywords, the practical

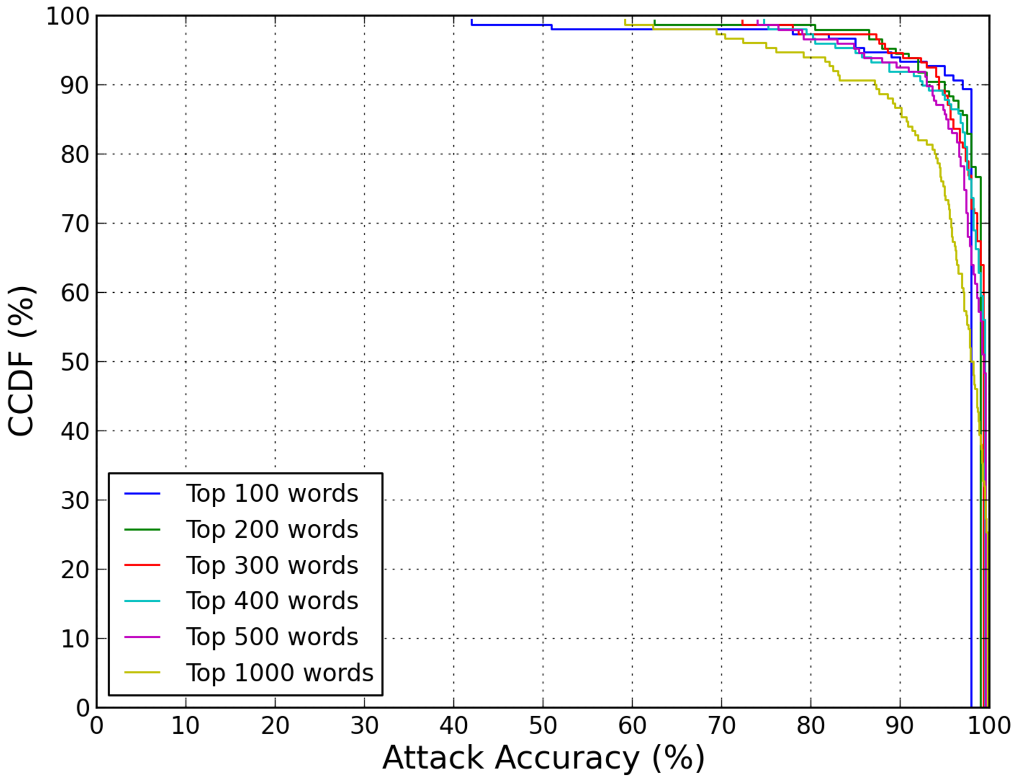


Figure 3.1: Accuracy of weighted graph matching attack (PATH algorithm) against ShadowCrypt for users in the Enron email corpus, using perfect auxiliary information

impact is greater. Whereas a 95%-accurate attack on SSE might retrieve more than 140 keywords, our attack on EDESE recovers more than 950.

On the other hand, these results may give an overly pessimistic estimate of the security of SSE and EDESE. When given access to only 50% of the target data, both of the attacks from prior work achieve accuracy very near to zero (c.f. Fig. 6 in Grubs et al. [14]). Next we look at what happens when we run our attack with no access to the target data.

3.3.2 Experiments with Imperfect Auxiliary Info

Here we consider a more realistic scenario, where the adversary does not have any specific knowledge of the messages in the encrypted corpus, but they still have very good estimates for the keyword frequencies. We conducted experiments in this model using data from the Enron email corpus and the Ubuntu chat corpus.

For each user in the Enron corpus, we randomly divided the user's emails into two non-overlapping sets. We took one half of the user's emails as the training set and used them to construct the adjacency matrix A_G for the adversary's auxiliary information. We took the other half of the user's emails as the test set and used them to construct the adjacency matrix A_H for the target data. We did this for several values of n between 100 and 1000.

To match the top n keywords, we first ran the attack using Frequency Analysis and the Weighted Graph Matching attack with the Umeyama algorithm [59]. Figure 3.2 shows the complementary cumulative distribution function (CCDF) of the accuracy for the Frequency Analysis attack and the Weighted Graph Matching with the Umeyama algorithm. For the top 500 words, the accuracy for these two attacks is less than 10% for almost all users.

We then performed the Weighted Graph Matching attack with the PATH algorithm [64]. Figure 3.3 shows the (CCDF) of the attack's accuracy across the 150 users in the Enron corpus. Over all, the accuracy of the attack decreases as we increase the number of keywords targeted. But even when attempting to match the top 1000 words, the adversary still achieves over 90% accuracy for about 10% of the users. To reiterate—for these 15 unlucky users, the adversary recovers more than 900 of the top 1000 words in their email. If the adversary is only interested in the top 200 words, they achieve greater than 80% accuracy for half the users in the corpus. Note that,

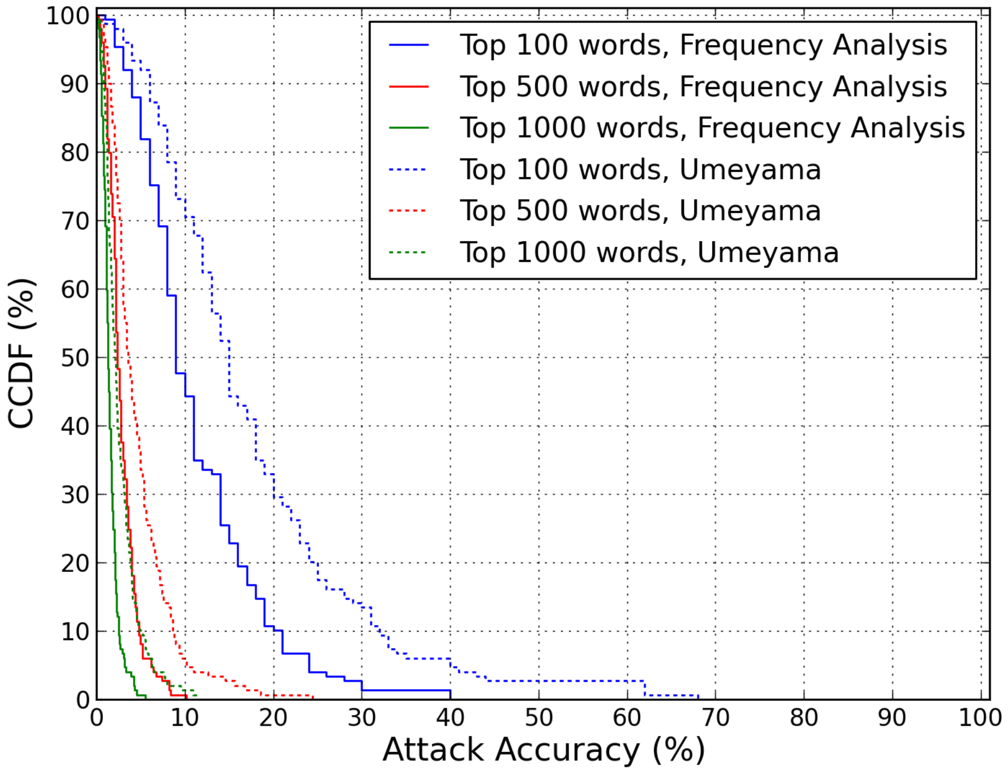


Figure 3.2: Accuracy of Frequency Analysis and Weighted Graph Matching (Umeyama) attacks for Enron data, with imperfect auxiliary information

unlike previous work [14, 38], this attack succeeds given *no access to the target data*, *zero known keywords*, and *zero known documents*.

Further work will be required to understand why the attack's effectiveness varies so much from user to user. Our working hypothesis is that the variation stems from differences in the users' *topic model*. Like the earlier IKK attack, our adversary assumes that the probability of seeing each word—or each pair of words—in a document is constant across the entire corpus. For natural language text, this assumption does not really hold. Instead, in more accurate models of text, such as latent Dirichlet allocation [7], the distribution of words is fixed for each of several *topics*, and the mix of topics can vary greatly from document to document. We suspect that users for

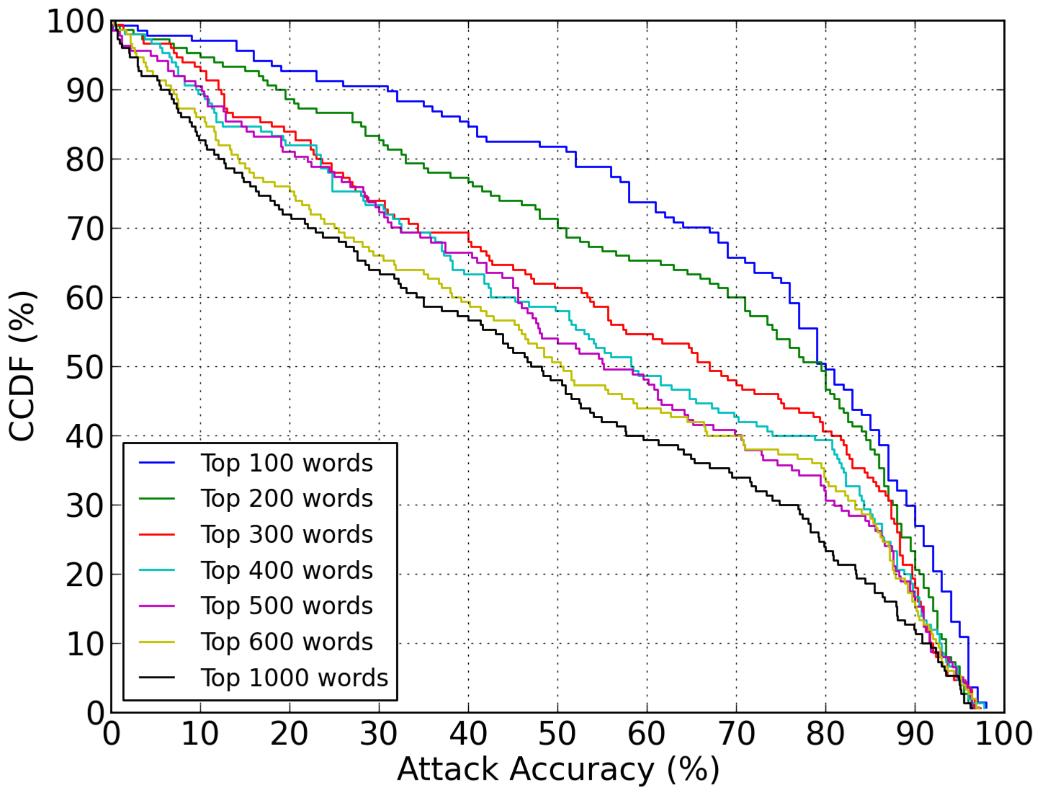


Figure 3.3: Accuracy of Weighted Graph Matching attack (PATH Algorithm) against ShadowCrypt for Enron data, with imperfect auxiliary information

whom the attack is very successful have a more stable distribution of topics in their email. It might be possible for a future attack to learn both the topic model and the word frequencies at the same time.

Following a similar procedure, we ran the experiment for each month of IRC chat logs from the Ubuntu corpus. We randomly assigned each day in the month to either the adversary's auxiliary information (i.e., the training set) or the target data (i.e., the test set). For each value of n, we created the adjacency matrices A_G and A_H as above, and we ran the `graphm` experiment for each month of the Ubuntu corpus.

Figure 3.4 shows the results for the Ubuntu experiment. Compared to the email data, overall the adversary's accuracy degrades more quickly as we increase the number of

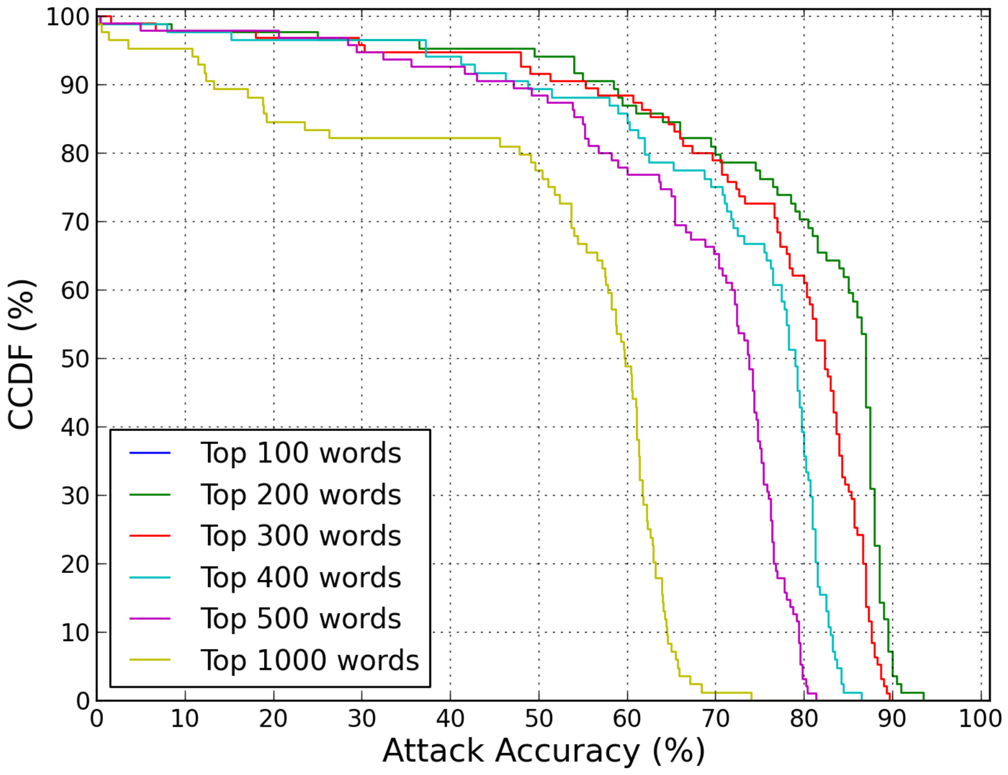


Figure 3.4: Accuracy of Weighted Graph Matching attack (PATH Algorithm) for Ubuntu data

keywords targeted, but the attack is still many times more accurate than random guessing. For 10 percent of the months, the adversary correctly recovers almost 400 of the top 500 keywords. They recover more than half of the top 500 keywords for nearly 90% of the corpus.

3.3.3 Runtime Performance

We ran all experiments on a cluster of HP Proliant servers with Intel Xeon L5520 processors at 2.26GHz, running CentOS Linux 6 and version 0.52 of the `graphm` software. The Umeyama algorithm required less than 40 seconds to run each attack in Chapter 3.3.1 with perfect 100% accuracy. However, its accuracy was substantially reduced when testing and training on different data.

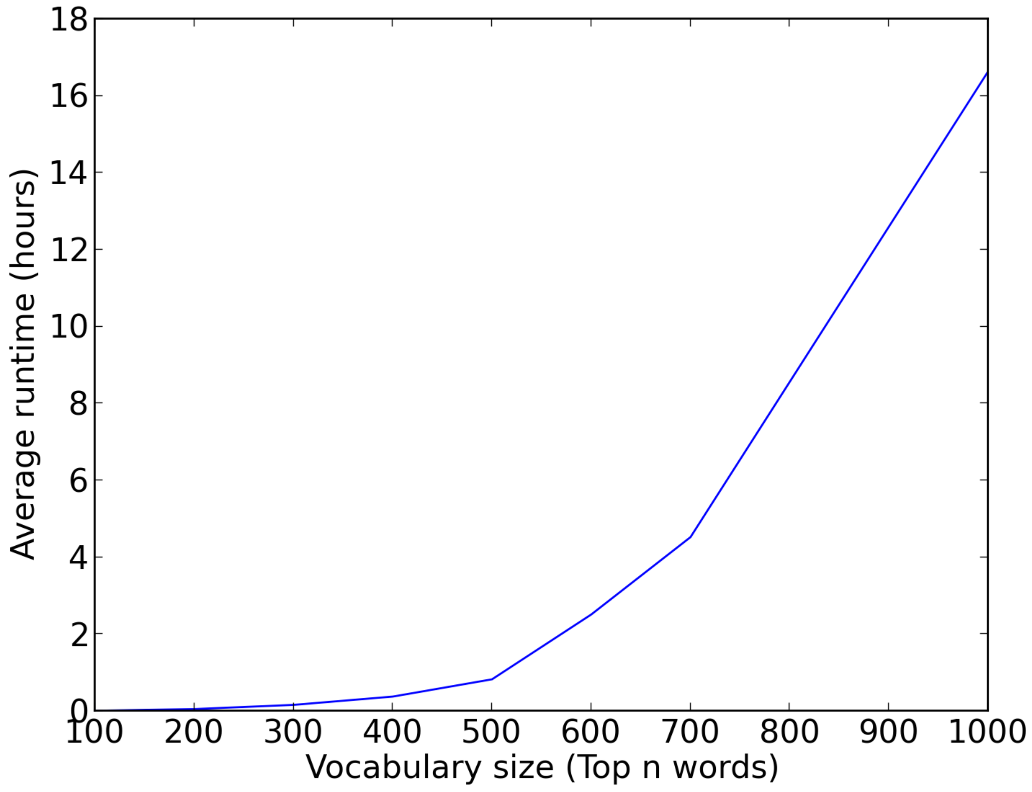


Figure 3.5: Runtime performance of the Weighted Graph Matching attack (PATH Algorithm) for the Enron email corpus

Figure 3.5 shows the average runtime for matching the top 100, 200, ..., 1000 keywords for each Enron user with the PATH algorithm. Matching the top few hundred keywords is very fast; even our older 2009-era CPUs can match the top 500 words for a user in under one hour. The Umeyama and PATH algorithms are $O(n^3)$ in the number of keywords to be matched, so attacking thousands of keywords becomes increasingly expensive. However, matching several thousand keywords would not be beyond the capacity of a large corporation or a nation state. It is also possible that a much faster solver could be implemented using graphics processing units or other specialized hardware. Memory does not appear to be a limiting factor: even when matching the top 1000 keywords, the `graphm` process uses less than 250MB of memory.

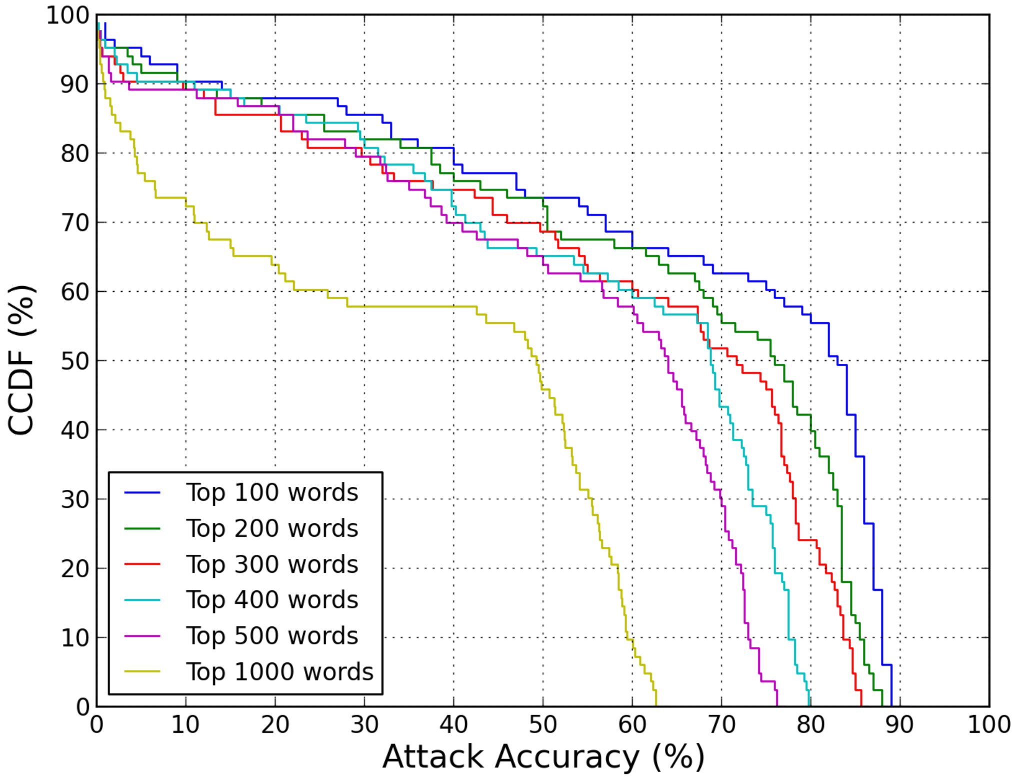


Figure 3.6: Accuracy of Weighted Graph Matching attack (PATH Algorithm) for Ubuntu data; 1 month delay between auxiliary and target

3.3.4 Experiments with Time Delay

Figure 3.6 shows the accuracy of our attack when the adversary's auxiliary information is from the previous month before the target data. Comparing this graph to Figure 3.4, the attack performance is considerably lower. The cause for this degradation is likely due to the differences in topics, and thus words from one month to the next. The result is our auxiliary training data is not as close to the actual data compared to the attack on messages within the same month.

3.4 Attacks on Mimesis

Mimesis is a more difficult target than ShadowCrypt. We cannot apply our graph matching attack directly, because Mimesis does not reveal a one-to-one correspondence between keywords and tags. Where ShadowCrypt uses a single PRF to generate a single search tag for a given keyword, Mimesis uses a family of h PRFs and generates up to h distinct tags for each keyword. If the Bloom filter is sufficiently small, there may be some collisions in the PRFs, so some tags may correspond to more than one keyword.

As we will show in Chapter 3.5, the Mimesis construction provides the opportunity to make inference attacks much more difficult by carefully tuning the parameters of the Bloom filter. On the other hand, a naive choice of Bloom filter parameters such as those proposed by Lau et al. [48] allows the adversary to mount the same graph matching attack with only a small amount of additional work and a variable decrease in accuracy.

Our inference attack on Mimesis proceeds in two steps. First, we analyze the Bloom filters to identify sets of bits that likely represent plaintext keywords. Then we use the graph matching attack to match each set of bits to the best-fitting keyword.

3.4.1 Recovering tags from Bloom filters

Our general strategy to discover the groups of bits that represent keywords in the Bloom filters begins with a simple frequency-based analysis. If the Bloom filters use h hash functions, then for each keyword we expect to see a group of h bits that (1) have the same bit counts where the bit count is the number of documents that the bit is set to one and (2) appear together in the same set of documents. For example, if a keyword w sets the bits 10, 20, 30 and 40, we expect each of these bits to have similar counts and appear together in the same documents. To find these bits for each keyword, we begin by counting the number of documents in which each bit is set, and we group together all the bits that have the same count.

For example, suppose we have a collection of Bloom filters with the parameters used in Mimesis Aegis: $m = 2^{24}$ bits and $k = 10$ hash functions. Figure 3.7 gives some example counts that arise for one Enron user with these parameters. Each row represents a set of bits that all occur in the same number of documents. The first column gives the count of the documents where these bits appeared, and the 2nd column gives the number of bits with that count.

