Searching Encrypted Data

Project Report

William Harrower
⟨william.harrower05@imperial.ac.uk⟩
Department of Computing
Imperial College London

Supervisor: Dr. Naranker Dulay ⟨nd@doc.ic.ac.uk⟩
Second Marker: Dr. Herbert Wiklicky ⟨herbert@doc.ic.ac.uk⟩

June 15, 2009

Abstract

Company data are very often outsourced to datacentres in order to lower costs of maintaining hardware. If the outsourced data are to be kept secure from a third party, the connection between the datacentre and the company could be secured by a protocol similar to SSL. This, however, requires that the data is stored at the datacentre in *plaintext* form, meaning the company has to *trust* the datacentre and its administrators.

Alternatively, the data themselves could be encrypted, however, the outputs of typical cryptographic algorithms are not amenable to *search*. This project explores the research area of searching over *encrypted* data, specifically using a secure pre-processed index approach. The output of the project is a system developed for the PostgreSQL 8.3 database management system that allows data to be encrypted and stored in a database table whilst allowing for secure, server-side searches with minimal performance overhead, both in space and time.

The datarate achieved by the encryption algorithm is around 2.21MB/s, which is shown to be *faster* than an existing *plaintext* PostgreSQL indexing algorithm. The search algorithm is shown to be around twice as fast as plaintext *linear* search, with a rate of 300MB/s, but slower than indexed plaintext search. These results demonstrate the system's practicality within an industrial setting.

Acknowledgements

I would first like to thank my supervisor, Dr. Naranker Dulay for all his support throughout the course of the project, as well as Changyu Dong for his input and help during meetings. I would also like to thank my second marker and personal tutor, Dr. Herbert Wiklicky for his early input on the report and providing help throughout the year.

I would also like to thank my parents, who have always been prepared to go out of their way to give their utmost support during my time at university.

Finally, I would like to thank my friends, in particular Will Deacon, Robin Doherty, Andrew Jones and Will Jones, for their friendship and their help during my time knowing them.

Contents

1 Introduction

"Gentlemen do not read each others' mail"
– Henry Lewis Stimson

1.1 The Problem

Cryptography is the practice of transforming data to make it indecipherable by a third party, unless a particular piece of secret information is made available to them. Many different forms of cryptographic algorithms exist, each designed for a different purpose. The most obvious of these is what we would intuitively think of as "encryption" – private-key algorithms that use the same *key* to secure the data as they do to restore the original. This sort of encryption allows users to hide their secrets and access them at a later date. An alternative is public-key cryptography, which is typically used to send messages to *other* people. A way of explaining the differences between private- and public-key systems is by thinking about it as sending the key in the former, but sending the *padlock* in the latter.

If I want people to be able to send me secure mail (of the snail variety, via the Post Office) that only I can read, one way of achieving this is to have a large number of padlocks made which all use the same key. I can then place this box of unlocked padlocks outside my front door (without their keys), allowing anyone who wants to send me a message to take one. They can then write me a letter, place it in a box and lock it with my padlock. Only I have the key that can unlock this, so now it is safe for them to leave the box on my front door-step. Of course in this analogy, in order for the system to be secure, we rely on the fact that no-body can take the padlock and deduce the shape of the key. Bruce Schneier describes this notion of security nicely in his book "Applied Cryptography":

"If I take a letter and lock it in a safe, and then give you the safe along with the design specifications of the safe and a hundred identical safes with their combinations so that you and the world's best safecrackers can study the locking mechanism – and you still can't open the safe and read the letter – that's security." [18]

This concept, known as Kerckhoffs' principle, is at the heart of cryptography: the security of a system should rely on the secrecy of a key, not the secrecy of the underlying algorithm. This idea has served the cryptographic community well, with algorithms being published and then broken, allowing development of stronger algorithms and techniques that aren't vulnerable to the attack. This cycle has helped the advancement of cryptographic algorithms from early substitution ciphers, right up to today's algorithms.

Regardless of whether you choose to encrypt your data with a public or private-key algorithm, you will inevitably end up with *ciphertext*. This is the encrypted form of the *plaintext* (the original data prior to transformation). As I will describe in Section 4.1, most encryption algorithms traverse the input data in discrete blocks, producing output that combines some subset of the key data with the input block. These blocks will be output sequentially, but, assuming the input data is some form of written text, it is very unlikely that the block boundaries are in-sync with the 'word' boundaries of the original input data.