Doc Count	Set Size
238	10
226	10
219	11
212	9
211	10
206	10
186	10
173	10
169	10
143	10
129	1
113	10
101	10
99	9
98	10
89	10
87	10
84	20
82	10
81	10
80	20
79	40

Figure 3.7: Example Bloom filter counts

Exact Matching. Sometimes it is easy to identify the bits for many keywords. In our example, there are exactly 10 bits that appear in 238 documents, another 10 bits that appear in 226 documents, and other sets of 10 that occur in 211, 206, 186 documents, respectively. It is very likely that these five sets of 10 bits correspond to five keywords that appear 238, 226, 211, 206, and 186 times in the plaintext corpus. Similarly, the other sets of 10 bits probably represent one plaintext keyword each. Each of these sets of 10 bits are equivalent to the tags from Chapter 3.2.

Algorithm 3.1 Bloom Filter Tag Extraction

```
Let $S$ be the set of sets of bits
Let $D$ be the set of encrypted documents
Let $B$ be the set of Bloom filters $B = \{B_d : d \in D\}$
Let $k$ be the number hash functions in each bloom filter

function FindTags($S$, $D$, $B$, $k$)
    Let $T \leftarrow \varnothing$                    ▷ $T$ will be the set of extracted tags
    for $b \in S$ do
        if $|b| == k$ then
            $T \leftarrow T \cup b$
        else
            $T \leftarrow T \cup \text{Split}(b)$
    return $T$

function Split($b$)
    for $d \in D$ do
        Let $s_1 \leftarrow b \cap B_d$
        Let $s_2 \leftarrow b \setminus s_1$
        if $|s_1| == k$ then return $\{s_1\} \cup \text{Split}(s_2)$
        else if $|s_2| == k$ then return $\{s_2\} \cup \text{Split}(s_1)$
        else if $k < |s_1| < |b| - k$ then return $\text{Split}(s_1) \cup \text{Split}(s_2)$
        else
            Continue
    return $b$
```

Other sets of bits likely include multiple plaintext keywords. For example in Figure 3.7, there are 20 bits that appear in 84 documents and 40 bits that appear in 79 documents. These sets probably represent two keywords that each appear 84 times and four keywords that each appear 79 times. We can identify the 10 bits that correspond to each distinct keyword if we can find an encrypted document that contains the given keyword but none of the other keywords that have the same count. Algorithm 3.1 gives a more formal specification of our technique for finding such a document.

In some cases there is no such document in the encrypted corpus, and so our algorithm fails to split a larger set of bits into individual keywords. This pattern means that the words in that set always appear together in the same documents, and therefore they will be indistinguishable under the graph matching attack anyway. The important thing is that we have identified a set of keywords that always appear together; this is sufficient for setting up the graph matching attack.

Inexact Matching. Finally, there are some sets whose sizes are not nice multiples of h. Some sets have extra bits, and some sets appear to be missing bits. In our example, there is only one bit with a count of 129, and the set with count 99 has only 9 bits in it. It is likely that we are seeing the results of a collision in one of the PRFs. The bit with count 129 probably belongs with the bits in the set with count 99, representing a plaintext keyword that appears in 99 documents. This same bit must also go with one or more other keywords that collectively occur in 30 other documents to bring its total count up to 129.

In cases like this example, it is tempting to treat the set of 9 bits as a "good enough" match for a plaintext keyword. Then we can ignore the left-over singleton bits such as the one above. But how likely is it that those 9 bits are a real word and not a false positive? To evaluate this likelihood, recall formula 2.1, the probability that a specific bit is set to one after all elements are entered into the Bloom filter:

$$PR[bit_i = 1] = 1 - (1 - \frac{1}{s})^{ht} \approx 1 - e^{\frac{-ht}{s}}$$

Also recall formula 2.2, the false positive rate for Bloom filters:

$$p^h \text{ where } p = PR[bit_i = 1]$$

We can modify this formula to calculate the false positive rate when we only require a match on $\ell \leq h$ bits:

$$(1 - e^{\frac{-ht}{s}})^{\ell}$$

Mimesis uses a Bloom filter with $s = 2^{24}, h = 10$. The Enron emails contain on average $t = 101$ unique keywords. The false positive rate with these parameters when matching on 10 bits $= 6.25 \times 10^{-43}$. The false positive rate with 9 bits $= 1.04 \times 10^{-38}$ and 8 bits $= 1.72 \times 10^{-34}$.

It appears that we can safely create a node in our graph matching step whenever we find a group of ℓ bits that tend to appear together, even if ℓ is smaller than h. We present experimental results for 8 and 10 bits in Chapter 3.4.2.

Graph Matching on Extracted Tags. After finding the sets of bits that we believe correspond to each of the top keywords in the corpus, we again use the graph matching attack to match each set of Bloom filter bits to its best-fitting plaintext keyword.

3.4.2 Empirical Evaluation

We evaluate our attacks on Mimesis in two parts. First, we measure the ability of the tag recovery algorithm to extract the correct set of bits from the Bloom filter for each keyword. Then, we measure the accuracy of the graph matching attack when its list of tags comes not from ground truth, but from the (possibly incorrect) set of tags extracted by the attack on Mimesis.

We created a Bloom filter for each email in the Enron corpus. The Bloom filter parameters we used are the same parameters from [48], $s = 2^{24}$ bits and $h = 10$ hashes. HMAC-SHA-256 is the PRF, with each hash in h receiving a unique key.

3.4.3 Bloom Filter Attack

An attack on the Bloom filter is successful when a set of h bits is successfully identified as a tag for the corresponding keyword. Figure 3.8 shows the average accuracy of our attack across all users in the Enron corpus changes as we increase the number of words targeted. With the majority of users, we were able to recover over 80% of the tags up to the top 1000 words used.

In addition to our experiments that required us to identify all 10 bits for each tag, we also ran experiments relaxing this restriction. The loosened restriction means if we can identify as few as 8 bits that have similar counts and belong to the same set of documents, those 8 bits are considered a tag. Figure 3.8 compares the tag finding accuracy on the Enron corpus matching 10 bits for each tag compared to the less restrictive matching on 8 bits. The dashed lines show the accuracy of the top 10% of the attack results against Enron accounts from the test. The dotted lines reflect the bottom 10% of the attack results.

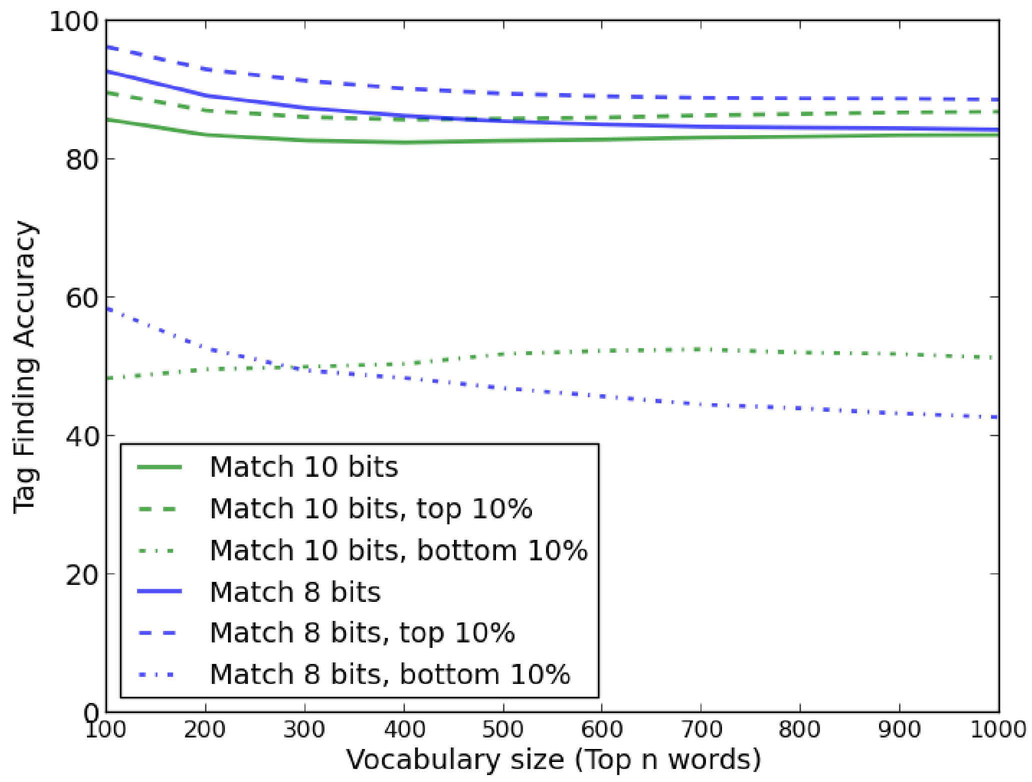


Figure 3.8: Tag Finding accuracy on Enron corpus Bloom filters with parameters: h=10 and Bloom filter size = 2^{24}, matching on 8 and 10 bits

3.4.4 Graph Matching Attacks

The real evidence regarding whether matching on less than k bits is effective, is to compare the results from graph matching. Figure 3.9 shows the `graphm` accuracy of the ShadowCrypt tags compared to the accuracy of the Mimesis tags matching on all 10 bits versus matching on 8 bits. As with the previous graph, the dashed lines show the accuracy of the top 10% of the attack performances from the test. The dotted lines reflect the bottom 10% of the attack performances.

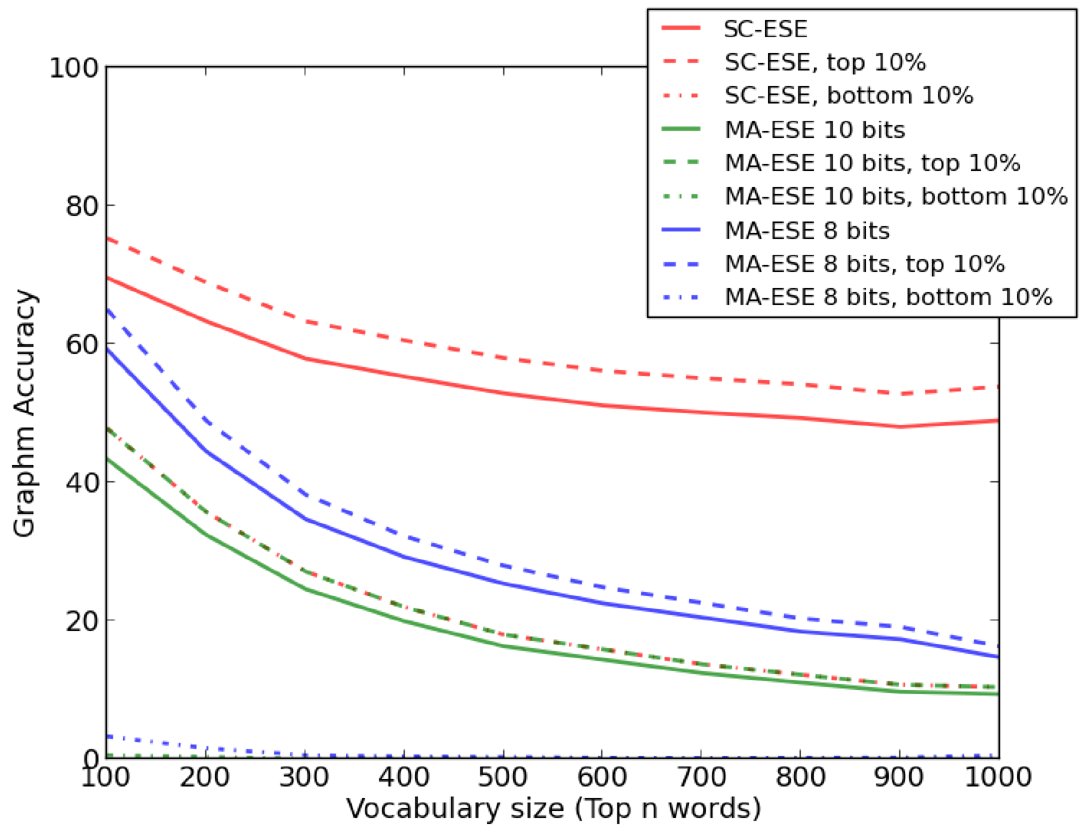


Figure 3.9: Accuracy of Weighted Graph Matching attack (PATH Algorithm) for Mimesis and ShadowCrypt on Enron Data

The accuracy on these `graphm` attacks for the Bloom filters appears to degrade quickly between the 100 and 300 word count. Discovering an improved tag finding attack would certainly help improve the attack accuracy. We believe it also makes a significant difference which tags are not found from the tag finding attack. A tag that has a high rate of occurrence will have a larger effect on the `graphm` algorithm than a tag with a lower rate. Missing the tag with the highest occurrence would certainly have an increased adverse effect compared to missing the tag with the 100th highest count.

Future work might involve analyzing the relationship among the occurrence counts of the tags that are not found from the graph matching and working on algorithms to maximize the attack on discovering those specific tags.

The success rate of the `graphm` attack with the tags recovered from the Bloom filter is fairly low with a vocabulary size of 1000. However, even an attack that has only a 10% success rate is still much better than random guessing and reveals much to an adversary.

3.5 Mitigation

The obvious effective defense against our attacks is to use an encrypted search construction that reveals much less information to the adversary, e.g. SSE [15]. However, for many real use cases the operational requirements only admit efficiently searchable schemes [32]. Here we describe a novel strategy for defending EDESE schemes against inference attacks by carefully tuning the Bloom filter parameters.

When deciding which Bloom filter parameters to use for encrypted search, many previous works [29, 48, 54] discuss how the parameters affect the false positive rate. Until now, little or no attention has been paid to how the choice of parameters may affect security. Our experiments in the previous section demonstrate that the parameters used in Mimesis Aegis are susceptible to attack. Our analysis in this section reveals that the attacks are possible because the Bloom filter used in Mimesis is much larger than necessary. With careful tuning of the parameters, we can make inference attacks much more difficult.

Broder and Mitzenmacher [13] describe a technique for picking the optimal number of hash functions in a Bloom filter to minimize its false positive rate. Given a Bloom filter with s bits and a document length of t keywords, the FP rate is minimized at

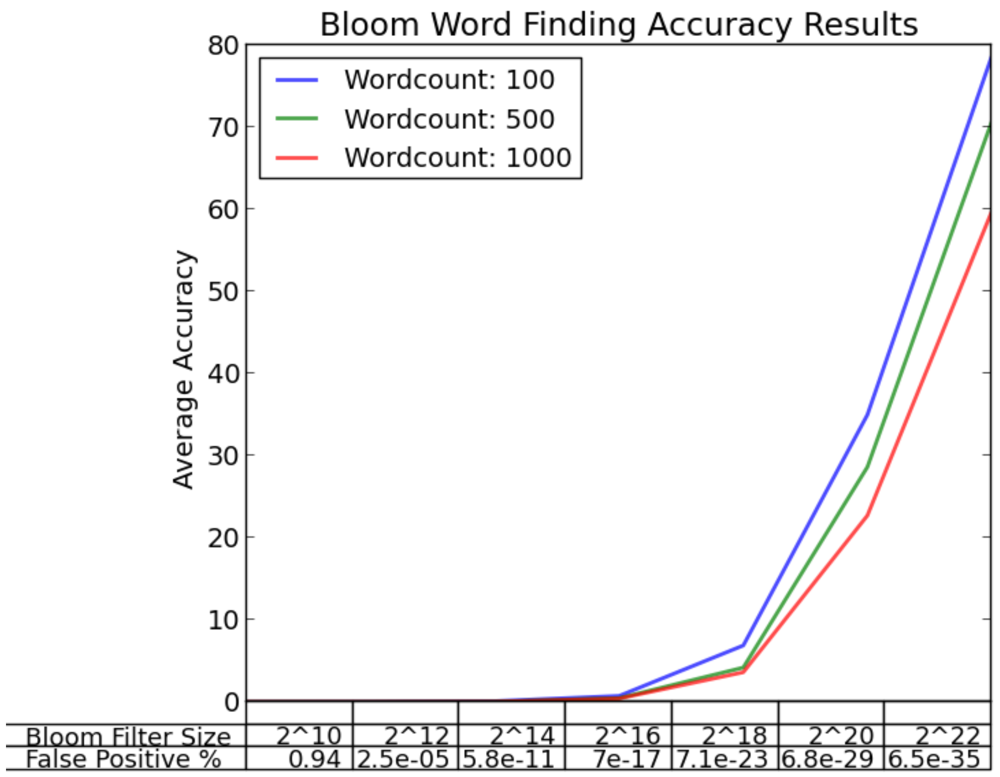
$$h = ln(2 * (s/t))$$

As a side effect of this parameter choice, it happens that each bit in the Bloom filter will be set with probability 50% in each document. Intuitively, this will make it more difficult for the adversary to extract information about which keywords appear in which documents.

After removing stop words, the Enron corpus has an average of 101 unique keywords per document. Applying this formula with $s = 2^{24}$ and $t = 101$ results in an optimal value of $h = 115,139$. Clearly, using more than 100,000 hash functions is not practical. So instead we looked at modifying s, the size of the bloom filter. Applying the same formula, but instead using $h = 10$, $t = 101$ and solving for s, we get a value in between 2^{11} and 2^{12}.

To test this approach, we re-ran our tag finding attack against the Enron corpus, using a constant value of $h = 10$ hash functions, but varying the size of the Bloom filter from 2^{10} to 2^{22}. For each configuration, we computed the average accuracy of the tag finding attack and the expected false-positive rate offered by the Bloom filter. The results of this experiment are shown in Figure 3.10. Setting the size of the Bloom filter close to the value derived from the formula above is very effective in reducing the accuracy of the attack. For $s = 2^{12}$, the attacker is unable to find the tags for even the top 100 most common keywords. Without tags, the attacker cannot even attempt the graph matching attack. Moreover, with these parameters, the Bloom filter still offers a very low false positive rate of 2.5×10^{-5}. A side benefit of this approach is that it is also more space efficient. With a Bloom filter of only 2^{12} bits, each tag can be much smaller than in the default Mimesis configuration.

But as s grows large relative to $h = 10$, the success of our attack grows very quickly. With 2^{22} bits, we can find tags for more than 60% of the top 1000 keywords. At the same time, the tags must also grow to encode more bits.



Bloom Filter Size	2^10	2^12	2^14	2^16	2^18	2^20	2^22
False Positive %	0.94	2.5e-05	5.8e-11	7e-17	7.1e-23	6.8e-29	6.5e-35

Figure 3.10: Tag finding accuracy on Enron corpus with variable sized Bloom filters

Analysis. With the naive choice of Bloom parameters, there is nearly a 1:1 correspondence between the bits in the Bloom filters and the keywords that generated those bits. The modified Bloom filter parameters weaken this relationship, and as a result, the attack is much less successful.

To make this more precise, let q be the "baseline" probability that any given bit, b, is set in any given Bloom filter. Using the equations from Broder et al. [13], we can compute q from the number of words per document, t, the number of hash functions, h, and the size of the Bloom filter, s:

$$q = 1 - (1 - \frac{1}{s})^{-ht} \approx 1 - e^{\frac{-ht}{s}}.$$

Using the default parameters from Mimesis Aegis [48], we obtain our original q value $q_0 = 6.01\text{x}10^{-5}$.

On the other hand, if some keyword w sets bit $b = 1$, and w occurs in fraction p of the documents, then we should expect to see bit b set in about $p \cdot 1 + (1 - p) \cdot q$ of the encrypted corpus. If the resulting frequency differs significantly from q, then the adversary can easily tell which bits go together. Table 3.2 illustrates this effect. With $q = q_0$, we expect the bits for each keyword to have a unique frequency very close to the frequency, p, of the keyword itself. For example, bits for the most common keyword should appear in about 54% of the encrypted corpus, and bits for the 100^{th} most common keyword should appear in about 9.4% of the documents.

By increasing q, we can make the attack more difficult. The optimized parameters from Broder et al. [13] set q at about 0.5, but it is also possible to drive q even higher while maintaining a low false positive rate. For example, with $s = 2^{11}$ and $h = 25$, we get $q = 0.709$, and the probability of a false positive is less than 10^{-3}. Table 3.2 shows how the bit frequencies change as we increase q. With $q = 0.7$, the frequencies for all bits belonging to the top 300–1000 words will be roughly similar. These similar frequencies makes it increasingly likely that large numbers of bits will be grouped together in the first phase of our attack, and increasingly likely that Algorithm 3.1 will fail to find the unique documents it needs in order to identify the groups of bits for individual keywords.

Word Rank	p	Pr[bit $b = 1$]			
		$q = q_0$	$q = 0.5$	$q = 0.7$	$q = 0.9$
1	0.540	0.540	0.770	0.862	0.954
10	0.311	0.311	0.656	0.793	0.931
50	0.135	0.135	0.568	0.741	0.914
100	0.094	0.094	0.547	0.728	0.909
200	0.062	0.062	0.531	0.719	0.906
300	0.048	0.048	0.524	0.714	0.905
400	0.039	0.039	0.520	0.712	0.904
500	0.034	0.034	0.517	0.710	0.903
600	0.029	0.029	0.515	0.709	0.903
700	0.026	0.026	0.513	0.708	0.903
800	0.023	0.023	0.512	0.707	0.902
900	0.021	0.021	0.511	0.706	0.902
1000	0.019	0.019	0.510	0.706	0.902

Table 3.2: Impact of Bloom filter parameters on bit frequency; Parameters from [48] give $q_0 = 6.01\text{x}10^{-5}$

We caution the reader that these results do not in any way constitute a proof of security, and it is possible that new attacks might still be devised against the improved Bloom filter parameters. Also, the analysis above depends on a few critical simplifying assumptions that do not necessarily hold in practice. First, the equations from Broder et al. [13] assume that the words in the documents are uniformly random; this uniform property is certainly not true for natural language texts. Second, our simplified analysis here assumes that all documents contain the same number of words; this is also untrue for any non-trivial text corpus.

Review. We presented new inference attacks on two recent schemes for efficiently deployable, efficiently searchable encryption. Unlike earlier attacks, ours do not require special knowledge of the documents in the target encrypted corpus. Our analysis of

Bloom filters in Mimesis Aegis illustrates the importance of Bloom filter parameters on the security of the system. We believe our attack would also be effective against other searchable encryption schemes that rely on Bloom filters, such as the SADS anonymous encrypted database [54]. These attacks validate the SADS author's decision to use different hash functions for each document in a later version of the system [51]. Similarly, Goh [29] briefly discusses the possibility of using Mimesis-style tags for efficient searchability. Our results also validate his decision to apply a second layer of protection to his Bloom filters before uploading them to the untrusted server.

Although we have shown that careful tuning of the BF parameters breaks the attacks presented here, we do not have a proof that this defense will be effective against *all* such attacks. The leakage analysis (5.4) of the construction from Chapter 5 shows that this defense is likely not adequate against more sophisticated attacks.