This is exemplified by Figure 1.1, in which a plaintext is encrypted using a block cipher in ECB mode (explained in Section 4.1.1) that works on 4-byte blocks as input (assuming the input is UTF-8). This means that some encrypted blocks (the lower set of hexadecimal numbers in the diagram) will contain parts of multiple words and will not necessarily start at the beginning of a word.

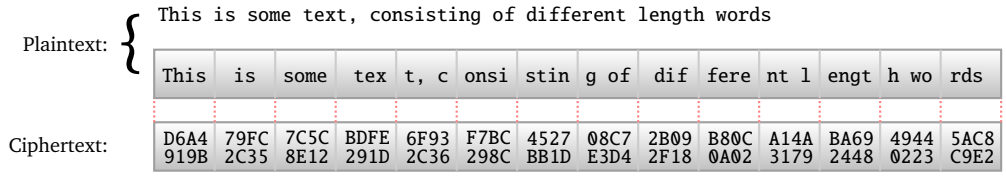

Figure 1.1: A simplified example of encryption using a block cipher with a 4 byte block size.

Using this example, how would we search the encrypted information for a given term? We might get lucky if we were to pick a word like 'some', which happens to fall within a single block. If we did pick 'some', we could encrypt it with the same key we used to encrypt the full document and then search for the result in the ciphertext. However, if we picked the word 'text' we would be faced with a problem. As 'text' falls across a 4-byte boundary in the plaintext, it has been encrypted into two ciphertext blocks. Block ciphers rely on the entire block in order to produce their output – if you change the last bit in the 4-bytes, for example, the entire output will be effected (not just the last bit(s) of the ciphertext) – and so there's no way we can use the simple pattern match approach as we used with 'some'.

This is a simplified example – things get even worse if we use the more secure (and hence popular) CBC mode of the cipher. This mode uses the ciphertext generated from the previous block in order to encrypt the current one, meaning a modification of the first bit of the plaintext will result in *all* ciphertext blocks being altered (a sort of ripple-chaining effect). How can we search in this mode? We would effectively have to know all words that precede the one we're searching for, including their exact order and any punctuation in order to regenerate the ciphertext, therefore rendering the search pointless to begin with.

I will further explain the individual components involved and terminology used in the above example in the coming sections, but for now it should be enough to demonstrate the shortcomings of existing encryption schemes when it comes to searching. That is to say that data is encrypted in such a way as to blur the lines between individual input words. A naïve solution to this problem would be to encrypt each input word individually and store the encrypted outputs in the same order as the inputs (using variable-length block sizes), perhaps with extra bytes added to the front of each to allow for words that produce multiple blocks of output. However this solution leaks information (if the same word appears twice in the plaintext, the corresponding ciphertext blocks will be identical – a trait necessary for searching) which could be used for statistical attacks.

1.2 Motivation for a Solution

As data sizes grow, so does the need for efficient search. Storing a back-catalogue of 5,000 emails serves no purpose, aside from filling up your hard disk, unless they can be easily searched. This becomes exponentially more apparent when dealing with company-wide email systems and other large scale data collections. These systems are also inherently distributed in nature, either internally to the company (perhaps in the main office buildings), or more likely, outsourced (maybe to a local datacentre, or even abroad). If we want data to remain secure, especially if outsourced to a third party, we need to use encryption.

Once the data is encrypted and outsourced to a far-off datacentre we need to be able to access it. If we need to perform a search over the encrypted data, we have two options:

- We can download the *entire* data set, decrypt it and search it client-side.
- We can give the remote server the secret key we used to encrypt the data and let the server decrypt and search it.

In order to discuss these options, let's assume a fairly plausible example of a small business that have outsourced an email archive to a third party datacentre. The email collection is a complete record of all emails sent and received by everyone in the company since it was founded 5 years ago. The data reaches roughly 500GB in size and has been encrypted using a standard, secure block cipher before being first uploaded to the datacentre.

In this scenario, both of the 'solutions' described above have fairly obvious ramifications. If the company decide to go with the first choice and download the data before decrypting and searching it client-side, they are going to experience an impossibly large amount of communication overhead. Downloading 500GB of data every time you want to search for all emails containing the word 'urgent' is quite clearly unacceptable.

If the company opt to hand their secret key to the remote server and allow it to decrypt and search the emails, the company are required to *trust* the datacentre. Knowing the key, a rogue datacentre admin could perform a number of harmful acts, ranging from simply decrypting the emails and learning about their contents, to modifying or deleting them. The encryption used is rendered useless once the key is made available and merely acts to add overhead to the search and retrieval process. It would be far more efficient to encrypt the *connection* between the datacentre and the company if the latter is willing to fully trust the former.