4 Weakly Randomized Encryption

The goal for the Weakly Randomized Encryption construction is maximizing security in operational scenarios where deployability is a strict requirement. As a consequence of this deployability requirement, the security of our schemes will necessarily be somewhat less compared to other approaches where easy real-world deployment is of no concern.

Still, we aim to provide provable security against the most common adversaries, for applications that otherwise could afford no security at all.

We present a new, efficiently searchable, easily deployable database encryption scheme that is provably secure against inference attacks even when used with low-entropy data from the real world.

The security of our schemes is tunable with a single parameter, allowing database owners to choose the most appropriate balance of security versus runtime performance and space overhead for the demands of their individual applications. We also achieve even stronger levels of security by bucketizing a small portion of the data.

Weakly Randomized Encryption. Our core technique is a generalization of a "folklore" encryption technique that we call *weakly randomized encryption* (WRE). WRE is a middle ground between deterministic encryption (DET) and conventional, strongly randomized encryption. With deterministic encryption, each time a plaintext is encrypted, it yields the same ciphertext. DET enables efficient, logarithmic-time search because it allows a legacy server to create an index from only ciphertexts, but it provides very little security for real data [49]. Conventional (strongly) randomized encryption prevents the adversary from learning even a single bit about the plaintext [31], but in doing so, it also precludes the possibility of efficient search.

In weakly randomized encryption, only a few bits of randomness sampled from a low-entropy distribution are used in each encryption. Our analysis shows that this weak randomness is sufficient to protect against inference attacks if we choose the distribution carefully. In order to perform our WRE schemes, one must know the probability distribution of the plaintexts. We believe it is not unreasonable to ask that the data owner must know his data at least as well as the attacker does. The distribution can also be calculated during database initialization.

Deployability. Our constructions are compatible with standard SQL relational databases. They can be deployed immediately on popular cloud service platforms including Google Cloud SQL[1] and Amazon Relational Database Service[2]. They are efficiently scalable up to databases containing millions of records. We performed queries returning up to 10,000,000 records. Our encrypted database, including its server-generated indexes, requires less than twice the space required for the plaintext DB. Query response time with our Poisson Random Frequency construction achieves response times within 27% of those of the plaintext database.

Security. We show that our construction is secure against a passive "snapshot" attacker. We give the adversary access to only the encrypted data and a source of auxiliary information. We assume he does not have access to the encrypted queries, the access patterns or return results. This model includes important real-world threats, including attackers who can obtain offline access to the encrypted database by SQL injection or by stealing a backup hard drive. In contrast, previous easily deployable, efficiently searchable schemes fail to achieve even this modest level of security [49].

We acknowledge that the adversary in this model is weaker than the more powerful adversaries that are typically considered for SSE or oblivious RAM. For new applica-

[1]https://cloud.google.com/sql/
[2]https://aws.amazon.com/rds/

tions that do not require deployability on legacy infrastructure, we recommend that system builders use those stronger but less-deployable constructions.

Outline. In the remainder of this chapter, we introduce our notion of security for Weakly Randomized Encryption (WRE) against inference attacks in Chapter 4.1 and in Chapter 4.2 we present the generic template for a Weakly Randomized Encryption. Then, in Chapter 4.3 we give sequentially stronger variations on this idea, leading up to our most secure construction, WRE with bucketized Poisson salt allocation. We evaluate the performance of our new constructions experimentally with real databases in Chapter 4.4.

4.1 Security Definitions

Our security definitions are closely modeled after the standard notion of security against a *chosen plaintext attack*. Like all previous efficiently searchable constructions, our scheme does not meet the standard definition of Indistinguishable Under Chosen Plaintext Attack (IND-CPA) security, as we must reveal the equality of some plaintexts in order to allow efficient searching. We extend the standard IND-CPA definition as follows:

Where the CPA adversary submits pairs of plaintext messages to its challenger, our adversary submits pairs of *lists* of messages. In the real world, a snapshot adversary does not know the order in which plaintext messages were added to the database. To capture this limitation on the adversary, in our game after the challenger randomly selects one list of messages, they randomly shuffle the selected list to prevent the adversary from using any information about the original order. Finally, the challenger encrypts all messages in the shuffled list and provides the encrypted list back to the adversary. The adversary's task is then to determine which list was selected. This security definition is very similar to the Indistinguishable Multiple Encryptions in Presence of an Eavesdropper [41] definition. The primary difference is the shuffled list.

The two lists of messages are required to contain the same number of messages, and the messages (across both lists) must all be of the same size. Otherwise the adversary could use the size of the ciphertexts to distinguish between the lists of messages.

We call our security game Indistinguishable Shuffled Multiple Encryptions (IND-SME).

Definition 4.1 (The IND-SME Indistinguishability Experiment)**.** *Let* $\Pi = ($Gen, Enc, Dec$)$ *be a WRE searchable encryption scheme with message space* $\mathcal{M}$ *and key space* $\mathcal{K}$*. Let* $\mathcal{X}$ *be the security parameter for* Π*. Let* $\mathcal{A}$ *be an probabilistic polynomial time adversary.*

IND-SME$_{\Pi,\mathcal{A}}(n, \mathcal{X})$*:*

- $(k_0, k_1) \leftarrow \textsc{Gen}(1^n)$.
- *Adversary* $\mathcal{A}$ *is given* $(1^n, \mathcal{X})$ *and chooses a pair of lists of messages* M_0, M_1 *where* $|M_0| = |M_1|$ *and for all* $m_i \in M_0, m_k \in M_1, |m_i| = |m_j|$
- *A uniform bit* $b \in 0, 1$ *is chosen.*
- $\mathcal{A}$ *is given* $edb \leftarrow \textsc{Enc}((k_0, k_1, \mathcal{X}), PRS(M_b))$.
- $\mathcal{A}$ *outputs a bit,* b'
- *If* $b = b'$*, then the output is 1 and* $\mathcal{A}$ *succeeds. A zero is output otherwise.*

Definition 4.2 (IND-SME Indistinguishability)**.** *We say that the encryption scheme* Π *with security parameters* λ *and* n *has IND-SME security if, for all probabilistic polynomial time adversaries* $\mathcal{A}$*,*

$$Pr[\textsf{\textit{IND-SME}}_{\mathcal{A}(n,\mathcal{X})} = 1] \leq \frac{1}{2} + negl(\mathcal{X}, n)$$

where $(\mathcal{X}, n)$ *are the security parameters of our scheme.*

In Chapter 4.3, we introduce our constructions and in Chapter 4.3.4 we use our security definitions to evaluate the single-column security of one of our constructions (which uses a Poisson distribution). Our schemes are tuneable to trade off performance for security, and have acceptable performance and security for sizable databases, as shown in Chapter 5.11.

4.2 Weakly Randomized Encryption

In this section we formalize and extend a "folklore" technique that we call *weakly randomized encryption* (WRE) in text and in Figure 4.1. This technique is the basis for all variants described in Section 4.3.

Weakly Randomized Encryption
Let F be a pseudorandom function with key length n_1. Let $\Pi' = (\textsc{Gen}', \textsc{Enc}', \textsc{Dec}')$ be an IND-CPA secure private key encryption scheme with message space $m \in \{0,1\}^*$ and key length n_0. Let *getSalts* be a function that on input message $m \in \{0,1\}^*$ and message probability distribution function P_m and a security parameter $\mathcal{X}$, outputs S, a list of integers representing the salts and P_S, a probability distribution over the salts. Define a private-key weakly randomized encryption scheme Π as follows:

- $\textsc{Gen}$: On input $1^{n_0}, 1^{n_1}$ run $\textsc{Gen}'(1^{n_0})$ receiving key k_0 and choose uniform $k_1 \in \{0,1\}^{n_1}$. Choose security parameter $\mathcal{X}$.
- $\textsc{Enc}$: On input keys k_0, k_1, security parameter $\mathcal{X}$, and a message m, choose a random salt:
  - $(S, P_S) \leftarrow getSalts(m, P_m, \mathcal{X})$
  - $s \leftarrow sample(S, P_S)$
  - Output the (search tag, ciphertext):

$$(t, c) \leftarrow \big(F_{k_1}(s||m),\ \textsc{Enc}'_{k_0}(m)\big)$$

- $\textsc{Dec}$: On input key k_0, and ciphertext (t, c) output plaintext message:

$$m \leftarrow \textsc{Dec}'_{k_0}(c)$$

- $\textsc{Search}$: On input keys k_0, k_1, parameter $\mathcal{X}$, and a message m:
  - $(S, P_S) \leftarrow getSalts(m, P_m, \mathcal{X})$
  - Output query (*query*) on table (T) containing search tag column (T_t) as shown below:

$$\begin{aligned} query \leftarrow & \big(T_t = F_{k_1}(s_1||m)\big) \vee \\ & \big(T_t = F_{k_1}(s_2||m)\big) \vee ... \vee \\ & \big(T_t = F_{k_1}(s_{|s|}||m)\big) \end{aligned}$$

Figure 4.1: Weakly Randomized Encryption, Decryption and Search

Previous efficiently searchable encryption constructions either have specific requirements on the plaintext data, such as high min-entropy [5], or place limitations on the adversary, such as limiting oracle queries to distinct plaintexts [2].

We can reduce the vulnerability of deterministic encryption due to frequency analysis and other leakage inference attacks by adding a small amount of randomness to the encryption.

We show how a weakly randomized encryption scheme can be constructed as the composition of (i) any efficiently-searchable encryption scheme that satisfies the security definitions from Amanatidis et al. [2] and (ii) a weak randomization, or "salting," function. In this work, we construct our schemes using a variation of the Amanatidis, Boldyreva, and O'Neill [2] ESE, which is itself composed of a randomized encryption scheme (RE) that leaks nothing about the plaintext and a pseudo-random function (PRF) that leaks nothing except equality.

Encryption. The WRE encryption takes as input symmetric keys k_0, k_1; a plaintext m; and the probability distribution P_M of the plaintexts. The encryption algorithm begins by calling the *getSalts* subroutine to pseudorandomly generate a probability distribution P_S over a set S of salts for the message m. The *getSalts* subroutine uses the plaintext distribution P_M to choose a distribution for the salts that makes the frequencies of the ciphertexts (nearly) independent of the plaintext. We give a handful of candidate algorithms for *getSalts*, and evaluate their security, in the following sections. A salt $s \in S$ is chosen at random according to P_S and is pre-pended to the message. The encoding of the pre-pended salts must ensure that no pairs of salts and messages of different lengths results in the same search tag. Finally, the salt and plaintext are input into the PRF to create the search tag and the plaintext is encrypted with the randomized encryption algorithm.

Search. To search the encrypted database for all records with plaintext equal to m, the client first computes all possible search tags $t_1, t_2, \ldots, t_n$ for m and then requests all records having tags equal to t_1 or t_2 ... or t_n. Because the number of unique search tags for each plaintext is small, WRE allows the server to build useful indexes on the encrypted data, just as with DET. To perform the search for each t_i, the server can use built-in indexing techniques to return the list of matching records on columns added by our scheme. Because no custom indexing scheme needs be used, it is deployable on unmodified DBMS services.

Decryption. Given a search tag and a randomized ciphertext, the WRE decryption routine discards the tag and uses the randomized encryption scheme's decryption function on the ciphertext to obtain the plaintext.

Updates. One advantage of WRE versus stronger searchable encryption schemes such as SSE, is that updates are simple with WRE. To insert a new record in the encrypted database, we use the encryption function to obtain its weakly randomized search tag and its (strongly randomized) ciphertext. Then we simply append the tag and ciphertext to the database. If we assume new records inserted are drawn from the same plaintext distribution, then adding new records will not affect the WRE tag frequencies. Thus it is secure under the snapshot adversary model. The challenge with SSE updates comes from a different security model that allows the adversary to query the database while providing forward security. Because of the security model and the encrypted indexes used by SSE, SSE typically performs updates in batches using new keys, resulting in multiple indexes.

Future work will address security when the distribution changes from updates or if the adversary has specific knowledge of the updated records.

The improvement in security, if any, of WRE over deterministic encryption is not immediately clear. Surprisingly, our analysis also shows that, with a carefully chosen *getSalts* algorithm, we can construct a weakly randomized encryption that leaks virtually no information about the plaintext to a snapshot adversary who knows the distribution P_M.

4.3 WRE Schemes

In this section, we present our variants that each complete the WRE construction described in Chapter 4.2. We first present simpler, weaker constructions to give the reader an understanding of our motivations for our later, more secure schemes in Chapters 4.3.4 and 4.3.5. We do not fully analyze the security of these weaker schemes because we believe they are inferior to later schemes.

4.3.1 Fixed Salts Method

We refer to the "folklore" version of weakly randomized encryption as the "fixed salts" method, because it always uses a constant number of salts for every plaintext, regardless of the frequency of the plaintext. We label the security parameter of this scheme as N, the number of unique salts per plaintext.

Notion of Security. If a plaintext m occurs in the unencrypted database with frequency p, then with fixed salts, each of m's N ciphertexts will occur in the EDB with frequency $\frac{p}{N}$. Intuitively, the fixed salt method improves on the security of deterministic encryption because it reduces the differences in the plaintext frequencies.

Limitations. First, the overall improvement to security is small. For large databases, the adversary can still guess the plaintext with very high accuracy. Second, the fixed salt WRE is not very efficient. In order to achieve any security for a database of moderate size, it needs a large number of salts, making query processing unnecessarily intensive, especially for low-frequency plaintexts. We could potentially improve both of these aspects if we modified the chance of picking each salt with the frequency of its respective plaintext. We formalize this idea in the next section.

4.3.2 Proportional Salts Method

The fixed salts method can be improved by taking into account the frequencies of the plaintexts in the database. Intuitively, we would like each search tag to occur with roughly the same frequency, regardless of the plaintext. In the *proportional salts* method, we allocate a different number of salts to each plaintext, in proportion to its frequency in the plaintext data. Let the security parameter be the total number of unique ciphertexts be N_T. Then for a plaintext m with frequency $P_M(m)$, we use $N_m \approx P_M(m) \cdot N_T$ salts. Therefore, for any two plaintexts $m_0, m_1 \in \mathcal{M}$, their search tags will appear in the EDB with approximately the same frequency.

Limitations. Unlike the fixed salts method, proportional salt allocation requires that the data owner must know the plaintext distribution P_M in order to encrypt a message.

Another limitation of proportional salts stems from the fact that we must allocate an integer number of salts for each plaintext. This limitation gives rise to an aliasing problem, where in certain situations using more salts can actually reduce the security. For example, consider an example database column with $P_M(m_1) = 0.7$ and $P_M(m_2) = 0.3$. For $N_T = 10$, this works out nicely, but if we encrypt this database with $N_T = 12$, then we will round our number of search tags to 8 for plaintext m_1, each with frequency 0.0875, and 4 for plaintext m_2, each with frequency 0.075. Given sufficiently many encrypted records, the adversary will be able to distinguish the plaintexts using this frequency disparity.

4.3.3 Remainder Salts

Scaling the number of salts with the frequency of the plaintext is effective, but these frequencies will still have discrepancies, allowing for analysis. With only proportional salt scaling, we must choose a discrete number of salts, which means we cannot create groups of unique ciphertexts with arbitrary frequencies from each plaintext. The granularity is inversely proportional to the max salt count. With a high enough salt count, we have more granularity for ciphertext frequencies, allowing their frequencies to be very close together, hindering analysis. But requiring more salts means requiring more complicated queries, slowing down query time and causing other efficiency problems.

With the Remainder Salts method, we try encrypt as much of the database as possible with ciphertexts that have statistically identical frequencies.

This extension biases our random number generator while choosing a salt, to choose one salt less often than the rest, allowing these other salts to be chosen with any frequency.

Algorithm 4.1 Remainder Salt Method

```
function GetSalts-Remainder($P_M$, $m$, $k$)
    $I \leftarrow P_M(m) \cdot N_T$
    $N_r \leftarrow \lfloor I \rfloor$
    $N_m \leftarrow N_r + 1$
    $S \leftarrow [1, N_m]$
    for each $s$ in $S$ do
        if $s < N_m$ then
            $P_S(s) = 1/N_T$
        else
            $P_S(s) = (1/N_T) * (I - N_r)$
    return $S, P_S$
```

Notion of Security. This technique allows us to set these “non-remainder salt” ciphertext frequencies to exactly the same value, removing all frequency analysis. Ciphertexts that are encrypted with a remainder salt are much easier to distinguish, but this ciphertext is only a single ciphertext for each plaintext, so it should be a small portion of the database. Because we can set the average frequency of non-remainder salt ciphertexts to any value, we set them all to the same value. This shared frequency will be equal to $1/N_T$, where N_T is the total number of non-remainder salts. Thus we are intentionally leaking these remainder frequencies in exchange for improved security for the rest of the data with identical frequencies and the efficiencies afforded by using fewer salts.

4.3.4 Poisson Random Frequencies

A Poisson process is a simple stochastic process often used to model the arrival of events in a system, for example the occurrence of earthquakes in a geographical region, or the arrival of buses at a bus stop. In a Poisson process with rate parameter λ, the times between arrival events, called the "interarrival times," are independent and identically distributed, and they follow an Exponential distribution with parameter λ. The number of arrivals in an interval of length t is independent of the events in all intervals before and after, and it is Poisson distributed with expected value λt.

In the Poisson variant of WRE, the security parameter is the Poisson rate parameter λ. On expectation, this method will generate about $\lambda + |\mathcal{M}|$ search tags in total across all plaintexts. To allocate salts for plaintext $m \in \mathcal{M}$ and to assign their relative weights, we sample arrivals in the interval $[0, P_M(m)]$ from a Poisson process with rate λ. Let the number of arrival events in the interval be N, and let their times be denoted $a_1, \ldots, a_N$. Additionally, we define $a_0 = 0$ and $a_{N+1} = P_M(m)$. The interarrival times are $x_i = a_i - a_{i-1}$ for $i \in 1, \ldots, N+1$.

Based on the outcome of this experiment, we allocate $N+1$ salts to plaintext m, and when we encrypt m, we choose salt i with probability $\frac{x_i}{P_M(m)}$. The resulting search tag will then have frequency equal to x_i in the encrypted database. Also note that N has a Poisson distribution, thus on average we will allocate about $\lambda \cdot P_M(m) + 1$

salts to plaintext m.

The pseudocode for our Poisson method's algorithm is shown below in Algorithm 4.2.

Algorithm 4.2 Poisson Salt Distributions

```
 1: function GetSalts-Poisson($P_M$, $m$, $k$, $\lambda$)
 2:     $s = 0$
 3:     $E = Exponential(\lambda)$
 4:     $total = 0$
 5:     while $total < P_M(m)$ do
 6:         $s = s + 1$
 7:         $weight[s] \leftarrow Sample(E)$
 8:         $total = total + weight[s]$
 9:     $weight[s] \leftarrow P_M(m) - (total - weight[s])$
10:     $S = [1, s]$
11:     for $s \in S$ do
12:         $P_S(s) \leftarrow \frac{weight[s]}{P_M(m)}$
13:     return $S, P_S$
```

Security. Our analysis shows how the unique properties of the Poisson process make it ideally suited for use in weakly randomized encryption. Most critically, the Poisson process guarantees that, subject to one constraint on λ, all search tag frequencies for all plaintexts are pseudorandom samples from indistinguishable Exponential distributions. Therefore a computationally bounded adversary learns nothing about the plaintext from the frequencies of the ciphertexts.

Proof sketch. In the Poisson approach, the frequency of the first salt for each plaintext is not drawn from the same Exponential distribution as the others. To see why this is so, notice that the Poisson process may generate zero arrival events in the interval $[0, P_M(m)]$. Zero arrival events happens whenever the first arrival time from the Poisson process occurs after the end of the interval; in other words, when the first interarrival time is greater than $P_M(m)$.

Then we have only a single salt, and hence a single ciphertext that appears in the encrypted database with the same probability as the plaintext, $P_M(m)$. Therefore the distribution of the first search tag frequency is not in fact an Exponential; all the probability mass that the Exponential would assign to values greater than $P_M(m)$ is instead lumped onto the point $P_M(m)$. We call this distribution a "capped Exponential" with parameters λ and $\tau = P_M(m)$. Figure 4.2 illustrates the difference between the capped and regular Exponential distributions.

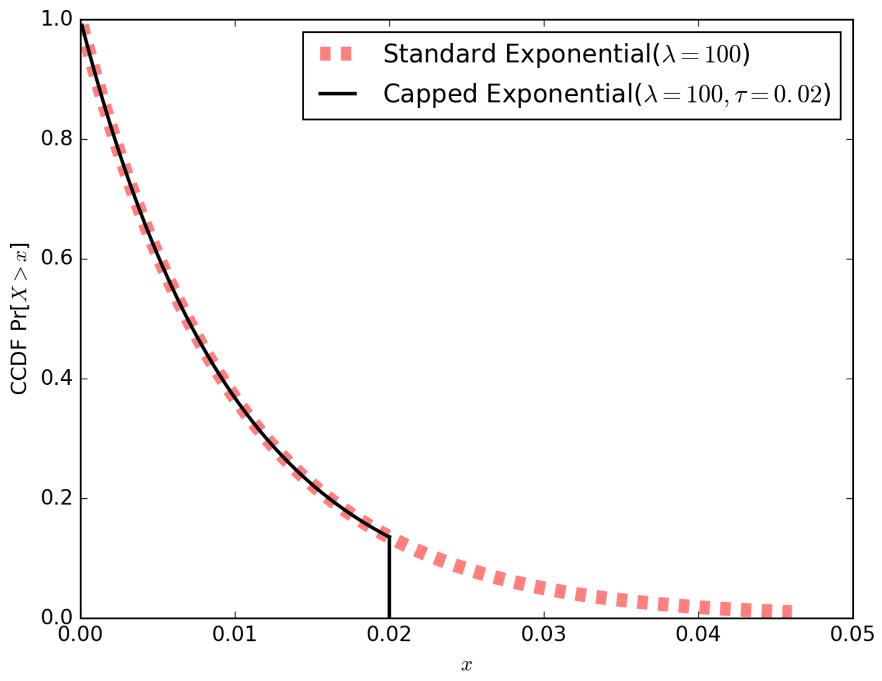


Figure 4.2: Complementary cumulative distribution for capped versus standard exponentials

The adversary could attempt to exploit this difference to their advantage in the IND-SME game. In the extreme case, the adversary would choose M_0 with all unique plaintexts, and choose M_1 where all $m \in M_1$ are the same plaintext. With a low value of λ, the messages from M_0 in this example would all be drawn from the capped exponential while all of the messages from M_1 would not. However, with a high enough λ in relation to the number of messages in M_0, all of the messages from M_0 would be drawn from the non-capped exponential with very high probability. The important point is to choose an appropriate λ parameter based on the distribution being encrypted.