Outsourcing of data, although one of the most obvious, isn't the only application area. Frequently, when given a set of data about an entity X, the most efficient place to store the data is actually on X itself. Customer reviews of a high-street shop for example would be best located in or around the shop. This would allow potential customers to read the reviews before they commit to shopping there. In this scenario, if the reviews are stored as plaintext, the shop could easily remove any bad reviews, or modify them in the shop's favour. If they're encrypted then the only option open to the shop is to remove the *entire* set of reviews or leave them as-is. A slight variation of this theme might be that the reviews, rather than being public reviews, are instead health and safety reports made by government officials. These reports should be concealed from the shop. In both these examples, searching the reviews/reports might well be a desired feature, whilst retaining the necessary levels of security.

1.3 Contributions

The main contribution of this project is the implementation of a searchable encryption scheme for textual data, in the form of a portable 'common' library, as well as a custom datatype and operator for the PostgreSQL 8.3 database management server that make use of this library (the design and implementation of which are detailed in Chapter 5). This allows a database server to be hosted on an *untrusted* server, whilst maintaining the ability to perform server-side, secure searches over encrypted data without leaking information (such as the search term, or which documents were determined to definitely match the given query) to the host server. The implemented system is based on Eu–Jin Goh's secure index scheme [9] (detailed, along with other research in the area, in Chapter 4), with a number modifications to the way index sizes are calculated in order to improve performance.

A client application for interaction with the database server was also developed and is used to explore the different possible search types (described in detail in Chapter 2), as well as extracting test data in order to perform a thorough performance evaluation (see Chapter 6).

The developed system allows for a number of different search variants: exact match, natural language search, case insensitive and left-most sub-matches, with this being the first time left-match support has been investigated. Through benchmarking, we have found the implementation to deliver acceptable performance, far exceeding what is possible using 'standard' encryption technologies (a standard block cipher, for example), which would require either the full trust of the host server, or that the entire document set is downloaded to the client prior to decryption and search.

The performance of the system is also found to be comparable to *unencrypted* data searches, meaning the addition of strong levels of security is not overly expensive (see Chapter 6 for a full evaluation). The encryption algorithm delivers a datarate of around 2.21MB/s (faster than an existing PostgreSQL plaintext indexing algorithm) and the search algorithm is capable of traversing data at a rate of 300MB/s (twice as fast as plaintext linear search and not unbearably slower than plaintext indexed search).

2 Search Considerations

When given a large amount of data, there are a number of different ways that you could "search" through it. Some of these are supported by different encryption schemes (discussed in Section 4.2) and some, due to the constraints that the schemes have, are not. These are described here for clarity, before I go on to look at different areas of application and the proposed encrypted search schemes. The data in question here is solely textual, with some of the search techniques being language-specific. When this is the case, it will be noted explicitly.

2.1 Inner Document Search

The process of "searching" through a single document can utilise a number of different techniques in order to determine whether the document matches a given query. This statement also assumes that the granularity of search results is a single document (whether it 'contains' the search query or not), but this could be extended to the line, or the sentence that the query matches against. The following describe criteria under which a query can be matched.

Exact-match

The most obvious process that falls under the umbrella term of "searching" is an exact match. This involves searching a document set for "X" and receiving the documents that contain at least one *exact* instance of X. This also assumes the documents' contents are delimited by all forms of punctuation. This is potentially language-specific, depending on how punctuation is used in different languages. For example, a document containing the sentence "*Hello, my name is William*" should be found as a match for the query "*Hello*", using an exact-match search. This means that words within the document must be delimited by punctuation. Unfortunately, there are occasions when using punctuation as a delimiter is not desired (some hyphens within words, for example).

Word Sub-match

This type of search allows a user to search for a substring within a document. For example, a document containing the word "successfully" should be returned when the user searches for "success". There are also subclasses of a sub-match, such as left-most match (which will only attempt to match the query against the left part of words within the document) and complete substring match (the query could be found at any index within another word). For example, a document containing "*It was successful*" would be returned as a match when the search engine is presented with the query "*ccess*" only if the sub-match technique in use was a full substring search. This document would not be returned for the same query if a left-most match system was in use.

Case Insensitivity

The comparison of a search query's contents with a word in a document can be made either case insensitive or sensitive. Case sensitivity decisions have to be made in tandem with a search technique, such as exact-match. For example, searching a document for X will result in all documents containing at least one instance of X, or any case variation of X when using exact-match. Although this is fairly simple when dealing with ASCII characters, case sensitivity can be more complex and sometimes not supported by the host environment when using extended character sets like Unicode (the case insensitive equality of æ and Æ, for example).