The statistical distance between the standard Exponential(λ) and the capped Exponential with λ and τ is defined in Definition 2.5 as one half of the total variation distance between the two distributions. Notice that the two distributions are identical to the left of τ. Therefore all of the difference in distribution comes from the upper tail of the standard Exponential, where the capped Exponential assigns zero probability. From the definition of the Exponential distribution, this quantity is

$$\begin{aligned}&\Delta(Exp(\lambda), CappedExp(\lambda, \tau))\\ =&Pr(X > \tau | X \sim Exp(\lambda))\\ =&e^{-\lambda\tau}\end{aligned}$$

Thus the probability of the adversary distinquishing between the capped exponential and exponential distributions is negligible in λ, which is the goal of our security definition.

If we let $\tau = \max_m P_M(m)$ be the smallest plaintext frequency, then by increasing λ relative to τ, we can make this probability arbitrarily small. Furthermore, we can calculate the Poisson rate parameter λ that is required to achieve a desired security parameter ω, where $\omega \leq Pr(X > \tau)$):

$$\lambda \geq \frac{\log \omega}{\tau}$$

Limitations. When the adversary has the frequencies of all search tags and knows P_M, Lacharite and Paterson [52] pointed out another possible attack, wherein the adversary finds a set of search tags whose counts sum up to the expected count for a (set of) target plaintext(s). The adversary might then reasonably conclude that those search tags all represent encryptions of the given plaintext(s).

When the adversary targets a single plaintext, it must solve an instance of the subset sum problem (SSP). When targeting multiple plaintexts simultaneously, the adversary must solve an instance of the multiple knapsack problem (MKP). While both problems are NP, there exist efficient approximation algorithms, for example Chekuri et al. [19].

Even if the adversary can find a solution to the computational problem, there is no guarantee that the matching it finds will be correct. To see that this is true, consider the case where each search tag occurs exactly once. Then all possible plaintext-to-search-tag matchings give equally valid solutions to the problem, and the adversary can do no better than random guessing. We leave a more detailed exploration of the efficacy of such attacks for future work. Instead, in the following section, we present an improved WRE construction using bucketization to eliminate the attack entirely.

4.3.5 Bucketized Poisson Random Frequencies

The Poisson WRE approach above generated randomized search tags for each plaintext. Doing so makes any one search tag equally likely under all possible plaintexts, so the adversary learns nothing by examining a single tag. Unfortunately, that does not guarantee security against an adversary who considers the combined frequencies of several tags at once. In this section, we show how a simple extension of the Poisson WRE approach, using bucketization, can protect against the matching attacks described in the previous section.

Figure 4.3 illustrates the difference in the two schemes. In the (non-bucketized) Poisson WRE, we sample a set of points from a Poisson process for each plaintext m, over the interval $[0, P_M(m)]$. We then use the inter-arrivals between the points to determine the frequencies of the search tags for m. This technique is equivalent to starting with the set of points $\{F_M(m) : m \in \mathcal{M}\}$ and then sampling more points from the Poisson process over the interval $[0, 1]$.

In the Bucketized Poisson approach, we omit the points from $F_M(m)$, and we simply sample from the Poisson process, independent of the plaintext frequencies. As a result, some inter-arrival intervals will overlap with the intervals for more than one plaintext. Notice that in Figure 4.3 with the bucketized construction, the tag t_3 can represent either plaintext m_1 or m_2.

The Bucketized Poisson also makes a slight modification to the encryption and search algorithms from Figure 4.1. Instead of inputting the plaintext appended to a salt to the PRF, just the salt is given.

- Enc: on input keys k_0, k_1, security parameter $\mathcal{X}$, plaintext distribution P_M and a message m, choose a uniform salt
  $(S, S_M) \leftarrow getSalts(P_M, \mathcal{X})$
  $s \leftarrow sample(S(m), S_M(m))$
  Output the (search tag, ciphertext):

$$(t, c) \leftarrow \big(F_{k_1}(s),\ Enc'_{k_0}(m)\big)$$

- Search: on input keys k_0, k_1, parameter $\mathcal{X}$, and a message m, let $(S, S_M) \leftarrow getSalts(P_M, \mathcal{X})\}$ $s = S(m)$. Output search query:

$$q \leftarrow T_t = F_{k_1}(s_1) \vee T_t = F_{k_1}(s_2) \vee ... \vee T_t = F_{k_1}(s_{|s|})$$

This additional ambiguity completely removes the small advantage that an adversary might obtain from the imperfection of the capped exponential distribution. It also

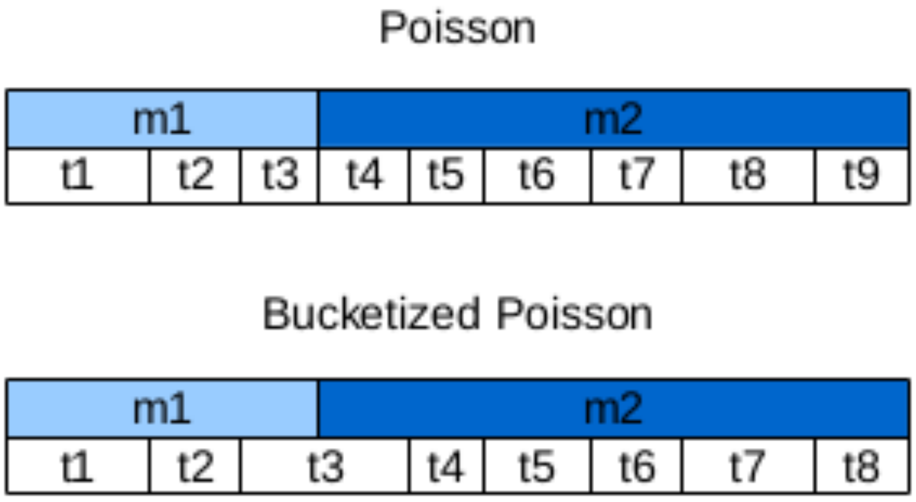


Figure 4.3: Poisson Search Tag Frequency Example

negates any kind of frequency-based matching attack, because the tag frequencies and the plaintext frequencies are independent. The downside is that with the bucketized WRE, query results will contain a small number of false positives. The false positive rate is controlled by parameter λ: increasing λ (thus decreasing the expected frequency of each tag) decreases the expected number of false positives.

Theorem 4.1 (Single-Column Security for Bucketized Poisson WRE). *A Bucketized Poisson WRE scheme with parameters* (λ, n_0, n_1)*, is IND-SME secure.*

Proof sketch. Assuming the adversary cannot attack the PRF or Π', then this construction only leaks frequency and ordering. With the Bucketized Poisson algorithm, the actual ciphertext search tags will have exactly the same values and the same frequency, no matter which M_0 or M_1 is encrypted. The only difference will be the ordering of these search tags. Since the ordering is determined by the output of a pseudo-random shuffle, the adversary cannot learn anything from this ordering either. The security of this construction also does not depend on λ like the first Poisson approach as λ only affects the false positive rate and performance.

Algorithm 4.3 Bucketized Poisson

```
1: function GetSalts-Poisson($P_M$,$M$,$m$, $k$, $\lambda$)
2:     $s = 0$
3:     $wordFr = [], salts = []$
4:     $E = Exponential(\lambda)$
5:     $total = 0$
6:     while $total < 1.0$ do
7:         $s = s + 1$
8:         $weight[s] \xleftarrow{\$} Sample(E)$
9:         $total = total + weight[s]$
10:    $weight[s] \leftarrow 1.0 - (total - weight[s])$
11:    $M' \leftarrow PRS(M)$
12:    $fr = P_M(m'_1) + ...P_M(m'_{x-1})$ where $m = m'_x$
13:    $i = 0, cdf = 0$
14:    while $cdf < fr$ do
15:        $cdf = cdf + weights[i]$
16:        $i = i + 1$
17:    $weights[i] = cdf - fr$
18:    $cdf = fr$
19:    while $cdf < (fr + P_M(m))$ do
20:        $wordFr.append\left(\frac{weights[i]}{fr}\right)$
21:        $salts.append(i)$
22:        $i = i + 1$
23:        $cdf \leftarrow cdf + weights[i]$
24:    if $cdf > (fr + P_M(m))$ then
25:        $dif \leftarrow (fr + P_M(m)) - cdf$
26:        $wordFr.append\left(\frac{weights[i] - dif}{fr}\right)$
27:        $salts.append(i)$
       return $(salts, wordFr)$
```

4.4 Performance Evaluation

We implemented several flavors of weakly randomized encryption, including the fixed salts method and Poisson salt allocation, in the Haskell programming language. To evaluate the performance of our prototype on realistic data and queries at a variety of scales, we used the SPARTA [61] framework from MIT-LL.

The SPARTA test framework includes a data generator and a query generator. The data generator builds artificial data sets with realistic statistics based on real data from the US Census and Project Gutenberg. Table 4.1 illustrates the table schema used in our tests. The query generator creates queries for this test database based on the desired query types and number of return results.

4.4.1 Experimental Setup

We used the database generator to generate databases with 100,000 records, 1 million records and 10 million records. We generated over 1,000 queries for each database, consisting of a mix of queries that returned result sizes between 1 and 10,000 records.

We encrypted the columns `fname`, `lname`, `ssn`, `city`, and `zip` with WRE. The rest of the SPARTA columns were inserted into the test database in plaintext.

Each encrypted column is expanded into two columns: one 64 bit Integer column for the WRE search tag and another column to hold the (strongly randomized) AES-encrypted data. The plaintext table contains 23 columns. Therefore the ciphertext table contains the 23 encrypted data columns, plus the 5 additional search tag columns. Each search tag column is indexed.

We tested the performance of the fixed salt method with 100 and 1,000 salts, and we tested Poisson salt allocation using λ of 100, 1,000, and 10,000.

Column Name	Column Type
id	BIGINT
fname	CHAR 11
lname	CHAR 15
ssn	CHAR 9
dob	DATE
age	INT
address	CHAR 100
city	CHAR 35
state	ENUM
zip	CHAR 5
sex	ENUM
race	ENUM
marital_status	ENUM
school_enrolled	ENUM
citizenship	ENUM
income	INT
military_service	ENUM
language	ENUM
hours_worked_per_week	SMALLINT
weeks_worked_last_year	TINYINT
last_updated	INT
foo	BIGINT
xml	VARCHAR 10,000

Table 4.1: Table Schema

We performed the tests with the client and the database server located on the same local network via a 1 Gbps Ethernet switch. The server has 12 CPU cores (dual Xeon E5645), 64GB of RAM, and an array of 10k RPM hard drives. It runs the Ubuntu Server 14.04 operating system and Postgres 9.6 as the DBMS.

4.4.2 Experimental Results

Ciphertext Expansion. Table 4.2 shows the overall ciphertext expansion, including the ciphertext expansion from the AES encrypted data, the additional search tag columns and the additional indexes on the search columns. Note that the number of salts used and whether a fixed salt or a Poisson Salt Distribution do not affect the database size. The database ciphertext expansion is directly related to the number and type of columns encrypted.

Encryption Type	DB Size	DB + Indexes Size
100k Plaintext	112 MB	136 MB
100k Encrypted	156 MB	244 MB
1M Plaintext	1116 MB	1365 MB
1M Encrypted	1558 MB	2447 MB
10M Plaintext	11 GB	13 GB
10M Encrypted	15 GB	24 GB

Table 4.2: Ciphertext Expansion

Database Creation. Inserting 10 million plaintext records took a total 6,356 seconds on average. Inserting 10 million ciphertext records took 58,604 seconds on average. Because the database must only be initialized once, the practical impact of this 9x slowdown is not especially significant for most applications. Also, we believe that with a little effort, the performance of our un-optimized implementation could be improved substantially.

Query Runtime. We performed two variations of each SPARTA-generated query. The first variation takes the form *SELECT ID from main where column = value*. Since column `ID` is the primary key, these queries only require that the DBMS scan the indexes to find the list of matching records. The second variation takes the form *SELECT * from main where column = value*. This selects the entire record, and thus requires retrieving the encrypted records from storage and transferring them across the network. The time shown for each query includes the time to compute the encrypted query.

Since caching can have a big impact on database performance, we ran each set of queries under two scenarios. In the first scenario we cleared the caches in the OS and in the Postgres database before running each query. To clear the Postgres cache, we restarted Postgres. To clear the OS cache, we ran the following:

```
echo 3 > /proc/sys/vm/drop_caches
```

In the other scenario, the cache was left alone.

Figures 4.4 and 4.5 display tests run with a cold cache. Figures 4.6 and 4.7 have the results of the warm cache tests.

The results of these experiments show that the WRE schemes achieve query response times with our Poisson Random Frequency construction within 27% of plaintext database response times on equality queries. As expected, as the number of unique search tags increases, so does the query response time. Across all of our experimental configurations, the Fixed Salt scheme with 1000 salts is slower than the Poisson construction with $\lambda = 1000$, and similarly, the Poisson with $\lambda = 1000$ performs slightly slower than Poisson with $\lambda = 100$. This result is not surprising, since the Fixed Salt technique generates 1000 tags for each plaintext, while the $\lambda = 1000$ results in $\lambda + |M|$ tags for the entire column.

Reset Cache Before Each Query

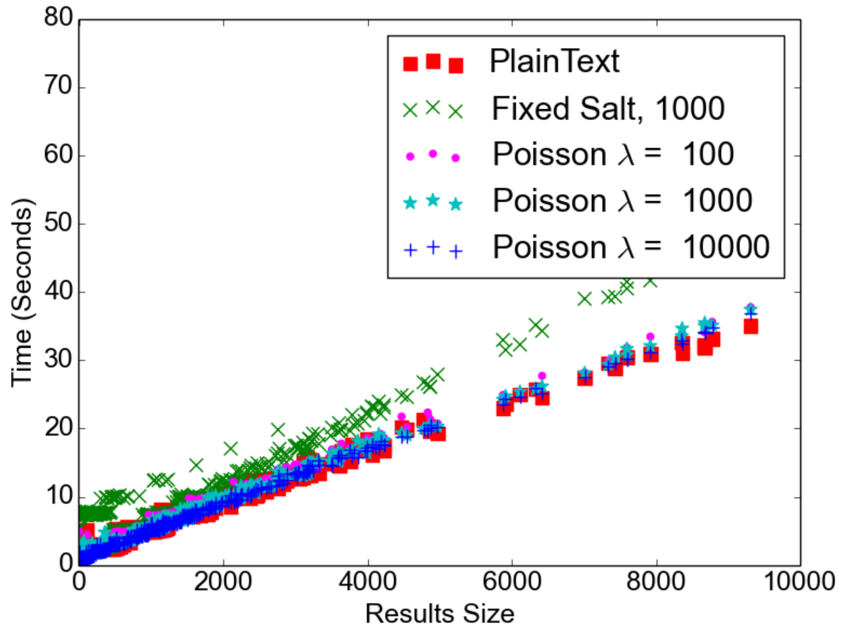


Figure 4.4: “SELECT ID” Equality Query Runtime

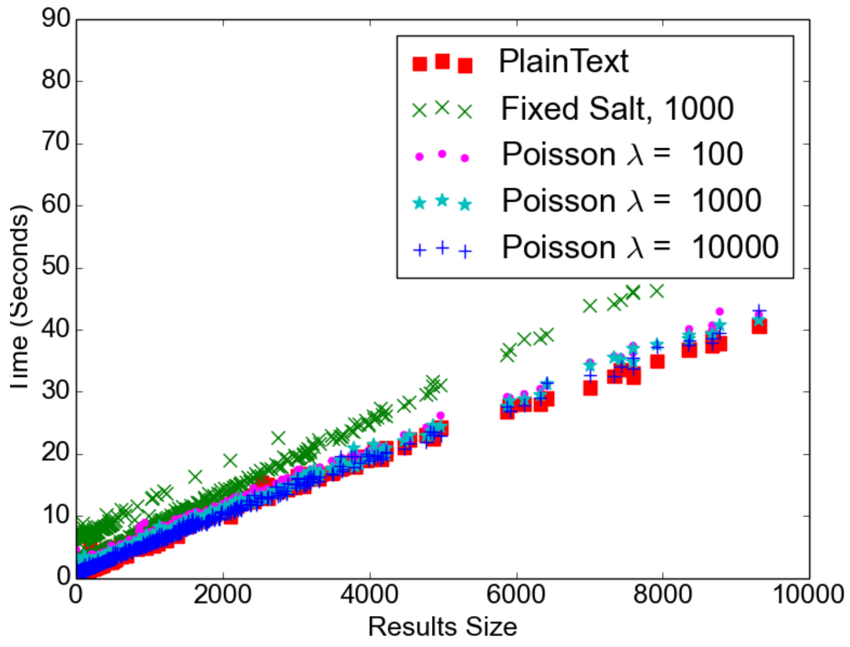


Figure 4.5: “SELECT *” Equality Query Runtime

Warm Cache No Cache Reset

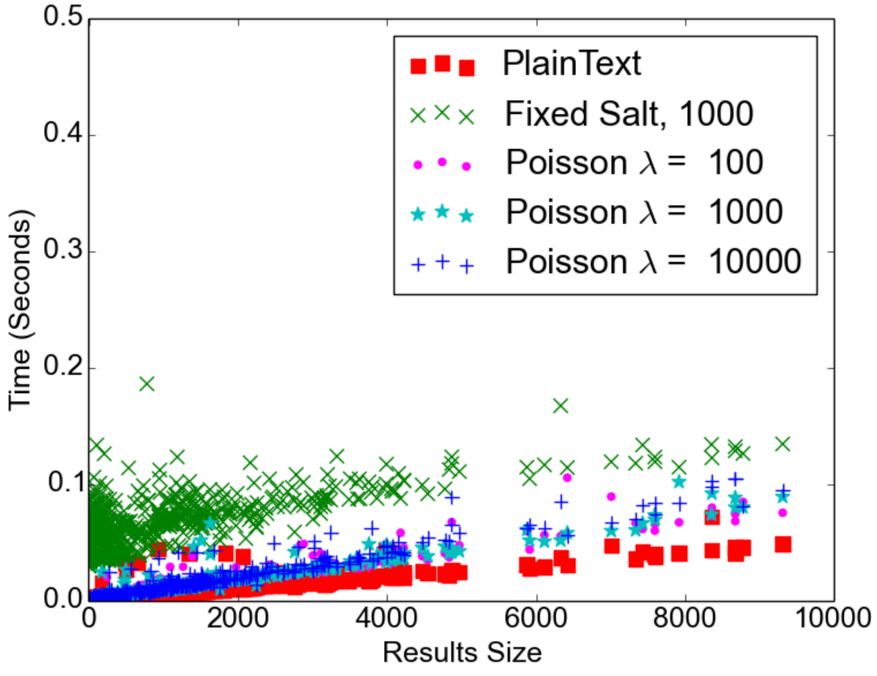


Figure 4.6: “SELECT ID” Equality Query Runtime

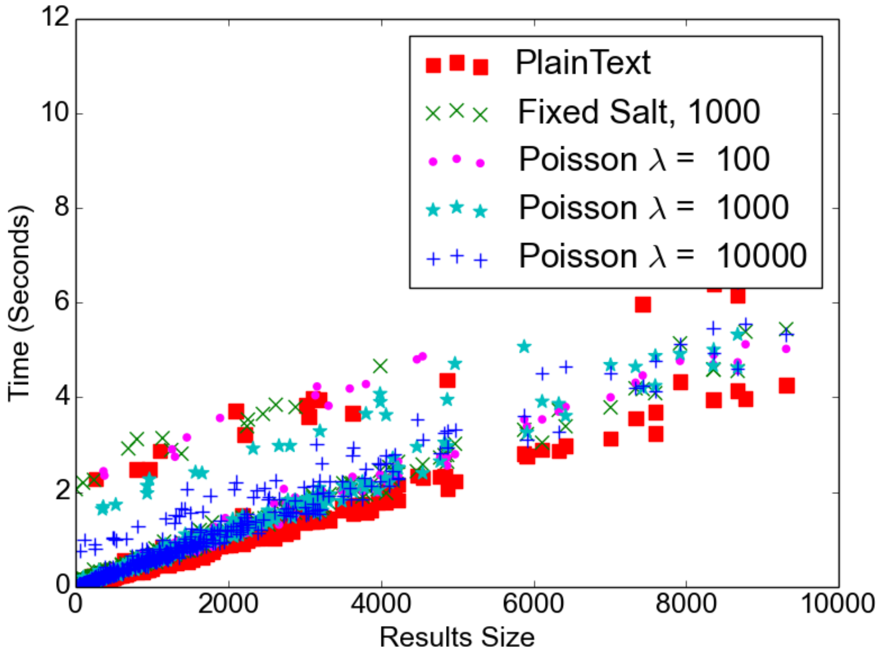


Figure 4.7: “SELECT *” Equality Query Runtime

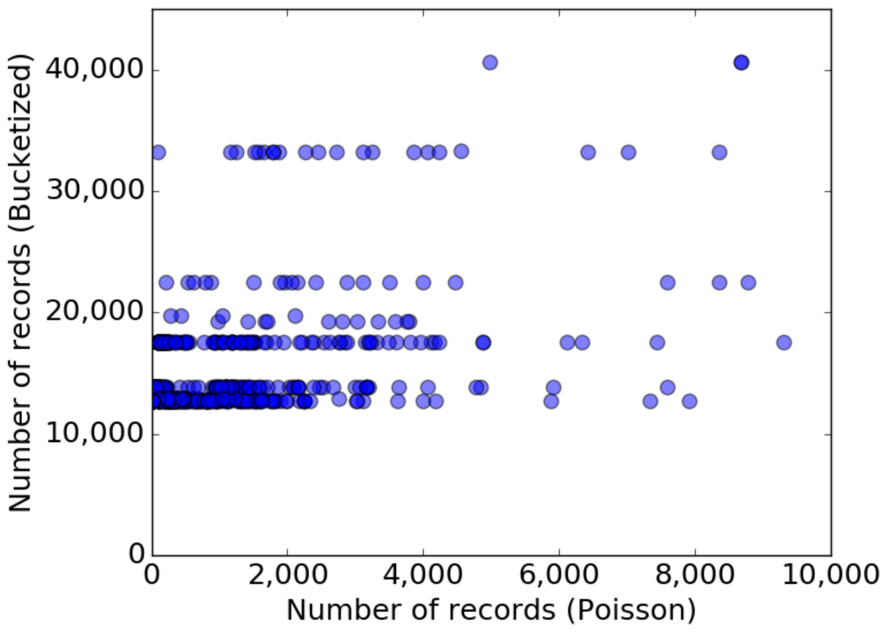


Figure 4.8: Bucketized Poisson False Positive ($\lambda = 1000$)

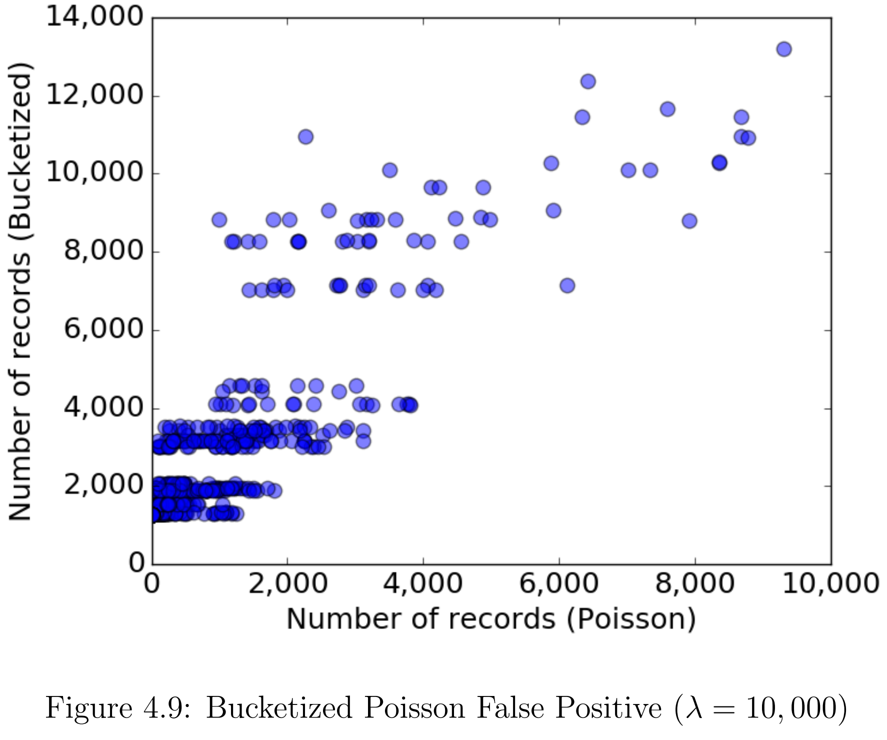


Figure 4.9: Bucketized Poisson False Positive ($\lambda = 10,000$)

Bucketized Poisson False Positives. In Chapter 4.3.5, we mentioned that the Bucketized Poisson algorithm may result in false positives in the search result. Figures 4.8 and 4.9 show the false positives introduced on the SPARTA queries used in our performance evaluation. The X axis shows the number of records returned for each query with Poisson salt allocation, which does not introduce false positives. The Y axis shows the number of records returned for the same queries with the bucketized version of the algorithm.