Regular Expressions

Regular expressions are a very powerful way of describing a *pattern* that you want to search for. The following regular expression, for example, describes any alphanumeric sentence beginning with the word "Hello": `^Hello[\wa-zA-Z0-9]*$`. Regular expressions are effectively a language that allow the searcher to represent their own search scheme (they can be used to represent all of the above, for example). They are typically implemented as a deterministic finite state machine, with each input symbol from the document to be searched 'fed' in, one at a time. If the state machine reaches a goal state, the regular expression is satisfied and the document is deemed a match.

Proximity Based Queries

Proximity queries allow you to return any documents that contain a word X that is *close* to word Y, or perhaps more specifically, within the same sentence. A variation of this is word ordering: X appears before Y, for example.

Natural Language Search

This doesn't refer to fully-fledged natural language search, as this has not yet been successfully (fully) implemented, but an imitation. One of the key principles behind allowing searches of this nature is word *stemming*. This is the process of taking a word and transforming it into the base word (lexeme) from which it is derived. For example, the word "searching" is derived from the word "search" and so using natural language search, we could search for documents containing words that relate to "search" (such as "searcher" and "searchable", or "search" itself).

Another process commonly associated with natural language search is the removal of 'stop words'. These are words that convey little meaning other than to form a complete sentence, such as "the," "and," and "a." These are typically removed prior to a search, so if a user searches for "how do I boil a kettle", the search algorithm will look for documents containing "boil" and "kettle" (or their derivatives).

2.2 Document Preparation and Search Strategies

As well as the different *types* of search, we must also consider the different possibilities for *how* the search will be performed on a given set of documents.

Linear Search

Each document is traversed linearly, from start to finish, in order to match against a given search query. This process can be very slow when used on large document sets, as well as computationally expensive. An advantage of this technique, compared to a pre-processed index is that there is no initial preparation time for a document (when it is initially stored, for example). Also, with the granularity of search results assumed to be full documents, the search can be performed *lazily*, meaning once a search term is matched the search can be terminated and the document returned as a match. This means that searching for common terms will potentially be very fast, as they are likely to be matched early. Determining that a document does *not* match against a query, however, requires a complete traversal of the document.

Pre-processed Index

Upon storage of a document, an index is created that contains an entry for each *unique* word in the document. When a search is performed, the index is consulted and the document is added to the result set, only if the word is contained in the index. The process of determining whether or not the index 'contains' the given search term is typically based on a *hash* of the term, used to access a data structure such as a hash map.

For very large documents, this can greatly reduce search time. It does, however, obviously increase the size on disk required to store each document, as they will need to be stored side-by-side with their index. Also, initial processing time is added by the index creation algorithm. This technique is more suited to applications where the frequency of queries exceeds that of updates.

3 Applications

In this chapter I shall review some potential areas of application. I will give an overview of the area, before looking at one or more specific examples.

3.1 Mail Servers

Mail servers come in a number of different forms, support different mail protocols and serve different purposes. There is a clear distinction between servers that use the POP and IMAP protocols for example. Post Office Protocol (POP) [63] servers store emails until a client program connects to them and downloads them, after which the emails are deleted from the server. This setup can cause problems if the user has multiple physical machines that she uses, perhaps one at work and one at home. Using a POP server will mean that emails are download to only one, unless some additional manual synchronisation is performed. Internet Message Access Protocol (IMAP) [61] servers keep all emails they receive (unless manually deleted by the user) and merely mark them as read/unread as necessary. This allows for multiple machines to access the same IMAP account on a given server and stay in-sync with each other. IMAP is clearly more suited to the application of remote search, as it maintains an archive of past emails.

Mail servers, by their nature, require a form of public-key encryption rather than private-key. There's no point sending emails that have been encrypted with a password that only the sender knows. I will discuss basic public-key cryptography in Section 4.1.4 and look at the search-specific efforts made in the public-key domain in Section 4.2.4.

University of Washington IMAP Toolkit

The toolkit created by the University of Washington is a collection of components designed to support IMAP. It is comprised [59] of a client library (`c-client`) which provides an API to programs wanting to act as email clients or servers, a pre-built IMAP server (`imapd`), as well as a collection of other utilities. It is written in C and is licensed under the version 2.0 of the Apache Licence.

A number of well known client applications have been built using the `c-client` library, such as Pine and Alpine. Since these are fairly well adopted programs, with both native and web front-ends, extending the underlying `c-client` API to support encrypted search queries would make it easy to extend the individual applications.

3.2 Database Management Systems

Databases are obvious candidates for executing encrypted search queries over. They can be hosted remotely or locally and usually maintain a large data collection. A wide variety of Database Management Systems (DBMSs) have existed over the years, with the current open source landscape being punctuated by three major competitors. These are PostgreSQL[40], MySQL[35] and SQLite[51]. In the following

sections, I will discuss the main differences between these systems, relevant to the implementation of an encrypted search scheme.

Although the different systems handle specifics slightly differently, the broad implementation of an encrypted search scheme into any of them would probably be best done through the introduction of a new SQL data type and possibly operator. An `ENCVARCHAR` type, for example, could be created and applied to columns. This data type would then represent the encrypted data. An operator, similar to `LIKE` could also be added, rather than modifications being made to the existing operators to support the new data type. This might be necessary for passing key information. For example, the following (extended) SQL could be used to check for 'searchterm' in the specified encrypted column:

```
SELECT * FROM mytable
WHERE myencryptedcolumn
ENCCONTAINS `searchterm´
WITHKEY `mypassword´
```

Obviously, for security reasons that will be more carefully explained in the background section for each individual search scheme, this statement would need to be transformed by the client before transmission to avoid handing the key over the network in plaintext. Other options could be explored, such as client flags that can be set in order to cache a global key, rather than having to include it in all encrypted searches. From this, it can be clearly seen that modifications to both server and client code will be necessary for any search scheme.

PostgreSQL

PostgreSQL is a fully featured object-relational database management system. It is split into a number of different entities, providing different functionality – the two of interest are the server (`postgres`) and the client terminal application (`psql`). These, respectively, host a PostgreSQL database and allow a user to connect to the server and enter SQL queries at a terminal.

A number of different options are available to the user, via the `psql` client terminal, when they want to search a database:

- The SQL `LIKE` condition: `SELECT * FROM mytable WHERE mycolumn LIKE `value%´`
- The case-insensitive counterpart of like, `ILIKE`
- Regular expressions are supported through the `SIMILAR TO` condition, the `~` operator and its derivatives (`~*`, `!~` and `!~*`).
- PostgreSQL also supports Full Text Search (FTS). This *"provides the capability to identify natural-language documents that satisfy a query, and optionally to sort them by relevance to the query"* [42]. FTS allows queries such as
  `SELECT `value1 value3 value4´::tsvector @@ `value1 & value2´::tsquery;`
  The above query will return a single row containing `false`, as 'value1' and 'value2' do not both appear in the document text (the `tsvector`). FTS can also be used with pre-processed indexes.
- An extension to PostgreSQL's FTS has also been developed. OpenFTS [36] provides features such as *"online indexing of data, proximity based relevance ranking, multilingual support and stemming"* [37]. It is written in Perl and TCL and licensed under the GNU GPLv2.

PostgreSQL is written in C and is released under the highly liberal BSD licence[43].

MySQL

MySQL is an extremely popular relational database management system used by a host of well known companies. The server (`mysqld`) daemon is normally run on dedicated server and the client terminal application (`mysql`) allows a user to connect to the server and issue it SQL commands.

Although the `LIKE` conditional is supported, MySQL handles string searches in a different way to PostgreSQL:

- Searches on strings (`CHAR`, `VARCHAR`, `TEXT`, etc) depend on the *collation* of the columns involved in the search. The collation set on a column defines how it is to be ordered (such as alphabetically) and the default collation is `latin1_swedish_ci`. The `_ci` suffix of this collation states that ordering is *case insensitive*. The result of this is that, by default, all string searches are case insensitive [34].
- The `ILIKE` conditional is not supported (as the column collation is always consulted before comparison).
- Regular expressions are supported through the `REGEXP` operator. For example,
  `SELECT ` document contents´ REGEXP ``^[a-z]´`
- MySQL also features Full-Text Search through its `MATCH()` function. Natural language searches are supported by the `IN NATURAL LANGUAGE MODE` modifier:

  ```
  SELECT * FROM mytable
  WHERE MATCH (id, contents)
  AGAINST ('searchterm' IN NATURAL LANGUAGE MODE);
  ```

  This will return the `id` and `contents` values for any row in `mytable` containing the word "searchterm". These results will be ordered by relevance. If the search term consists of multiple words, not all of these words have to appear in a row in order for it to be deemed relevant (although, obviously, the more that are found, the more relevant it will be deemed). Word stemming will also be used to search for terms that are derived from the specific query given ("searchterm").

MySQL is written in C and C++ and is licensed under both the GNU GPLv2 and a proprietary licence.