With lower values of λ, the Bucketized Poisson algorithm appears to mask the true number of return results. In Figure 4.9, with $\lambda = 10,000$, we see some correlation between the number of matching records in the database and the number of ciphertext records that match the bucketized query. However, in Figure 4.8 with $\lambda = 1000$, the relationship is much weaker. In the future, this masking might be leveraged to prevent reconstruction attacks [42, 47], where an adversary uses access pattern leakage to recover the contents of the database.

5 Easily Deployable Database Encrypted Search (EDDiES)

This chapter describes an easily deployable searchable encryption construction that provides security against inference attacks, even using the SSE security model where the adversary can issue queries.

Bloom Filter Index. Our technique uses Bloom filters as indexes, with one Bloom filter per database record. Our variation of inserting words into the Bloom filter indexes ensures that the bit address frequencies set to one in these indexes are uniform. Postgres does not come with an efficient indexing solution for a Bloom filter, so we built a custom Postgres index, allowing faster sub-linear search of these indexes.

Deployability. This construction is compatible with standard SQL relational databases and can be deployed immediately on popular cloud service platforms including Google Cloud SQL[3] and Amazon Relational Database Service[4]. It is efficiently scalable up to databases containing millions of records.

Performance. We tested our construction on databases of 1,000,000 records with queries returning up to 10,000 records. Performances varied between 5 and 40 time slowdown, depending on the security parameters used. However this slowdown appears as a constant factor. The performance mimics plaintext performance based on the size of the return results, with a constant factor added to traverse the custom index.

Security. We show that our construction meets the SSE non-adaptive security definition. Informally, this definition means we define the leakage of our system and show that we do not leak anything else. In addition, we take the necessary extra step of analyzing the leakage to show what an adversary can learn from it, depending on

[3]https://cloud.google.com/sql/
[4]https://aws.amazon.com/rds/

the security parameters used.

5.1 Background

5.1.1 Probability Background

Probability of the union of two events.

$$Pr[A \cup B] = Pr[A] + Pr[B] - Pr[A \cap B] \tag{5.1}$$

Probability of the union of three or more events.

$$Pr[A \cup B \cup ... \cup Z] = 1.0 - Pr[\neg A \cap \neg B \cap ... \cap \neg Z] \tag{5.2}$$

Independent Events. Two events are independent if the probability that one event occurs has no effect on the probability of the other. If A, B are independent, then

$$Pr[A \cap B] = Pr[A] \cdot Pr[B] \tag{5.3}$$

If Equation 5.3 does not hold, then the events are dependent. If If A, B are dependent then

$$Pr[A \cap B] = Pr[B|A] \cdot Pr[A] \tag{5.4}$$

Binomial Probability. If a binomial experiment consists of n trials and results in x successes with the success of an individual trial is P. Then the binomial probability $b(x; n, P)$ is:

$$b(x; n, P) = \frac{n!}{x! \cdot (n-x)!} \cdot P^x \cdot (1-P)^{(n-x)}$$

5.1.2 Bloom Filter Background

Bloom filters [8] are probabilistic data structures that represent sets and support membership queries. For applications that can tolerate a small false positive rate, Bloom filters offer a space-efficient alternative to a full inverted index. Conceptually, the Bloom filter is an array or bit vector of m bits, all initially set to zero.

The typical use of Bloom filters in searchable encryption uses one Bloom filter per document or database record. Each keyword in the document or row is inserted into the Bloom filter. Searching for a keyword involves simply checking each Bloom filter, or searching a structure such as a tree of Bloom filters.

Recall from chapter 2.1.2, formula 2.1 provides the probability of any bit being set to one in a Bloom filter:

$$PR[bit_i = 1] = 1 - (1 - \frac{1}{s})^{ht} \approx 1 - e^{\frac{-ht}{s}}$$

Where

s is the size of the Bloom filter

t is the number of words inserted into the Bloom filter

h is the number of hash functions

And formula 2.2 states the false positive rate:

$$p^h \text{ where } p = PR[bit_i = 1]$$

However, these formulas assume the input to the Bloom filter is uniform. When the input is not uniform, some bits will have a higher probability of being set to one than others. In most database applications in the real world, the data is not uniform. The probability of each bit set to one varies with non-uniform data. Using a Bloom filter with a classical construction will not hide the frequencies of non-uniform data.

Hiding the frequencies is the key to our construction. We modify the classic Bloom filter insertion technique to ensure that the bit frequencies are uniform, even when non-uniform data is inserted.

5.2 Security Definitions

We define the security of our construction using the SSE framework from Curtmola et al. [21]. Their framework requires a formal definition of the leakage. It then provides an ideal-v-real game used to prove that a SSE adversary does not learn anything more than the leakage. We use their framework with two modifications. Our encryption algorithm has supplemental security parameter inputs. In addition to an encryption key, we have Bloom filter parameters as input. Our leakage function also differs by necessity, since our construction has a distinctive profile compared to a typical SSE scheme.

5.2.1 SSE Definitions

SSE definitions focus on document collections. In a structured database application, a table row is very similar to a document and we will use a table row in the same setting as a document and treat them equivalently.

A table in a relational database contains one or more data categories in columns. Each row, also called a record or tuple, contains a unique instance of data, for the categories defined by the columns. Each table has a unique primary key, which identifies the information in a table. We will call this primary key the record ID, which is analogous to a document ID.

Definition 5.1 (History). *Let Δ be a dictionary and $D \subseteq 2^{\Delta}$ be a document collection over Δ. A* q-query history *over D is a tuple $H = (D, w)$ that includes the document collection D and a vector of q keywords $w = (w_1, ..., w_q)$.*

Definition 5.2 (Access Pattern). *Let Δ be a dictionary and $D \subseteq 2^{\Delta}$ be a document collection over Δ. The* access pattern *induced by a q-query history $H = (D, w)$, is the tuple $\alpha(H) = (D(w_1), ..., D(w_q))$ where $D(w_i)$ is the set of documents accesses from query of keyword i.*

Definition 5.3 (Search Pattern). *Let Δ be a dictionary and $D \subseteq 2^{\Delta}$ be a document collection over Δ. The* search pattern *induced by a q-query history $H = (D, w)$, is a symmetric binary matrix $\sigma(H)$ such that for $1 \leq i, j \leq q$, the element in the i^{th} row and j^{th} column is 1 if $w_i = w_j$, and 0 otherwise.*

The original leakage definition from [21] (trace) is below:

Definition 5.4 (Trace). *Let Δ be a dictionary and $D \subseteq 2^{\Delta}$ be a document collection over Δ. The* trace *induced by a q-query history $H = (D, w)$, is a sequence $\tau(H) = (|D_1|, ..., |D_n|, \alpha(H), \sigma(H))$, comprised of the lengths of the documents in D, and the access and search patterns induced by H.*

Since our construction has additional leakage, we replaced Definition 5.4 with Definition 5.5, which we call leakage. Our additional leakage comes from our unencrypted indexes. For now, we simply label this leakage $\mathcal{L}(I)$. In section 5.4, we will go into detail about this leakage.

Definition 5.5 (Leakage)**.** *Let Δ be a dictionary and $D \subseteq 2^{\Delta}$ be a document collection over Δ. The* leakage *induced by a q-query history $H = (D, w)$, is a sequence $\tau(H) = (\mathcal{L}(I), \alpha(H), \sigma(H))$, comprised of the index leakage $\mathcal{L}$ and the access and search patterns induced by H.*

Definition 5.6 (Non-adaptive semantic security)**.** *Let $SSE = (\textsc{Gen}, \textsc{Enc}, \textsc{Trpdr}, \textsc{Search}, \textsc{Dec})$ be an index-based SSE scheme, pm be the security parameters, A be an adversary, S be a simulator and consider the following probabilistic experiments:*

```
$\boldsymbol{Real}_{SSE,A}(pm)$
$K \leftarrow Gen(1^k)$
$(st_A, H) \leftarrow A(pm)$
$parse\ H\ as\ (D, w)$
$(I, c) \leftarrow \textsc{Enc}_K(D)$
$for\ 1 \leq i \leq q,$
    $t_i \leftarrow \textsc{Trpdr}_K(w_i)$
$let\ t = (t_1, ..., t_q)$
$output\ v = (I, c, t)\ and\ st_A.$
```

```
$\boldsymbol{Sim}_{SSE,A,S}(pm)$
$v \leftarrow (S(\tau(H)))$
$output\ v = (I, c, t)\ and\ st_A.$
```

We say that SSE is semantically secure *if for all polynomial-size adversaries A, there exists a polynomial-size simulator S such that for all polynomial-size distinguishers D:*

$$\Big| Pr[D(v, st_A) = 1 : (v, st_A) \leftarrow Real_{SSE,A}(pm)] - \\ Pr[D(v, st_A) = 1 : (v, st_A) \leftarrow Sim_{SSE,A,S}(pm)] \Big| \leq negl(pm)$$

To prove security, the output of a simulator that receives only the leakage of the construction must be indistinguishable from the output of the construction.

5.3 Searchable Encryption Construction with Bloom Filters

Our index construction has much in common with other searchable encryption constructions that use Bloom filters. There is one Bloom filter index per database table row, and each keyword from the each column is inserted into the Bloom filter index. The column name or identifier is prepended to the keyword before insertion.

Our search index looks and acts like a Bloom filter in many ways. It is represented by an array of bits that are set to zero or one. Each word inserted sets h bits of this array to one. However, unlike Bloom filters, we do not use hash functions to determine which bits are set to one.

We strayed away from hash functions due to our goal of having uniform bit frequencies. When using deterministic keyed hash functions (where each row uses the same keys), the frequencies of the bits in the Bloom filter indexes are correlated with the frequencies of the words inserted.

Our way of preventing this correlation is to use a construction where each word that is inserted into the Bloom filter can set every possible bit in the index with uniform probability.

This property has an adverse effect on the false-positive rates. Our construction has techniques to counter these false-positives. The parameters to our construction have trade offs with security, performance and false-positive rates.

5.3.1 Easily Deployable Database Encrypted Search (EDDiES) Construction

For each word inserted, we permute an array of numbers $[1..s]$ where s is the index size in bits. This deterministic permutation function takes in a secret key, the search term and the array as inputs. This permutation represents the possible combinations of bit addresses that this word can set to one in the Bloom filter.

Parameter	Description
$\mathcal{IS}$	Size of inner lists
$\mathcal{OS}$	Number of outer lists
h	Number of bits set by each search term
s	Size of index, $s = \mathcal{IS} \cdot \mathcal{OS} \cdot h$
t	Search terms per index
x	Fake terms per index
λ	Distribution(s) of fake terms
n	The size of the keys

Table 5.1: Index Parameters

This permuted list is divided into $\mathcal{OS}$ smaller lists of equal size. One of these is selected at random. This smaller list is then divided into h lists of $\mathcal{IS}$ size. One number from each of these lists is chosen randomly. These are the index values that are set to one for this word in this Bloom filter.

There are $\mathcal{OS} \cdot \mathcal{IS}^h$ unique combinations of bits each word can set in the bloom filter. Searching involves the same permutation and dividing techniques, but the query looks for all the possible combinations of bits that this word can set. See Figure 5.1 for a formal definition of the encryption and search functions and Figure 5.2 for a simple example. The size or number of conjunctions and disjunctions in the queries grow with the security parameters:

$$((\mathcal{IS}-1)*h)+h-1)*\mathcal{OS}+\mathcal{OS}-1 \tag{5.5}$$

5.3.2 False-Positive Rate

Because the search term of our construction contains many conjunctions and disjunctions of possible bit addresses set to one, the false positive rate for our construction is different from that of a typical Bloom filter. Let p be the base probability of a bit set to one in our index. Then the false positive rate for our construction is:

$$(1-(1-(1-p)^{\mathcal{IS}})^h)^{\mathcal{OS}} \tag{5.6}$$

The false-positive rate is derived from Formula (5.2). First we calculate probability that any one of the bits in one $\mathcal{IS}$ set is set to one. This probability is $(1-(1-p)^{\mathcal{IS}})$. Then for a false positive to occur, this event has to happen in h of these sets. Thus we have $(1-(1-p)^{\mathcal{IS}})^h$. But we have $\mathcal{OS}$ outer sets, so a false positive could occur in any of these. Applying (5.2), we arrive at $(1-(1-(1-p)^{\mathcal{IS}})^h)^{\mathcal{OS}}$.

5.3.3 Parameters – Effects on Security and False Positive Rates

Below is a list of the Bloom filter parameters and the effects they have on security and false positive rates:

- t - number of items inserted into the Bloom filter. Increasing this number provides better security, while also increases the false-positive rate.
- x - number of fake tags inserted into the Bloom filter. Increasing x improves security. Increasing x also increases the false positive rate.

- $\mathcal{IS}$ - The size of the inner sets. Increasing $\mathcal{IS}$ improves security. With $\mathcal{IS} = 1$, and auxiliary data, this construction's security becomes similar to deterministic encryption security.

- $\mathcal{OS}$ - The number of outer sets. $\mathcal{OS}$ affects false-positive rates. Increasing this parameter decreases the false-positive rate. If $\mathcal{OS} = 1$, the false-positive rate will approach 1.

- h - This parameter affects false-positive rates. Increasing h decreases false-positive rates.

Increasing any of the $\mathcal{IS}, \mathcal{OS}, h$ parameters increases the size s of the Bloom filter indexes, which affects performance.

EDDiES Construction. Let PRS be a Pseudo Random Shuffle algorithm with key length n_1. Let Π' be an IND-CPA secure private key encryption scheme with message space $m \in \{0,1\}^*$ and key length n_0. Define a private-key searchable encryption scheme Π:

- Gen: on input $(1^{n_0}, 1^{n_1})$ run $\text{Gen}'(1^{n_0})$ receiving key k_0 and choose key $k_1 \in \{0,1\}^{n_1}$ uniformly.

- Enc: on input keys k_0, k_1, index parameters $(\mathcal{IS}, \mathcal{OS}, h, x, \lambda)$, and records D:
  For each $d_i \in D$:

  1. Initialize the Bloom filter BF.
  2. Randomly select x fake search words (ft), with each ft_j is drawn from probability distribution λ_j.
  3. Let $m_1, m_2, ..., m_t$ be the keywords in d_i
  4. For each $m \in m_1, m_2, ..., m_t, ft_1, ..., ft_x$
     (a) Let $bits \leftarrow \text{Trpdr}(k_1, (\mathcal{IS} \cdot \mathcal{OS} \cdot h), m)$
     (b) Divide $bits$ evenly into $\mathcal{OS}$ sublists, where each list has size $\mathcal{IS} \cdot h$ and $bits = (bits_1 || bits_2 || ... || bits_{\mathcal{OS}})$
     (c) Choose one of the lists randomly: $bitz \leftarrow \{bits_1, bits_2, , ..., bits_{\mathcal{OS}}\}$
     (d) Divide $bitz$ evenly into h lists, where each list has size $\mathcal{IS}$ and where $bitz = (bitz_1 || bitz_2 || ... || bitz_h)$
     (e) From each list $bitz_i$, randomly select one element:
     $y_1 \leftarrow bitz_1, y_2 \leftarrow bitz_2, ..., y_h \leftarrow bitz_h$
     (f) Set bits $bf_{y_1}, bf_{y_2}, ..., bf_{y_h}$ to 1.
  5. Output (Index, Ciphertexts): $(I, c_1, c_2, ..., c_t) := \left(BF,\ \text{Enc}'_{k_0}(m_1), ..., \text{Enc}'_{k_0}(m_t)\right)$

- Trpdr: on input key k_1, Bloom filter size s and message m, return $\text{PRS}(k_1, m, \{1..s\})$

- Dec: on input key k_0, ciphertext (I, c) output message $m_1, m_2, ..., m_t := \text{Dec}'_{k_0}(c_1), \text{Dec}'_{k_0}(c_2), ..., \text{Dec}'_{k_0}(c_t)$

- Search: on input keys (k_0, k_1), index parameters $(\mathcal{IS}, \mathcal{OS}, h)$ and a message m:

  – Let $bits \leftarrow \text{Trpdr}(k_1, (\mathcal{IS} \cdot \mathcal{OS} \cdot h), m)$
  – Divide $bits$ evenly into $\mathcal{OS}$ sublists, where each list has size $\mathcal{IS} \cdot h$ and $bitz = (bits_1 || bits_2 || ... || bits_{\mathcal{OS}})$
  – Divide each $bitz_i \in bitz$ evenly into h lists, where each list has size $\mathcal{IS}$ and where each $bitz_i = (bitz_{i_1} || bitz_{i_2} || ... || bitz_{i_h}$
  – Output search query:

$$\Big(\big(bf_{bitz_{1_{1_1}}} = 1 \vee bf_{bitz_{1_{1_2}}} = 1 \vee ... \vee bf_{bitz_{1_{1_{\mathcal{IS}}}}} = 1\big)$$

$$\wedge ... \wedge$$

$$\big(bf_{bitz_{1_{h_1}}} = 1 \vee bf_{bitz_{1_{h_2}}} = 1 \vee ... \vee bf_{bitz_{1_{h_{\mathcal{IS}}}}} = 1\big)\Big)$$

$$\text{OR} \ ... \ \text{OR}$$

$$\Big(\big(bf_{bitz_{\mathcal{OS}_{1_1}}} = 1 \vee bf_{bitz_{\mathcal{OS}_{1_2}}} = 1 \vee ... \vee bf_{bitz_{\mathcal{OS}_{1_{\mathcal{IS}}}}} = 1\big)$$

$$\wedge ... \wedge$$

$$\big(bf_{bitz_{\mathcal{OS}_{h_1}}} = 1 \vee bf_{bitz_{\mathcal{OS}_{h_2}}} = 1 \vee ... \vee bf_{bitz_{\mathcal{OS}_{h_{\mathcal{IS}}}}} = 1\big)\Big)$$

Figure 5.1: Easily Deployable Database Encrypted Search, Decryption and Search

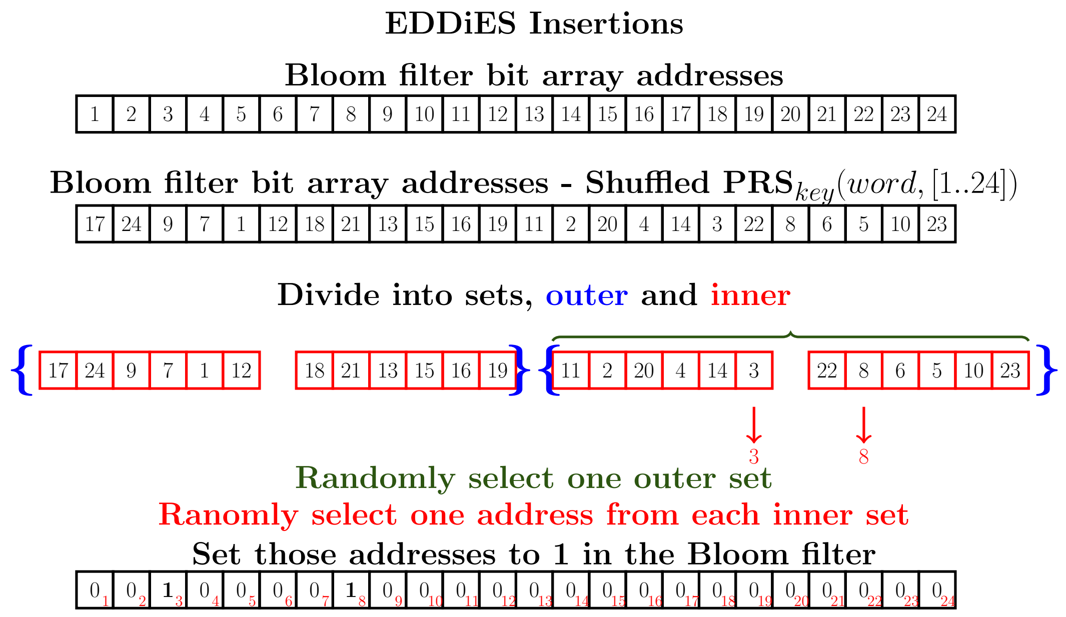


Figure 5.2: Insertion, $\mathcal{OS} = 2, \mathcal{IS} = 6, h = 2$

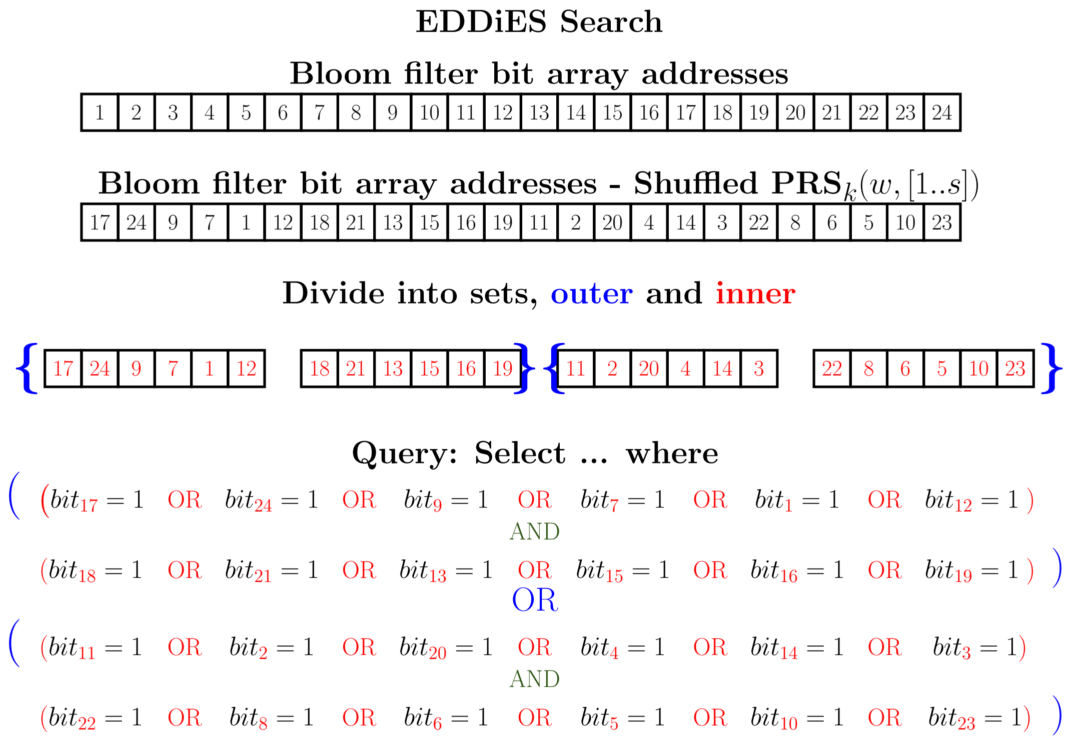

EDDiES Search
Bloom filter bit array addresses
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
Bloom filter bit array addresses - Shuffled PRS$_k(w, [1..s])$
17 24 9 7 1 12 18 21 13 15 16 19 11 2 20 4 14 3 22 8 6 5 10 23
Divide into sets, outer and inner
{ 17 24 9 7 1 12 18 21 13 15 16 19 } { 11 2 20 4 14 3 22 8 6 5 10 23 }
Query: Select ... where
(($bit_{17} = 1$ OR $bit_{24} = 1$ OR $bit_9 = 1$ OR $bit_7 = 1$ OR $bit_1 = 1$ OR $bit_{12} = 1$)
AND
($bit_{18} = 1$ OR $bit_{21} = 1$ OR $bit_{13} = 1$ OR $bit_{15} = 1$ OR $bit_{16} = 1$ OR $bit_{19} = 1$))
OR
(($bit_{11} = 1$ OR $bit_2 = 1$ OR $bit_{20} = 1$ OR $bit_4 = 1$ OR $bit_{14} = 1$ OR $bit_3 = 1$)
AND
($bit_{22} = 1$ OR $bit_8 = 1$ OR $bit_6 = 1$ OR $bit_5 = 1$ OR $bit_{10} = 1$ OR $bit_{23} = 1$))