SQLite

SQLite is a reasonably small library that *"implements a self-contained, serverless, zero-configuration, transactional SQL database engine"* [51]. It is different to both MySQL and PostgreSQL (and most other database management systems) because it is typically used as an *embedded* database. Rather than being run as a separate process, it is built as a library. An application would generally be deployed with a copy of this library which it would dynamically link to at runtime. This application could then use the SQLite API to access a local database which is stored as a *single file*.

Because it is designed to be embedded in this manner, the current number of SQLite deployments is extremely large, making it, by some estimates, the most widely deployed SQL database [52].

SQLite's support for searching is similar to MySQL and PostgreSQL:

- Supports the `LIKE` conditional, which is always case-insensitive [53].
- Regular expressions are supported by the `REGEXP` operator.
- Full-Text Search is supported through a separate module, **fts1** [54]. A full-text table consisting of columns of fully indexed text can be created using the `CREATE VIRTUAL TABLE` command [54].

SQLite is written in C and is released in the public domain [55].

3.3 File Systems

File systems are another obvious candidate for both encryption and searching. They meet the requirements of generally being quite large in size and often contain data amenable to search. However, they are typically local to the application that wants to search, rather than remote. This removes the bottleneck that is the network connection and makes the need for a specialist encrypted search scheme less important. There are, however, a number of file systems that are designed to run remotely:

- **FUSE/SSHFS**

  Filesystem in Userspace (FUSE) [28] is a Linux kernel module which allows simple development of virtual file systems without modifying kernel code. A variety of different file systems have been created using FUSE, one of which is SSHFS [56]. This allows you to locally mount and access (read from/write to) a remote directory. The network communication and authentication is handled by the Secure Shell (SSH) protocol and the filesystem is maintained by FUSE. As the filesystem will appear locally as a native directory hierarchy, standard tools such as `grep` and `find` can be used to search the remote data.

  One potential option for introducing an encrypted search scheme is to build it into the SSHFS client so the decryption is transparent to the user.

- **Samba**

  Samba [49] is an open source implementation of a number of protocols, including the Server Message Block (SMB) protocol, also known as the Common Internet File System (CIFS) protocol. This protocol provides network access to file and printer shares and could be extended to support secure search queries, with results being transparently decrypted locally using a cached copy of the client's private key.

3.4 File Indexers

File indexers have gained popularity recently, with hard disk drives and their respective, cluttered contents growing in size at an alarming rate. They attempt to solve the needle in a haystack problem without having to actively trawl through the entire contents of the hard disk for each search. They typically work by creating an index in the background, whilst you perform other tasks, or your computer idles. This index allows very fast retrieval of a document list for a given search term through techniques such as hash maps and Bloom filters (as I will describe later, in Section 4.2.2). A number of proprietary indexers exist, targeting different systems, such as Google Desktop [31] and Spotlight [50]. Open source efforts are also available, with Beagle [24] being an option available to Unix-like operating systems. It is capable of indexing a variety of content, from normal files, to emails and tagged music files.

Indexers such as these are, however, inherently *local* to the data, meaning the network communication overhead benefits available to us thanks to these schemes are lost. A system would also have to be in place to encrypt the data in the first place, meaning either a secure area within a hard drive which an Indexer is then extended to support, or a modified filetype of some form.

3.5 LDAP

The *Lightweight Directory Access Protocol* (LDAP) is a *"standard that computers and networked devices can use to access common information over a network"* [6]. It is typically used for applications that are focused on updates and requests for data, with one primary example being to handle employee contact details within a number of large corporations. In this scenario, the protocol allows look-ups to be performed to search for people by name, department, phone number, etc.

A system that supports LDAP will consist of at least two applications: a server and a client. The server can represent the stored data that is accessed by the LDAP capable client in a number of ways, including using a database back-end, or through the LDIF format [10]. Figure 3.1 demonstrates some example data stored in LDIF format describing "John Smith" and his contact details (this example is taken from page 55 of [12]).

```
dn: cn=John Smith, ou=people, o=ibm.com
objectclass: top
objectclass: organizationalPerson
cn: John Smith
sn: Smith
givenname: John
uid: jsmith
ou: Marketing
ou: people
telephonenumber: 838-6004
```

Figure 3.1: An example entry stored in LDIF format, taken from [12]

It should be fairly clear how an LDAP system could benefit from an encrypted search scheme. It would, for example, allow a company to host their entire employee directory in an outsourced datacentre that they are not willing to fully trust. This could potentially save a large amount of money by not storing all of the confidential information regarding their employees on in-house servers that would have required regular maintenance and upgrades.