Figure 5.3: Search, $\mathcal{OS} = 2, \mathcal{IS} = 6, h = 2$

5.4 Leakage

In order to show our construction meets the SSE security definition, we first must define its leakage. Our construction has the same access pattern leakage that all SSE constructions share. The primary difference in the leakage profiles is the index leakage. Unlike tradition SSE constructions, we do not encrypt our indexes.

First, since our system is designed for a database table, we can assume the size of the records or documents are all the same. If one of the columns has variable size, such as varchar, we will assume the ciphertexts will be padded to equal lengths. With SSE designed for document collections, the size of the encrypted document is part of the leakage.

Our construction sets each bit in the index to one with equal probability. Thus frequencies of individual bits set to one in the Bloom filter do not leak anything.

5.4.1 $\mathcal{L}_0(I)$ Leakage

The first part of the leakage comes from the co-occurrence frequencies. These consist of multi-occurrences of two bits up to $(t+x)\cdot h$ bits, since that is the highest number of bits that will be set to one in any index.

Definition 5.7 (Bloom Filter Multi-occurrence Frequency $\mathcal{F}(\cdot)$)**.**

- *Let BF be a list of Bloom filters.*
- *Let $\mathcal{S}$ be the set of Bloom filter addresses $\{bf_1, bf_2, ..., bf_s\}$*
- *Let γ be a subset of $\mathcal{S}$ where $|\gamma| \leq (t+x)\cdot h$*

Then the Bloom filter multi-occurrence frequency $\mathcal{F}(\gamma)$ is the frequency that the bit addresses in γ are set to one in the same Bloom filter in BF.

Our construction does not set these sets of bits to one with the same frequency because each word has some sets of bits that it cannot set to one. The construction randomly selects one bit address from each $\mathcal{IS}$ set equal to one. Thus it is impossible for word w to assign one to more than one bit address from each of these sets. Likewise, it is not possible for a word to set bit addresses together that appear in different $\mathcal{OS}$ sets. If word w can set $\{bf_i, bf_x, ...\}$ to one in the same Bloom filter, then the frequency of those bits occurring together should be slightly higher than if word w does not set those bits to one. Thus one part of the leakage $\mathcal{L}_0(I)$ is the multi-occurrence frequencies of bits set to one.

Definition 5.8 (Index Leakage, $\mathcal{L}_0(I)$)**.** *Each Bloom filter will have at most* $(t+fk)\cdot h$ *bits set. The* $\mathcal{L}_0(I)$ leakage *is the multi-occurrence frequencies for all subsets of bit addresses from size 2 up to* $(t+x)\cdot h$.

$$\{Y \subset \{bit_1 ... bit_s\} : |Y| \geq 2, |Y| \leq h \cdot (t+x)\}$$

$$\mathcal{L}_0(I) = \forall \gamma \in Y, \mathcal{F}(\gamma)$$

5.4.2 $\mathcal{L}_1(I)$ Leakage

Since the adversary can choose the list of records that are encrypted in the security model, they will see the individual Bloom filter indexes corresponding with each record. They will then know the lists of words that each index is associated with.

Definition 5.9 (Index Leakage, $\mathcal{L}_1(I)$)**.** *The* $\mathcal{L}_1(I)$ leakage *is a list of tuples, one tuple for each document that consists of the the list of words in that document and* *the Bloom filter index for that document.*

$$\begin{aligned}\mathcal{L}_1(I) = &(w_{(1,1)}, w_{(1,2)}, ..., w_{(1,i)}, BF_1),\\ &(w_{(2,1)}, w_{(2,2)}, ..., w_{(2,i)}, BF_2),\\ &...,\\ &(w_{(n,1)}, w_{(n,2)}, ..., w_{(n,i)}, BF_n)\end{aligned}$$

Where n is the number of records and i is the number of columns.

5.4.3 Trapdoor Combinations and Probabilities

An understanding of these trapdoor combinations and probabilities is essential for analyzing the leakage and security of this construction.

Number of Possible Trapdoors. Even though there are $s!$ different permutations of a set of size s, many of them are equivalent. For example, if $\mathcal{OS} = 1, \mathcal{IS} = 2, h = 2$, then the permutation $[2, 4, 3, 1]$ is equivalent to $[4, 2, 3, 1]$ for our construction and $[2, 4, 3, 1]$ is also equivalent to $[3, 1, 2, 4]$, since they result in the same bits being set. So while there are $4! = 24$ different permutations, the number of unique trapdoors for a given set of parameters is:

$$\prod_{i=0}^{(\mathcal{OS}\cdot h)-1} \binom{S - (i \cdot \mathcal{IS})}{\mathcal{IS}} \tag{5.7}$$

The reason the formula is not simply $S!$ is because each set of $\mathcal{IS}$ bits is unordered. We arrive at Formula (5.7) by first selecting the first set of bits, which there are $\binom{S}{\mathcal{IS}}$ possible ways to do so. Then we select the 2nd set with $\binom{(S-\mathcal{IS})}{\mathcal{IS}}$ possibilities and so on.

Number of Possible Trapdoors for a given Index. We know that for a given Bloom filter index, each trapdoor sets h bits in that index. Assuming that $h \cdot t$ are set in each Bloom filter, Formula (5.8) shows the number of possible sets of size h.

$$\binom{(h \cdot t)}{h} \tag{5.8}$$

Given h bits that are part of a trapdoor in the same $\mathcal{OS}$ set, the number of possible combinations for the rest of the bit addresses in the $\mathcal{OS}$ set is:

$$\prod_{i=0}^{h-1} \binom{(s - h - ((\mathcal{IS} - 1) \cdot i))}{(\mathcal{IS} - 1)} \tag{5.9}$$

Then in the other remaining $\mathcal{OS}$ sets the number of possible bit address combinations is:

$$\prod_{i=h}^{(\mathcal{OS} \cdot h)-1} \binom{s - (i \cdot \mathcal{IS})}{\mathcal{IS}} \tag{5.10}$$

Thus, given $h \cdot t$ bits set in a BF, combining equations (5.8),(5.9), and (5.10) gives us the total number of possible trapdoors could have set those $h \cdot t$ bits:

$$\binom{(h \cdot t)}{h} \cdot \prod_{i=0}^{h-1} \binom{(s - h - ((\mathcal{IS} - 1) \cdot i))}{(\mathcal{IS} - 1)} \cdot \prod_{i=h}^{(\mathcal{OS} \cdot h)-1} \binom{s - (i \cdot \mathcal{IS})}{\mathcal{IS}} \tag{5.11}$$

The percentage of possible trapdoors that can set $h \cdot t$ bits in a Bloom filter is:

$$\begin{aligned}
&\frac{\binom{h\cdot t}{h}\cdot\prod_{i=0}^{h-1}\binom{s-h-((\mathcal{IS}-1)\cdot i)}{(\mathcal{IS}-1)}\cdot\prod_{i=h}^{(\mathcal{OS}\cdot h)-1}\binom{s-(i\cdot\mathcal{IS})}{\mathcal{IS}}}{\prod_{i=0}^{(\mathcal{OS}\cdot h)-1}\binom{s-(i\cdot\mathcal{IS})}{\mathcal{IS}}} \\
= &\frac{\binom{h\cdot t}{h}\cdot\prod_{i=0}^{h-1}\binom{s-h-((\mathcal{IS}-1)\cdot i)}{(\mathcal{IS}-1)}}{\prod_{i=0}^{h-1}\binom{s-(i\cdot\mathcal{IS})}{\mathcal{IS}}} \qquad (5.12)
\end{aligned}$$

Asymptotic Behavior. As the parameters other than t grow, this ratio gets very small. For example, this ratio with parameters of $\mathcal{OS} = 10, \mathcal{IS} = 100, h = 30, t = 12 \approx 1.68 \times 10^{-62}$. As t increases, (assuming the rest of the parameters are fixed and that $h \cdot t \leq s$ since you cannot set more than s bits in the Bloom filter), this ratio approaches 1.0, but the false positive rate approaches 1.0 as well. With values of t that produce practical false positive rates, this ratio will always approach zero as the other parameters are increased.

The conclusion here is as the parameters other than t grow, there will be a very large number of trapdoors that match any given index, but the ratio of trapdoors that match the index to total trapdoors will be small.

5.4.4 SSE Security

To prove the EDDiES construction meets SSE security, we will create a simulator that when given the leakage as input, creates an output that is indistinguishable from the output of the construction. The output is defined in Definition 5.6 as a tuple of indexes, ciphertexts, and trapdoors, $v = (I, c, t)$.

Theorem 5.1 (SSE Security). *If* PRS *is a pseudo-random permutation and Enc is CPA secure, then* EDDiES *is non-adaptively SSE secure.*

Proof. Let S be a simulator that when given the leakage $\tau(H)$, generates the tuple (I^*, c^*, t^*).

- Simulating c: Set c_i^* to an l-bit string chosen uniformly random.
- Simulating I: Set I_i^* according to the value in the $\mathcal{L}_1(I)$ leakage.
- Simulating t: If $|q| > 1$ then mark each bit address in each index as unused. For each $q_i \in q$ in random order, choose a trapdoor:
  1. Start with an empty Trapdoor $\mathcal{TD}$, i.e. all $\mathcal{OS}$ and $\mathcal{IS}$ sets are empty.
  2. Choose each index I_j^*, in random order, that is in the access pattern of q_i. For each I_j^*:
     (a) Let $\mathcal{U}$ represent the set of unselected bit addresses in I_j^* that are set to 1. Mark all of the bit addresses in this set as unselected. Let $\mathcal{U}'$ equal the subset of unselected bit addresses.
     (b) Randomly select an $\mathcal{OS}$ set from $\mathcal{TD}$.
     (c) For each $\mathcal{IS}$ set in the $\mathcal{OS}$ set chosen above:
        i. Mark all bit addresses bf_j from $\mathcal{U}$ that are not in the $\mathcal{OS}$ set as temp selected. Mark all bit addresses in the $\mathcal{OS}$ set, but not in the $\mathcal{IS}$ set as temp selected.
        ii. If the $\mathcal{IS}$ set is full, randomly select from the set of addresses in the $\mathcal{IS}$ set that is also in the unselected bits from $\mathcal{U}$, $bf_l \leftarrow (\mathcal{IS} \cap \mathcal{U}')$. If this set is empty, start over with step 1. Otherwise place this address in the $\mathcal{IS}$ set and remove it from $\mathcal{U}$.
        iii. If the $\mathcal{IS}$ set is not full, randomly select a bit address from the set of addresses in $\mathcal{U}'$, $bit_l \leftarrow \mathcal{U}'$. Place this address in the $\mathcal{IS}$ set and remove it from $\mathcal{U}$.

iv. Set all temp selected bit addresses in $\mathcal{U}$ to unselected.

3. For each $\mathcal{IS}$ set that is not full, randomly select from the set of bit addresses not in $\mathcal{TD}$ and add them until full.

4. For each I_l^* not in access pattern of q_i, if $\mathcal{TD}$ matches I_l^*, that is if a query produced from $\mathcal{TD}$ would result in I_l^* being included in the access pattern, then start over with step 1.

Let v be the outcome of a $\mathbf{Real}_{SSE,A}(k)$ vs $\mathbf{Sim}_{SSE,A}(k)$ experiment. We will show that each part of v, (I, c, t) is indistinguishable to any polynomial distinguisher D that receives v and v^*.

- (t_i, t_i^*) t_i is the evaluation of $\mathrm{PRS}_K(w_i)$ and t_i^* is a random selection of a trapdoor that has the same access pattern as t_i. The definition of PRS assures that an adversary cannot distinguish between the outcome of a PRS and a random permutation. Our algorithm randomly selects a $\mathcal{TD}$ from the set of $\mathcal{TD}$'s that match the access pattern.

  While the algorithm $\mathcal{S}$ does select a random $\mathcal{TD}$ that matches the access pattern, there is some probability that it will fail. There are three scenario in the algorithm for potential failure:

  - In step 2(c)ii, the $\mathcal{IS}$ set is full. In the worst case scenario, the probability of the algorithm succeeding then is:

    $$Pr[|\mathcal{IS} \cap \mathcal{U}| \geq 1]$$

    There are $\binom{s-\mathcal{IS}}{|\mathcal{U}|}$ ways of choosing the values in the set $\mathcal{U}$ where the size of the intersection is zero out of $\binom{s}{|\mathcal{U}|}$ possible. Thus the probability the size of the intersection is at least 1:

$$1.0 - \frac{\binom{s-\mathcal{IS}}{|\mathcal{U}|}}{\binom{s}{|\mathcal{U}|}}$$

As $\mathcal{IS}$ and $|\mathcal{U}|$ increase, the probability that the size of the intersection is greater than 1 grows very close to 1.0. The adversary can limit $|\mathcal{U}|$ in the SSE game by issuing queries for each keyword in each record. However we can counter the adversary's attempts to reduce $|\mathcal{U}|$ with fake tags. If we have x fake tags, then $|\mathcal{U}| \geq x \cdot h$. Thus, increasing the parameters that have the highest effect on security – $\mathcal{IS}, t, x$ – also greatly increases the probability that the algorithm succeeds to find a $\mathcal{TD}$.

– The second scenario is in step 2(c)iii where all but two of the $\mathcal{IS}$ sets are full (if all but one are full, then by default the last is full, because $\binom{\mathcal{IS}}{\mathcal{IS}} = 1$), which will minimize the size of $\mathcal{U}'$. Here, the probability of the algorithm succeeding is:

$$Pr[|\mathcal{U}'| \geq 1] = 1.0 - \frac{\binom{s-(\mathcal{IS}+1)}{|\mathcal{U}|}}{\binom{s}{|\mathcal{U}|}}$$

This probability is almost the same as the first scenario, but will always have a lower probability of failure.

– There is also a possibility of failure in Step 4. However, Formula (5.12) shows that this likelihood is low.

- (c_i, c_i^*): The CPA security of Π' ensures that c_i and , c_i^* are indistinguishable.
- (I, I^*): Since I^* is created from the $\mathcal{L}_1$ leakage, I and I^* are identical.

□

5.5 Leakage Implications

We have shown that our construction meets the SSE security definitions; that is, we have quantified its leakage and shown that it leaks nothing else. For the more tradition SSE constructions, this is often enough.

Traditional SSE leaks the document sizes, access and search patterns and nothing else. The additional leakage from our indexes needs analysis. Depending on the data and the index parameters, this leakage could allow near perfect success rates with inference attacks, or it might prove highly resistant to these attacks.

In the SSE security model, the adversary is not given power to use the leakage. However in the real world, they will do so. Thus, our leakage analysis looks at what if an adversary is able to use the $\mathcal{L}_0$ and $\mathcal{L}_1$ leakage in the SSE game. Removing these leakage restrictions, it is very likely that they win the SSE game with greater than negligible probability. That does not mean they can learn anything useful. The adversary may or may not be able to exploit the leakage. Our leakage analysis will focus on the adversary's concrete probability to learn an unknown trapdoor or partial trapdoor from the leakage.

Definition 5.10 (Partial Trapdoor)**.** *Let w be a word in document set D and let γ be a subset of Bloom filter addresses $\{bf_1, bf_2, ..., bf_s\}$ for Construction 5.1. Let $\mathcal{TD}$ be the trapdoor for w. We say γ is a partial trapdoor for word w if each address in γ is in the same $\mathcal{OS}$ outer set of $\mathcal{TD}$, and each address in γ are in different inner $\mathcal{IS}$ sets of $\mathcal{TD}$. This partial trapdoor is denoted $\gamma \in \mathcal{TD}(w)$.*

We look at the leakage assuming the adversary plays the SSE game, but is allowed to use the $\mathcal{L}_0, \mathcal{L}_1$ leakage to attempt to learn a partial trapdoor of a word w.

5.6 $\mathcal{L}_0(I)$ Leakage, Multi-Occurrence Frequencies

When an adversary observes a multi-occurrence frequency of γ, they can attempt to use this to figure out which plaintext words can set these bit addresses to one, which provides partial information about the trapdoor for that word.

- Let $\mu = |\gamma|$
- Let $b(w) = \frac{1}{sc_1^{\mu} \cdot sc_2}$ if word w sets the multi-occurrence, zero otherwise. $\frac{1}{sc_1^{\mu} \cdot sc_2}$ is the frequency that w will set this multi-occurrence.
- Let z equal the number of unique words in the database. Note this is not t, the number of words inserted into the Bloom filter.
- Let $f(\cdot)$ be a function that returns the frequency of a word in the document collection and $\mathcal{F}(\cdot)$ be a function that returns the multi-occurrence frequency of bit addresses in the Bloom filter index. Then

  $$\mathcal{F}(\gamma) = p^{\mu} \cup \bigcup_{w \in z} f(w) \cdot b(w) \tag{5.13}$$

  The adversary needs to find the correct assignment of $b(w_i)$ values to decipher which words set these bits. This formula is not as straightforward as it seems. Some of the events are mutually exclusive and some are not.
- To make this problem even harder, we can insert fake words into each Bloom filter. If frequency of these fake words is chosen randomly, then the adversary will also not have knowledge of some of the $f(w_i)$ frequencies.

With z words in the database, there are 2^z possible assignments of $b(w_1), b(w_2), ..., b(w_z)$ because each $b(w_i)$ is one of two values, zero or $\frac{1}{sc_1^{\mu} \cdot sc_2}$. However, not all of the solutions have the same probability.

There are $\binom{h}{u}$ different ways to choose the $\mathcal{IS}$ sets. After choosing the $\mathcal{IS}$ sets, there are $\mathcal{IS}^u$ unique combinations of bit addresses. Combined with $\mathcal{OS}$ outer sets, the number of possible multi-occurrences of size μ that a word can set is $\binom{h}{u} \cdot \mathcal{IS}^u \cdot \mathcal{OS}$. With $\binom{s}{\mu}$ total multi-occurrences, the number of $b(w_i)$ set to non-zero values follows a binomial distribution $b(x; n, P)$ with

$$P = \frac{\binom{h}{u} \cdot \mathcal{IS}^u \cdot \mathcal{OS}}{\binom{s}{u}}, \text{ and } n = z \tag{5.14}$$

For example, assume a database with a corpus of three words w_a, w_b, w_c. For each multi-occurrence γ, there are $2^3 = 8$ possible assignments of $b(w_a), b(w_b), b(w_c)$ as seen in figure 5.4. If we let α equal the number of $b(w_i)$ set to nonzero, then the attacker can calculate the probability of each assignment equaling $\mathcal{F}(\gamma)$.

$$\boxed{\begin{array}{l}
Pr\big[\mathcal{F}(\gamma) = p^2\big] \cdot Pr[\alpha = 0] \\
Pr\big[\mathcal{F}(\gamma) = (f(w_a) \cdot b(w_a) \cup p^2)\big] \cdot Pr[\alpha = 1] \\
Pr\big[\mathcal{F}(\gamma) = (f(w_b) \cdot b(w_b) \cup p^2)\big] \cdot Pr[\alpha = 1] \\
Pr\big[\mathcal{F}(\gamma) = (f(w_c) \cdot b(w_c) \cup p^2)\big] \cdot Pr[\alpha = 1] \\
Pr\big[\mathcal{F}(\gamma) = (f(w_a) \cdot b(w_a) \cup f(w_b) \cdot b(w_b) \cup p^2)\big] \cdot Pr[\alpha = 2] \\
Pr\big[\mathcal{F}(\gamma) = (f(w_a) \cdot b(w_a) \cup f(w_c) \cdot b(w_c) \cup p^2)\big] \cdot Pr[\alpha = 2] \\
Pr\big[\mathcal{F}(\gamma) = (f(w_b) \cdot b(w_b) \cup f(w_c) \cdot b(w_c) \cup p^2)\big] \cdot Pr[\alpha = 2] \\
Pr\big[\mathcal{F}(\gamma) = (f(w_a) \cdot b(w_a) \cup f(w_b) \cdot b(w_b) \cup f(w_c) \cdot b(w_c) \cup p^2)\big] \cdot Pr[\alpha = 3]
\end{array}}$$

Figure 5.4: w_a, w_b, w_c $\mathcal{L}_0$ Example

Using the $\mathcal{L}_0$ leakage along with auxiliary $f(w_i)$ data, they can attempt to find an assignment for $b(w_i)$ of the above equations that maximizes the probability.