The implementation of a scheme into an LDAP system would require modification of both the client and the server. The storage format used by the server would have to be modified to store the documents encrypted using the scheme, rather than in LDIF. The client would also have to be modified to include a key management process, automatic trapdoor creation for search queries and decryption of results. Extensions to the scheme could be employed to squeeze out extra performance by taking advantage of the LDIF format, e.g. a row based encryption scheme, or an index that takes advantage of the mapping of keys to values of each LDIF row (secure indexes are introduced in Section 4.2.2).

A number of implementations of LDAP systems exist, some open source and some proprietary. Open source variants include OpenLDAP [38] (written in C and released under its own open-source licence) and the Apache Directory Server [23] (written entirely in Java and released under v2.0 of the Apache Licence).

3.6 Conclusion

Each of the application areas discussed have different advantages and disadvantages, and come with different requirements for search.

- Mail servers are, by their nature, a public-key problem. As I will discuss in Chapter 4, research in the area of searching asymmetrically encrypted data isn't quite as mature as that of symmetric cryptography.
- File systems are designed to be transparent to applications that access them. This means that an implementation of search features that are specific to the underlying data storage would have to be added on as extra API functions that the file system exports. Existing programs such as `grep` couldn't then make use of them without modification. If the underlying data is stored in encrypted, searchable form, existing applications such as `grep` wouldn't know how to handle them, making a number of separate applications specific to encrypted search necessary. This adds a learning curve for users and muddies the transparency of the system.
- LDAP servers almost always use an underlying database server (OpenLDAP, for example, uses Berkeley DB), so it makes little sense to develop a system for an LDAP server over a generic database server.

Databases are used as the foundations of many different programs and as such are an obvious candidate for encrypted search. They are also often outsourced to datacentres. They are generic data stores, and they tend to be the 'lowest common denominator' when comparing different applications and their data storage techniques. An implementation of a secure search scheme in a database server could be easily utilised by a wide variety of applications that make use of the chosen database server as their data storage method.

Because of this, PostgreSQL was chosen as the system in which to implement the encrypted search scheme. It is very popular within industry and highly modular in its design, making the integration of a secure search scheme relatively simple.

4 Background

4.1 Cryptography Fundamentals

In order to explain the latest research in the area of searching over encrypted data, the fundamental building blocks used by the search schemes must first be covered. I have assumed a basic knowledge of what encryption is and will explain only the information that I don't consider well known. In the following sections, I will discuss the necessary fundamentals and any specific details that are made use of by the search schemes.

4.1.1 Block Ciphers

Block ciphers do what their name suggests – they manipulate a fixed size block of input (plaintext) into a fixed size block of ciphertext. A block cipher will utilise all of the bits of the input block simultaneously whilst generating the ciphertext, so modifying a single bit in the plaintext block will result in an entirely different ciphertext block (rather than a slightly different ciphertext block, possibly only differing from the original ciphertext at the same bit index as the bit that was modified in the plaintext).

A fixed sized key and block size are normally specified by the particular algorithm (sometimes with a number of choices). A type of padding will also have to be specified, as the input data will often not be an exact multiple of the algorithm's block size. There are a number of different padding schemes, including:

- Simply appending 0-bits. This, on decryption, will often result in extra, unwanted null bits/bytes at the end of the plaintext. As such, this type of padding should only ever be used on data that *doesn't* rely on a fixed structure, or contain any checksums (such as MP3 audio or MPEG video files).
- PKCS #7 [2]. This padding scheme counts the number of bytes that need to be appended to the plaintext in order to bring it to a multiple of the algorithm's block size. If only one byte needs to be added, the byte `0x01` will be added to the plaintext. If two bytes are necessary, `0x02 0x02` will be added. Three bytes, `0x03 0x03 0x03`, and so on.

Block ciphers can be used in a number of different *modes* of operation. These modes do not alter the way the block cipher algorithm itself works internally, but rather how the different input and output blocks are used. Different modes offer different levels of security and are sometimes designed for particular purposes (such as XTS mode, which is designed specifically for use in encrypting hard disk sectors). The two modes of operation that are made use of by the different search schemes are ECB and CBC.

- **Electronic Codebook (ECB)** mode is the most basic of all cipher modes. In this mode, the block cipher will read in one block of plaintext at a time and produce a block of ciphertext as output. This mode is also the least secure, as it is highly susceptible to statistical attacks – matching blocks of plaintext will produce matching ciphertext blocks, which leaks structural information about the document to a potential attacker.