This assignment is a complex instance of a subset-sum problem. The subset sum is an NP-complete problem. However there are some polynomial-time approximations and small instances are easily solvable as well. Thus if there were only one subset-sum solution to each of these frequencies, the adversary would likely break this construction if they had quality auxiliary data for inference attacks.

This type of attack is one reason why we add fake words with a distribution unknown to the adversary to each Bloom filter. With fake words of unknown frequency, finding an assignment of $b(w_i)$ becomes a subset-sum problem with numerous solutions as any set of known words that is less than the desired frequency is a solution when one or more unknown frequencies are included. Also note it is important that the expected value of α is greater than one. Otherwise the attacker's problem becomes much easier, similar to the best case scenario below. It is simple to ensure $\alpha > 1$ by tweaking the parameters or simply adding more fake words.

The best case scenario for the adversary is with $f(w_a) = 1.0$ and the rest of the $f(w_i)$ are negligible or the adversary knows the $b(w_i)$ values. Now the adversary only has determine which is most likely:

$$Pr[\mathcal{F}(\gamma) = \frac{1}{sc_1^u \cdot sc_2} \cup p^u] \tag{5.15}$$

$$Pr[\mathcal{F}(\gamma) = p^u] \tag{5.16}$$

Where p is the overall probability of a bit being set to one.

The probability of an adversary distinguishing between the worst case scenario of $\frac{1}{\mathcal{IS}^u \cdot \mathcal{OS}} \cup p^u$ and p^u is bounded by the statistical distance between these two binomial probabilities. The $\mathcal{IS}, \mathcal{OS}$ parameters are the primary difference maker in minimizing the statistical distance: as they grow, the probabilities grow closer together. Our goal with setting the parameters is to minimize $\frac{1}{\mathcal{IS}^u \cdot \mathcal{OS}}$ while ensuring that $p^u \gg \frac{1}{\mathcal{IS}^u \cdot \mathcal{OS}}$, minimizing the distance. Ideally this statistical distance would be negligible, but that is not the case here. As our parameters grow, the statistical distance grows smaller, but as the size of the database grows, this distance grows larger.

However, non-negligible, but small distance is still useful to the defender. For example, assume the probability of the adversary correctly distinguishing Probabilities (5.15) and (5.16) is 0.034. This probability is well short of typical cryptographic standards. Table 5.2 shows an example of these probabilities with varrying sizes of n and u However, this lone multi-occurrence is not very useful to an attacker. As Chapter 5.8 shows, the false-positive rate for a partial match of a trapdoor is likely to be too high to be of any use to an adversary. Even getting a multi-occurrence of size 15 still likely means a false positive rate of over 60%. Also, one multi-occurrence is only one small part of a trapdoor. The number of multi-occurrences of size u is:

$$\binom{h}{u} \cdot \mathcal{IS}^u \cdot \mathcal{OS}$$

To get the complete trapdoor, the adversary must learn most of the multi-occurrences.

	$u = 2$	$u = 6$	$u = 10$
$n = 1,000,000$	0.11	$2.6x10^{-10}$	$7.8x10^{-20}$
$n = 100,000$	0.034	$2.6x10^{-11}$	$7.6x10^{-21}$

Table 5.2: Worst Case Statistical Distance Example, $\mathcal{IS} = 240, \mathcal{OS} = 10, h = 20$

Multiple Multi-Occurrences. A multi-occurrence of size $u = 2$ is easiest for an adversary to distinguish, but also the least useful. Larger multi-occurrences can be constructed from many smaller. For example, if the adversary learns that word w sets multi-occurrence $(bf_i, bf_j), (bf_j, bf_k), (bf_i, bf_k)$, then it learns that w sets (bf_i, bf_j, bf_k). The probability of distinguishing multiple multi-occurrences is not quite independent because some combinations are not possible. For example, let $\mathcal{IS} = 2, h = 2, \mathcal{OS} = 1$. There are $\binom{4}{2} = 6$ possible size 2 multi-occurrences. Then the $PR[(bf_1, bf_2), (bf_1, bf_3), (bf_1, bf_4) \in \mathcal{TD}(w)] = 0$ because it is not possible for any word to set those multi-occurrences with the chosen parameters.

Specifically, a set of multi-occurrences cannot share an index location bf_i more than $\mathcal{IS} \cdot (h-1)$ times. While this extra information helps the adversary, it only helps if they have already identified $\mathcal{IS} \cdot (h-1)$ multi-occurrences.

It seems natural to think that if $(bf_i, bf_j) \in \mathcal{TD}(w)$ and $\mathcal{F}(bf_i, bf_j) \approx \mathcal{F}(bf_k, bf_l)$ then it is likely that $(bf_k, bf_l) \in \mathcal{TD}(w)$. Since the adversary controls the document collection D in the SSE game, they can ensure this is the case. Fake tags come to the rescue again.

5.6.1 Fake Tag Distribution

Let z be the number of fake tags in our corpus. Each fake tag is assigned a random frequency drawn from a distribution. In our case, this distribution is the exponential distribution with parameter λ.

Each of these tags can cause the event E of setting u bits in a multi-occurrence in the Bloom filter index. Thus the probability of this event happening is the sum of these frequencies, which is an Erlang distribution. An Erlang distribution has two parameters, a positive integer k, called the shape, and a positive real number λ, the rate. It is the sum of k independent exponential variables with parameter λ.

Each fake tag has the same probability of producing this event. So the number of fake tags that produces event E follows a binomial distribution with probability q. The overall probability of event E is a sum of independent random exponential variables, where the number of terms in the summation is itself random with binomial probability q. This is the hyper-Erlang distribution.

The hyper-Erlang distribution takes an Erlang distribution E_i with probability q_i. The probability density function for the hyper-Erlang distribution with binomial probability q is:

$$
\begin{aligned}
f(x) &= \sum_{i=1}^{z} q_i \cdot E_{l_i}(x) && (5.17)\\
&= \sum_{i=1}^{z} q_i \cdot \frac{\lambda^i x^{(i-1)} e^{-\lambda x}}{(i-1)!} && (5.18)\\
&= \sum_{i=1}^{z} \binom{z}{i} q^i (1-q)^{z-i} \cdot \frac{\lambda^i x^{(i-1)} e^{-\lambda x}}{(i-1)!} && (5.19)\\
&\text{where } q = \frac{\mathcal{OS} \cdot \binom{h}{u} \cdot \mathcal{IS}^u}{\binom{s}{u}}
\end{aligned}
$$

with each $q_i > 0$ and $\sum_{i=1}^{n} q_i = 1$. Each $E_{l_i}(x)$ is an Erlang distribution with l_i shape and rate λ_i. For our specific case, $l_i = i$ and all of the λ variables are the same, $\lambda_i = \lambda_{i+1}$.

The multi-occurrence frequencies are now changed by a random frequency from Formula (5.19). If we let FK be the likelihood that any fake tag sets this multi-occurrence, then the worst case scenario is now:

$$Pr[\mathcal{F}(\gamma) = \frac{1}{sc_1^u \cdot sc_2} \cup p^u \cup FK] \qquad (5.20)$$

$$Pr[\mathcal{F}(\gamma) = p^u \cup FK] \qquad (5.21)$$

Using the statistical distance upper bound on the likelihood of distinguishing between two probabilities, distinguishing between Probabilities (5.20) and (5.21) is the same as (5.15) and (5.16), because both sets of distributions vary by the same $\frac{1}{sc_1^u \cdot sc_2}$.

5.6.2 Number of Solutions for All Frequencies

Using all of the frequencies gives the adversary a little more leverage. They know that $b(w_i)$ is set exactly $\binom{h}{u} \cdot \mathcal{IS}^u \cdot \mathcal{OS}$ times. Thus there are:

$$\binom{\binom{s}{u}}{\binom{h}{u} \cdot \mathcal{IS}^u \cdot \mathcal{OS}}$$

different possible assignments of $b(w_i)$.

Also note that any word w sets $\binom{h}{u}$ multi-occurrences in each Bloom filter. Thus it sets $n \cdot \binom{h}{u}$ multi-occurrences, not necessarily distinct. It is easy to set the parameters so that given a database size n, a word will not set all possible multi-occurrences. Thus, while a word might set a multi-occurrence, it might not in a specific database.

One could intentionally set the parameters so that the expected number of times any word sets a multi-occurrence is less than or equal to one. If it were a lot less than one, then the multi-occurrence frequencies would depend more on the randomness of which multi-occurrences were actually set by the word insertions than by the actual word frequencies. To achieve this property, we simply need to set the $\mathcal{IS}, \mathcal{OS}$ parameters such that:

$$\frac{n \cdot \binom{h}{u}}{\binom{h}{u} \cdot \mathcal{IS}^u \cdot \mathcal{OS}} = \frac{n}{\mathcal{IS}^u \cdot \mathcal{OS}} < 1$$

5.6.3 Known Trapdoor Effect

When an adversary makes a query and is given a trapdoor, they are given one of the $b(w_i)$ solutions, so it makes that problem a little smaller. The worst-case scenario already assumes the attacker either has knowledge of all of the $b(w_i)$ except for the word for which they are attempting to distinguish a partial trapdoor or the frequencies are tiny. Thus knowledge of trapdoors helps the adversary in the general case, such as Example 5.4, but it does not change the worst case scenario.

5.6.4 $\mathcal{L}_0(I)$ Conclusion

While the general case of the $\mathcal{L}_0$ leakage analysis involves complex probabilities, the worst case security is a relatively simple calculation of the statistical distance between Probabilities 5.15 and 5.16. This worst case formula also has the advantage that it is not affected by other trapdoors known by the adversary.

5.7 $\mathcal{L}_1$ Leakage – Chosen Plaintexts

The SSE security game allows the adversary to mount a "chosen plaintext attack". The adversary chooses the plaintexts for encryption, with some restrictions. In our case, with a structured database, the number and type of words must match the database schema.

The game does not allow multiple accesses to an encryption oracle: the adversary cannot encrypt one row, see the results, then craft another. They must choose the entire list of plaintexts at once.

This attack is still powerful. It is simple to demonstrate that with poorly chosen parameters, an adversary can win the 5.6 game. For example, with $t = 1, z =$

$0, \mathcal{IS} = 1$ parameter settings, each plaintext will only set one of $\mathcal{OS}$ sets of h bits with each encryption. The construction with these parameters is similar to deterministic encryption and its insecurity is easy to see.

It seems natural that as $t, z, \mathcal{IS}$ grow, it is harder for the adversary to distinguish between two encrypted data sets or it will take much larger data sets to distinguish.

5.7.1 Number of Possible Trapdoors for a Given Bloom Filter Index

In the SSE security game, the adversary chooses the list of plaintext documents that the challenger will encrypt. Thus for a given plaintext, they will know which records the plaintext appears in and which it does not. If they can find a trapdoor that matches, then they have narrowed down the possible trapdoors that could match that plaintext.

However, just enumerating the number of trapdoors that match one Bloom filter index can take too long for a computationally bound adversary.

As an example, with $\mathcal{IS} = 20, \mathcal{OS} = 10, h = 30, t = 5$, given one Bloom filter index where $h \cdot t$ bits are set to one, there are $1.88x10^{1622}$ possible trapdoors that could have set those bits.

However, we are concerned about preventing the adversary from learning a partial trapdoor. Given an index I, one word w sets $\binom{h}{u}$ multi-occurrences in that index. The ability of an adversary to properly guess which multi-occurrences were set by word w depends on t, how many words were inserted into the index. With $t = 1$, word w is the only word inserted and thus it set all of the multi-occurrences in I. As t grows, the percent of multi-occurrences in I set by w shrinks.

$$\frac{\binom{h}{u}}{\binom{h \cdot t}{u}} \tag{5.22}$$

Figure 5.5 illustrates an example of the effect of increasing t with other parameters of $h = 40, u = 2$.

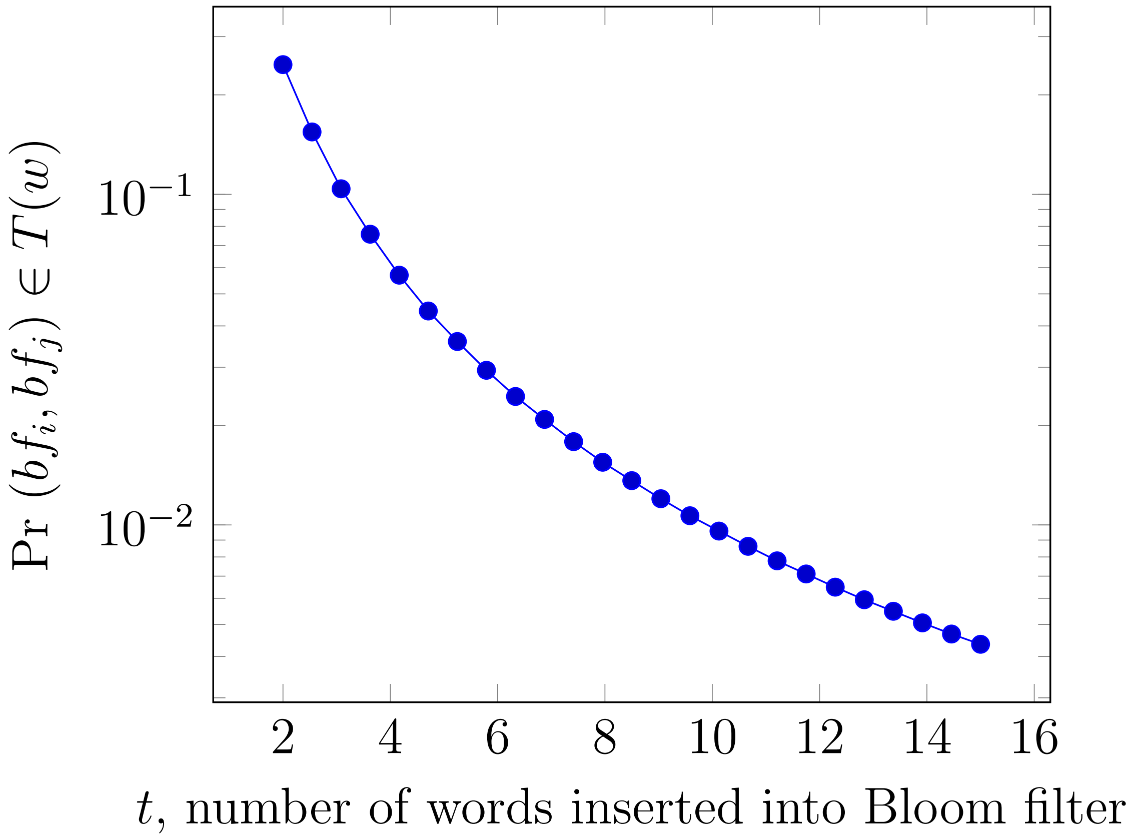


Figure 5.5: Co-occurrence Frequency Example, $h = 40, u = 2$

With such a low probability of the partial $\mathcal{TD}$ belonging to a keyword, the adversary now is reduced to looking at patterns: how often a partial $\mathcal{TD}$ occurs in multiple indexes, which is the $\mathcal{L}_0$ leakage.

Known Trapdoors. In the SSE security Definition 5.6, the adversary can choose words to query and thus receive the trapdoors of those words. They could choose these queries such that in an index, they have the trapdoors for all but one word, thus making the effective value of t one. This adversarial power is another reason for the fake tags. The challenger can setup their system with x fake tags inserted into each index such that the value from Formula (5.22) where $x + 1$ is substituted for t is within their comfort level.

5.8 Partial Learning

What if an adversary learns some of the multi-occurrences for a word, thus a partial trapdoor? As shown in Table 5.2, it is easy to make this probability small, but not necessarily possible to make it small by typical cryptographic standards.

The false-positive rate comes into play to limit the usefulness of this partial trapdoor to an attacker. In the example shown in Figure 5.6, if an adversary learns one partial trapdoor of size 15 for a word, this partial trapdoor produces in a false positive rate of over 60% of the database.

Also, learning one multi-occurrence is only a fraction of the total trapdoor for a given word. The number of multi-occurrences of size u for a given word is:

$$\binom{h}{u} \cdot \mathcal{IS}^u \cdot \mathcal{OS}$$

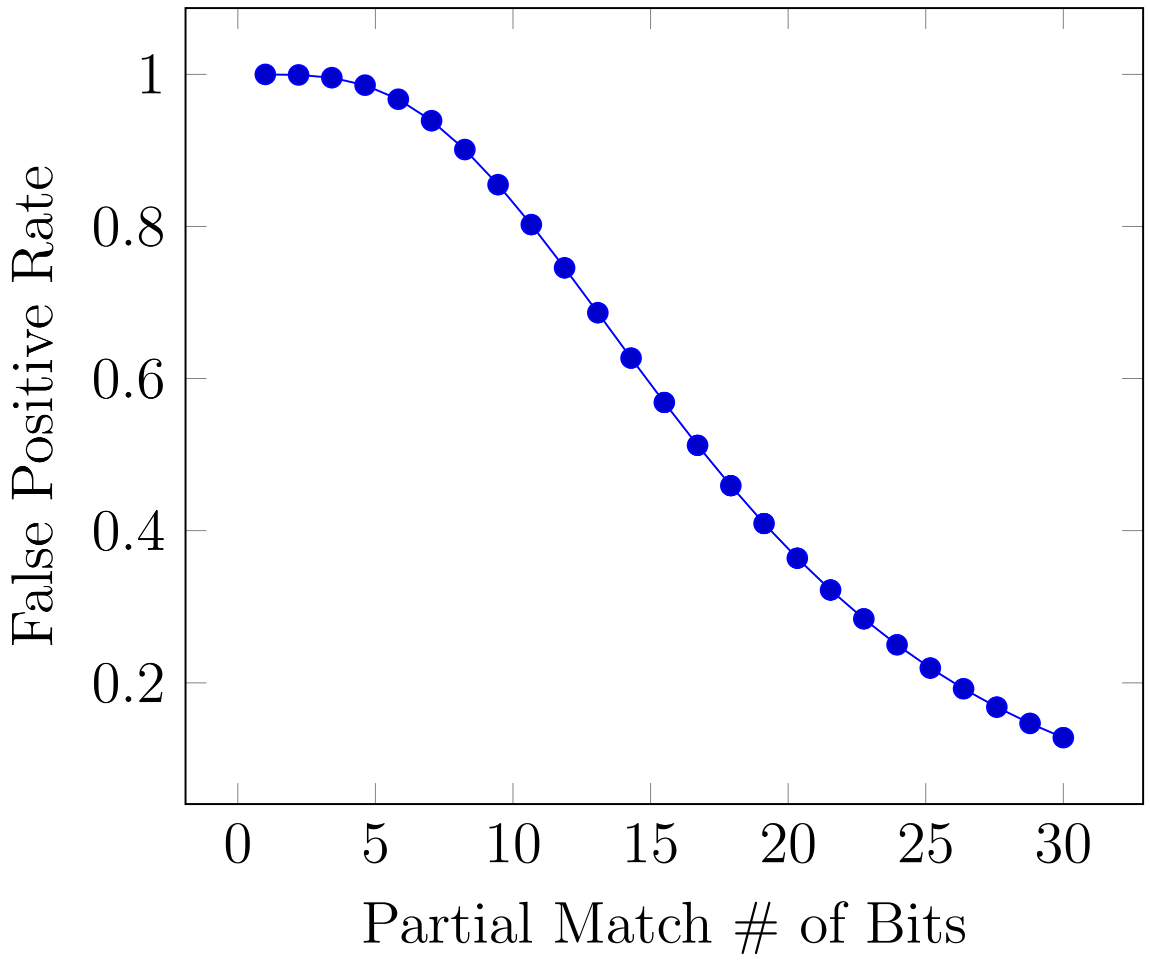


Figure 5.6: Partial Trapdoor Knowledge - False-Positive Rates

5.9 Updates – Forward Privacy

Updates for SSE are challenging because the index structures are encrypted. Also recall that the client is solely responsible for creating and updating indexes. An update that modifies part of the index will leak. This update leakage is why new trapdoors and batch updating is a common technique for forward secrecy with SSE constructions.

When a batch update occurs, an additional index is created with a new key. Searching provides trapdoors for the search term for all of the indexes. Periodically the indexes are combined into one new index.

Our construction can provide forward security using these same techniques. We do not require that updates are performed in batches. We simply need to start using a different key for inserting new records to achieve forward privacy.

5.10 Range Queries

There are many solutions created by the searchable encryption community that take an equality-searchable encryption solution and use it to create a searchable encryption construction with range queries [22]. These techniques convert a range query into a multi-keyword equality query.

Most of these techniques will work with our construction. We implemented Constant-BRC [22] and ran performance tests on these techniques with our system.

5.11 Performance

We implemented a client in the Haskell programming language. To evaluate the performance of our prototype on realistic data and queries at a variety of scales, we used the SPARTA [61] framework from MIT-LL.

The SPARTA test framework includes a data generator and a query generator. The data generator builds artificial data sets with realistic statistics based on real data from the US Census and Project Gutenberg. The query generator creates queries for this test database based on the desired query types and number of return results.

5.11.1 Custom Index

The standard indexes offered by Postgres were not suitable to our needs in terms of performance and index size. Thus we created a custom Postgres index. Our index uses the GiST framework [37]. The GiST framework utilizes a tree structure to store its data types. We utilized the Postgres bit-string data type to store our indexes. Our custom index is a GiST index for bit strings.

GiST is a height balanced tree where the leave contain pairs (key, recordID). Internal nodes contain pairs (p, ptr), where p is a predicate (used as a search key) that is executed for all descendant nodes, and ptr is a pointer to another node in the tree. The number of pairs in each node of the tree is determined by the size of a Postgres page (8K default) and the size of the pairs. This 8K default page size created problems for larger Bloom filter indexes. Exceeding this size caused the index creation to fail. Testing was performed with 32K page sizes.

The GiST framework requires implementation of the following methods:

- Consistent(E, q): given an entry $E = (p, ptr)$, and a query predicate q, returns false if $p \wedge q$ can be guaranteed unsatisfiable, and true otherwise.
    - EDDiES's Consistent function implements a custom query type for Postgres that checks for all the appropriate conjunctions and disjunctions of

bits set in a Bloom filter index.

- Union(P): given a set P of entries $(p_1, ptr_1), ...(p_n, ptr_n)$, returns some predicate r that holds for all tuples stored below ptr_1 through ptr_n.
  - EDDIES's Union is a simple bitwise OR of all the entries $p_1, ..., p_n$.
- Compress(E): given an entry $E = (p, ptr)$, returns an entry (pp, ptr) where pp is a compressed representation of p.
  - EDDIES's Compress is the shortest of the following:
    - Uncompressed Bloom filter
    - List of integers that represent all of the addresses set to 1
    - List of integers that represent all of the addresses set to 0
    - Runlength encoding of the Bloom filter
- Decompress(E): given a compressed representation $E = (pp, ptr)$, returns an entry (r, ptr) where r is a decompressed representation of pp.
- Penalty(E_1, E_2): given two entries $E_1 = (p_1, ptr_1), E_2 = (p_2, ptr_2)$, returns a domain specific penalty for inserting E_2 into the subtree rooted at E_1, which is used to aid the splitting process of the insertion operation.
  - EDDIES's Penalty calculates the number of Bloom filter addresses that differ in p_1, p_2.
- PickSplit(P): given a set P of $M + 1$ entries (p, ptr), splits P into two sets of entries $P1$ and $P2$, each of size at least kM, where k is the minimum fill factor.
  - EDDIES's PickSplit finds the two entries with the highest penalty. Then it assigns the rest of the entries to one of these two according to their penalty scores.

5.11.2 Experimental Setup

We used the database generator to generate databases with 1 million records. We generated over 1,000 queries for each database, consisting of a mix of equality and range queries that returned result sizes between 1 and 10,000 records.

We encrypted the columns `fname`, `lname`, `ssn`, `city`, and `zip`. The rest of the SPARTA columns were inserted into the test database in plaintext.

We performed the tests with the client and the database server located on the same local network via a 1 Gbps Ethernet switch. The server has 12 CPU cores (dual Xeon E5645), 64GB of RAM, and an array of 10k RPM hard drives. It runs the Ubuntu Server 18.04 operating system and Postgres 9.6 as the DBMS.

5.11.3 Experimental Results

Ciphertext Expansion. Table 5.3 shows the overall ciphertext expansion, including the ciphertext expansion from the AES encrypted data and the index column. The database ciphertext expansion is directly related to the number and type of columns encrypted and the index parameters.

Bloom Filter Size	DB Size	DB + Indexes Size
Plaintext	1116 MB	1296 MB
6,000	4363 MB	6350 MB
15,000	5210 MB	11 GB
300,000	7813 MB	13 GB

Table 5.3: Ciphertext Expansion

Database Creation.

Index Size	Insertion Time (minutes)
Plaintext	13.25
6,000	242.31
15,000	544.50
300,000	1,089.25
600,000	2,422.50

Table 5.4: Insertion Times

There are a few factors affecting the time in database creation and insertion times. One is the time to create the ciphertexts and Bloom filter indexes. The other is the time on the server end to create the server indexes.

Query Runtime. We measured query runtime as the time it takes to create the query, send it over the network, receive the data, decrypt, and then filter out the false positive results. Figure 5.7 shows the performance of equality queries and Figure 5.8 shows the performance of equality queries run in parallel. Figure 5.9 shows the performance of range queries.

The slowdown from our system compared to plaintexts comes primarily from two sources. One is traversing the index and checking for matches in the complex query. The other is from the false positives. Obviously the more false positive records returned, the slower the query will be due to transmission of the extra data and decrypting and filtering out the false positives by the client. For example, we ran one test with parameters $\mathcal{IS} = 10, \mathcal{OS} = 10, h = 20$ and other with $\mathcal{IS} = 10, \mathcal{OS} = 10, h = 40$. The test with $h = 40$ has a much larger index size, and thus the index operations will take longer. However it has a much smaller false positive rate. With $h = 20$, each query returned about 59,000 false positive records and with $h = 40$, each query returned fewer than 1,000 false positive records. Because of the lower false positive rate, its average time was three times faster than the test with $h = 20$.

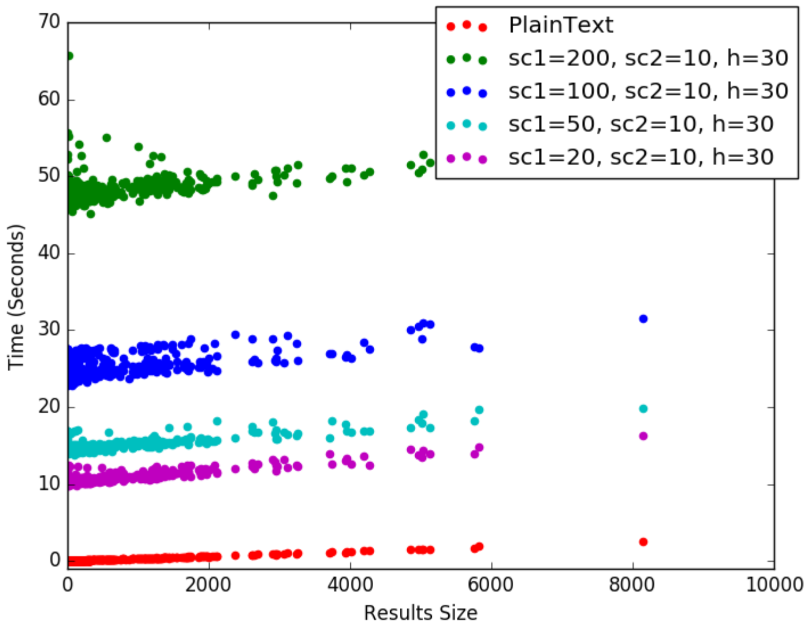


Figure 5.7: Equality Queries

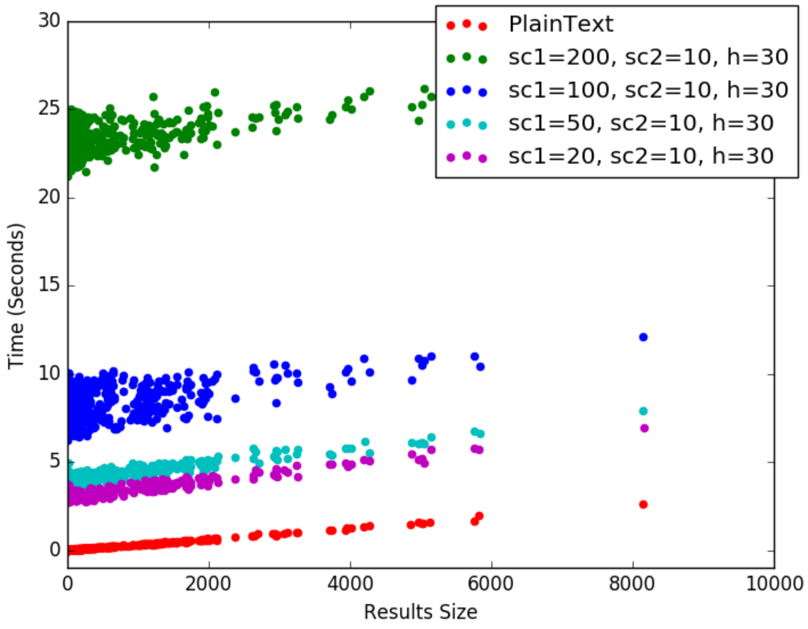


Figure 5.8: Equality Queries - Parallel

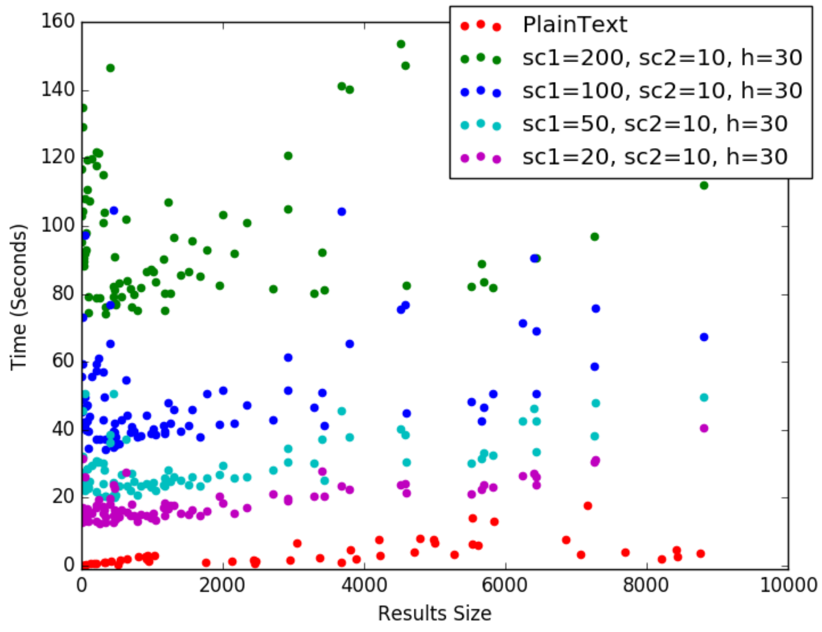


Figure 5.9: Range Queries

The structure of the query lends itself to easily running the query in parallel. Using 10 cores, we averaged a 2-3× speedup on equality queries run in parallel. Part of the reason for the less than linear speedup is our naive implementation of the parallel query traverses many parts of the top of the index tree multiple times.

The range query performance as seen on Figure 5.9 does not follow as tight of a patten as the equality queries. This is because the nature of the range query implementation means the query length and complexity is longer for some queries than others, independent of the size of the return results.

5.12 Review

We showed that it is possible to get the best of both worlds, security similar to SSE, and ease of deployment similar to PPE. Our construction is not as secure as SSE, because we do not meet their adaptive-security definition and our construction leaks more. However, our construction offers another option that provides ease of use with existing DMBS in trade for this leakage while providing superior security to other "easily deployable" constructions.

There are still issues that are not addressed by our construction (or SSE), such as access-pattern leakage attacks [33, 42, 45].

Future work also involves improving the performance of our system. This performance improvement would likely be in the form of a more efficient custom index for Postgres or other DMBS.

6 Conclusions

In this dissertation, we explore the trade-offs necessitated by the leakage inherent with efficient searchable encryption. In particular we reviewed the challenges that come with the constraints from legacy systems and database management systems.

The first major contribution of this dissertation is an attack against property preserving encryption. Prior to this work, property preserving encryption (PPE) had been employed in several systems. It was known that this type of encryption leaked some information. What was not fully known was the consequences of this leakage. This work is not the first attack against PPE, but prior attacks were against low entropy data such as database columns with small domains. This attack showed even against unstructured text with higher entropy, attackers can abuse the leakage from PPE with high success rates.

The second major contribution is a new PPE construction, weakly randomized encryption (WRE). WRE provides superior security compared to other efficient PPE constructions with performance rivaling plaintext. Like all efficient searchable encryption schemes, it has its shortcomings. It is secure only against passive or offline adversaries. It requires knowledge of the plaintext data distributions upfront. It is not clear what the security ramifications are if this distribution later changes.

The third major contribution is another new construction EDDiES. It addresses the weaknesses from WRE. It is secure in the persistent adversarial model. It has no plaintext distribution knowledge requirements. However it suffers a performance drop compared to WRE.

There are many situations with legacy systems where PPE is the only practical option for encryption. With the numerous attacks published in recent years against PPE systems, it appeared that PPE was too vulnerable for deployment. The work in this dissertation reflects that it is possible to create PPE systems more resistant to these attacks and that it is also possible to achieve security against the persistent adversary.

Future work could involve further research into techniques using partial randomization, attempting to improve the performance and security above the constructions presented here. Potential solutions to the deployability and security problems could come from different research areas. For example, it is possible that hardware improvements and distributed databases could enable linear search techniques to perform well enough for practical purposes. The field of encrypted search will likely continue to involve trade-offs with security, performance and utility for a lengthy period of time.

References

[1] H. A. Almohamad and Salih O. Duffuaa. A Linear Programming Approach for the Weighted Graph Matching Problem. *IEEE Trans. Pattern Anal. Mach. Intell.*, 15(5):522525, 1993.

[2] Georgios Amanatidis, Alexandra Boldyreva, and Adam O'Neill. Provably-Secure Schemes for Basic Query Support in Outsourced Databases. In Steve Barker and Gail-Joon Ahn, editors, *DBSec*, volume 4602 of *Lecture Notes in Computer Science*, pages 14–30. Springer, 2007.

[3] David W. Archer, Dan Bogdanov, Liina Kamm, Y. Lindell, Kurt Nielsen, Jakob Illeborg Pagter, Nigel P. Smart, and Rebecca N. Wright. From keys to databases – real-world applications of secure multi-party computation. Cryptology ePrint Archive, Report 2018/450, 2018. https://eprint.iacr.org/2018/450.

[4] Gilad Asharov, Moni Naor, Gil Segev, and Ido Shahaf. Searchable symmetric encryption: Optimal locality in linear space via two-dimensional balanced allocations. In *Proceedings of the Forty-eighth Annual ACM Symposium on Theory of Computing*, STOC '16, pages 1101–1114, New York, NY, USA, 2016. ACM.

[5] Mihir Bellare, Alexandra Boldyreva, and Adam O'Neill. Deterministic and Efficiently Searchable Encryption. In Alfred Menezes, editor, *CRYPTO*, volume 4622 of *Lecture Notes in Computer Science*, pages 535–552. Springer, 2007.

[6] Vincent Bindschaedler, Paul Grubbs, David Cash, Thomas Ristenpart, and Vitaly Shmatikov. The tao of inference in privacy-protected databases. Cryptology ePrint Archive, Report 2017/1078, 2017. https://eprint.iacr.org/2017/1078.

[7] D. Blei, A. Ng, and M. Jordan. Latent Dirichlet Allocation. *Journal of Machine Learning Research*, 3:9931022, January 2003.

[8] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[9] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'Neill. Order-preserving symmetric encryption. pages 224–241, 2009.

[10] Dan Boneh and Victor Shoup. A graduate course in applied cryptography. http://toc.cryptobook.us/. Accessed: 2018-5-1.

[11] Raphael Bost. Forward secure searchable encryption. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1143–1154, New York, NY, USA, 2016. ACM.

[12] Raphaël Bost, Brice Minaud, and Olga Ohrimenko. Forward and backward private searchable encryption from constrained cryptographic primitives. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1465–1482, New York, NY, USA, 2017. ACM.

[13] Andrei Z. Broder and Michael Mitzenmacher. Survey: Network Applications of Bloom Filters: A Survey. *Internet Mathematics*, 1(4):485–509, 2003.

[14] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. pages 668–679, 2015.

[15] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014.

[16] David Cash and Stefano Tessaro. The locality of searchable symmetric encryption. 05 2014.

[17] Yan-Cheng Chang and Michael Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. pages 442–455, 2005.

[18] Melissa Chase and Seny Kamara. Structured Encryption and Controlled Disclosure. In *Advances in Cryptology - ASIACRYPT 2010*, volume 6477 of *Lecture Notes in Computer Science*, page 577594. Springer Verlag, December 2010.

[19] Chandra Chekuri and Sanjeev Khanna. A polynomial time approximation scheme for the multiple knapsack problem. *SIAM Journal on Computing*, 35(3):713–728, 2005.

[20] Donatello Conte, Pasquale Foggia, Carlo Sansone, and Mario Vento. Thirty years of graph matching in pattern recognition. *International journal of pattern recognition and artificial intelligence*, 18(03):265–298, 2004.

[21] Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *ACM Conference on Computer and Communications Security*, pages 79–88. ACM, 2006.

[22] Ioannis Demertzis, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, Minos Garofalakis, and Charalampos Papamanthou. Practical private range search in depth. *ACM Trans. Database Syst.*, 43(1):2:1–2:52, March 2018.

[23] Ioannis Demertzis and Charalampos Papamanthou. Fast searchable encryption with tunable locality. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1053–1067, New York, NY, USA, 2017. ACM.

[24] Mohammad Etemad, Alptekin Kp, Charalampos Papamanthou, and David Evans. Efficient dynamic searchable encryption with forward privacy. *Proceedings on Privacy Enhancing Technologies*, 2018(1), 2018.

[25] Ben A. Fisch, Binh Vo, Fernando Krell, Abishek Kumarasubramanian, Vladimir Kolesnikov, Tal Malkin, and Steven M. Bellovin. Malicious-client security in blind seer: A scalable private DBMS. pages 395–410, 2015.

[26] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. Tworam: Efficient oblivious ram in two rounds with applications to searchable encryption. In *CRYPTO*, 2016.

[27] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–169, 2009.

[28] Javad Ghareh Chamani, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. New constructions for forward and backward private symmetric searchable encryption. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1038–1055, New York, NY, USA, 2018. ACM.

[29] Eu-Jin Goh. Secure Indexes. *IACR Cryptology ePrint Archive*, 2003:216, 2003.

[30] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *JACM*, 43(3):431–473, 1996.

[31] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of computer and system sciences*, 28(2):270–299, 1984.

[32] Paul Grubbs. On deploying property-preserving encryption. Real World Cryptography, 2016.

[33] Paul Grubbs, Marie-Sarah Lacharite, Brice Minaud, and Kenneth G. Paterson. Pump up the volume: Practical database reconstruction from volume leakage on range queries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 315–331, New York, NY, USA, 2018. ACM.

[34] Paul Grubbs, Marie-Sarah Lacharit, Brice Minaud, and Kenneth G. Paterson. Learning to reconstruct: Statistical learning theory and encrypted database attacks. Cryptology ePrint Archive, Report 2019/011, 2019. https://eprint.iacr.org/2019/011.

[35] Peeyush Gupta, Yin Li, Sharad Mehrotra, Nisha Panwar, Shantanu Sharma, and Sumaya Almanee. Obscure: Information-theoretic oblivious and verifiable aggregation queries. *Proc. VLDB Endow.*, 12(9):1030–1043, May 2019.

[36] Warren He, Devdatta Akhawe, Sumeet Jain, Elaine Shi, and Dawn Xiaodong Song. ShadowCrypt: Encrypted Web Applications for Everyone. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM Conference on Computer and Communications Security*, pages 1028–1039. ACM, 2014.

[37] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized search trees for database systems. In *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, pages 562–573, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[38] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*, volume 20, page 12, 2012.

[39] Seny Kamara and Charalampos Papamanthou. Parallel and dynamic searchable symmetric encryption. In *Financial Cryptography and Data Security - 17th International Conference, FC 2013, Okinawa, Japan, April 1-5, 2013, Revised Selected Papers*, pages 258–274, 2013.

[40] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic searchable symmetric encryption. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 965–976, New York, NY, USA, 2012. ACM.

[41] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. Chapman & Hall/CRC, 2nd edition, 2014.

[42] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. Generic attacks on secure outsourced databases. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM Conference on Computer and Communications Security*, pages 1329–1340. ACM, 2016.

[43] Kee Sung Kim, Minkyu Kim, Dongsoo Lee, Je Hong Park, and Woo-Hwan Kim. Forward secure dynamic searchable symmetric encryption with efficient updates. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1449–1463, New York, NY, USA, 2017. ACM.

[44] Bryan Klimt and Yiming Yang. The enron corpus: A new dataset for email classification research. In *Machine learning: ECML 2004*, pages 217–226. Springer, 2004.

[45] Evgenios M. Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. The state of the uniform: Attacks on encrypted databases beyond the uniform query distribution. *IACR Cryptology ePrint Archive*, 2019:441, 2019.

[46] Marie-Sarah Lacharité and Kenneth G. Paterson. A note on the optimality of frequency analysis vs. ℓ_p-optimization. Cryptology ePrint Archive, Report 2015/1158, 2015.

[47] M. Lacharit, B. Minaud, and K. G. Paterson. Improved reconstruction attacks on encrypted data using range query leakage. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 297–314, May 2018.

[48] Billy Lau, Simon Chung, Chengyu Song, Yeongjin Jang, Wenke Lee, and Alexandra Boldyreva. Mimesis aegis: A mimicry privacy shield–a system's approach to data privacy on public cloud. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 33–48, San Diego, CA, August 2014. USENIX Association.

[49] Muhammad Naveed, Seny Kamara, and Charles V. Wright. Inference Attacks on Property-Preserving Encrypted Databases. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM Conference on Computer and Communications Security*, pages 644–655. ACM, 2015.

[50] Vasilis Pappas, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Angelos D. Keromytis, and Steve Bellovin. Blind Seer: A Scalable Private DBMS. In *IEEE Symposium on Security and Privacy*, pages 359–374. IEEE Computer Society, 2014.

[51] Vasilis Pappas, Mariana Raykova, Binh Vo, Steven M. Bellovin, and Tal Malkin. Private search in the real world. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 83–92, New York, NY, USA, 2011. ACM.

[52] Kenneth Paterson and Marie-Sarah Lacharité. Personal communication, December 2017.

[53] Raluca A. Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: protecting confidentiality with encrypted query processing. In Ted Wobber and Peter Druschel, editors, *SOSP*, pages 85–100. ACM, 2011.

[54] Mariana Raykova, Binh Vo, Steven M. Bellovin, and Tal Malkin. Secure anonymous database search. In Radu Sion and Dawn Song, editors, *CCSW*, pages 115–126. ACM, 2009.

[55] Tahmineh Sanamrad, Lucas Braun, Donald Kossmann, and Ramarathnam Venkatesan. Randomly Partitioned Encryption for Cloud Databases. In Vijay Atluri and Günther Pernul, editors, *DBSec*, volume 8566 of *Lecture Notes in Computer Science*, pages 307–323. Springer, 2014.

[56] D. Song, D. Wagner, and A. Perrig. Practical Techniques for Searching on Encrypted Data. In *S&P*, pages 44–55, 2000.

[57] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. Practical dynamic searchable encryption with small leakage. 01 2014.

[58] Shi-Feng Sun, Xingliang Yuan, Joseph K. Liu, Ron Steinfeld, Amin Sakzad, Viet Vo, and Surya Nepal. Practical backward-secure searchable encryption from symmetric puncturable encryption. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 763–780, New York, NY, USA, 2018. ACM.

[59] Shinji Umeyama. An Eigendecomposition Approach to Weighted Graph Matching Problems. *IEEE Trans. Pattern Anal. Mach. Intell.*, 10(5):695–703, 1988.

[60] David C Uthus and David W Aha. The ubuntu chat corpus for multiparticipant chat analysis. In *AAAI Spring Symposium: Analyzing Microtext*, 2013.

[61] Mayank Varia, Benjamin Price, Nicholas Hwang, Ariel Hamlin, Jonathan Herzog, Jill Poland, Michael Reschly, Sophia Yakoubov, and Robert K. Cunningham. Automated Assessment of Secure Search Systems. *SIGOPS Oper. Syst. Rev.*, 49(1):22–30, January 2015.

[62] Jiafan Wang and Sherman S. M. Chow. Forward and backward-secure range-searchable symmetric encryption. Cryptology ePrint Archive, Report 2019/497, 2019. https://eprint.iacr.org/2019/497.

[63] Andrew C. Yao. Protocols for secure computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, SFCS '82, pages 160–164, Washington, DC, USA, 1982. IEEE Computer Society.

[64] Mikhail Zaslavskiy, Francis Bach, and Jean-Philippe Vert. A path following algorithm for the graph matching problem. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 31(12):2227–2242, 2009.

[65] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 707–720, 2016.