- **Cipher Block Chaining (CBC)** mode is the next step up from ECB. When encrypting a block of plaintext, it is combined with the ciphertext that was generated from the *previous* block using a `XOR` operation prior to encryption. This means that two identical blocks of plaintext within the same document should not produce identical ciphertext.

  Because encrypting a block of plaintext relies on the previous block of ciphertext, an *Initialisation Vector* (IV) must be introduced in order to encrypt the first block of plaintext. This IV is used in place of the ciphertext from the previous block and is usually derived from the key that is being used, or it is randomly generated and stored in plaintext along with the ciphertext.

A wide variety of different block ciphers have been developed, with some becoming popular whilst others proven insecure. The Data Encryption Standard (DES) was one of the most widely used block ciphers for many years until it was deemed potentially insecure due to its small key size, making it susceptible to brute force attacks. Triple-DES was then introduced which increased the key size by applying the DES algorithm three times. The Advanced Encryption Standard (AES) is now the most obvious choice for a block cipher. First published under the name Rijndael, it was adopted by the USA's National Institute of Standards and Technology (NIST) for use as the government's official block cipher [1].

4.1.2 Stream Ciphers

A stream cipher is effectively a pseudo-random number generator, seeded on a private key. It will generate a *stream* of bits which are then combined with a plaintext input stream using a `XOR` operation. Because the generator's seed is private, the stream can only be reproduced by the person who knows it and the plaintext restored.

Because the cipher stream is combined with the plaintext stream on-line (bit-by-bit, side-by-side), a change to the plaintext will only result in a change to the the corresponding ciphertext bits (not every ciphertext bit that follows). Because of the random nature of the cipher stream, identical input 'blocks' will not produce matching ciphertext 'blocks'.

Some popular stream ciphers include A5 (used in the GSM mobile telephone standard) and RC4.

4.1.3 Hashing

The one-way-hash, also known as the pseudo-random function, is such a well known topic that I won't explain what they do, other than to say they transform an arbitrarily sized block of input data into a statistically unique output block of a predetermined length. Different hash algorithms will produce different length outputs, some will be faster than others at producing the output and some are considered more secure. MD5 was long used as the primary all-purpose hash algorithm until an attack technique was developed [21] that made the task of finding two values that would be hashed to the same known value (a collision) much more feasible.

Some popular hash algorithms currently in use are SHA-1 (and its bit-length derivatives: SHA-256 and SHA-512), Tiger and WHIRLPOOL.

4.1.4 Public-key Cryptography

Public-key algorithms are the solution to the old problem of how to communicate with a person without having to tell them a secret in the first place. Also known as asymmetric algorithms, they are typically far slower than their symmetric counterparts (such as block ciphers) and so are often only used to exchange

a key which is subsequently used for *symmetric* encryption. Since their background is fairly common knowledge, I will not explain it here. Some notable public key ciphers currently in use include the very popular RSA, ElGamal and Cramer-Shoup.

4.2 Prior Art

In the following sections, I will discuss the current state-of-the-art in the area of searching over encrypted data. I will discuss the internals of each search scheme before talking about their advantages and disadvantages, specifically the space and computational overhead, as well as how they can cope with the possible search options that were discussed in Chapter 2.

Note to the reader: I have attempted to keep the notation used throughout the following sections consistent. E, when used as a function will always refer to *encryption* performed using a block cipher. A subscript expression following the E denotes the key used – $E_k(W)$, for example, describes "W" being encrypted by a block cipher using the key k. The subscript notation is also applied to hash functions, usually termed $f(X)$ or $F(X)$ (these two would describe *different* hashes). If a subscript expression follows the name of the hash function, this expression is to be *combined* with the argument, before the hash is computed. The specific method of combination is not relevant, but should be consistent throughout. $f_k(x)$, for example, could be implemented as `hash(k XOR x)`, or maybe `hash(k + x)`. $\downarrow_n$ is used to denote the *projection* of the n^{th} element in the preceding tuple.

4.2.1 The First Attempt

In 2000, *Practical Techniques for Searches on Encrypted Data* [19] was published by Song et al. In this paper, the authors develop a set of algorithms that allow searches over encrypted data. These algorithms provide a linear search ($O(n)$) for each document and introduce relatively little space overhead. Proofs of the security of their model are also included which show that the server the data is hosted on *"cannot learn anything about the plaintext given only the ciphertext"* [19].

The encryption and decryption algorithms are fairly simple processes that I shall now explain in-depth. I'll start with the encryption process, as the decryption will be easier to understand afterwards.

Encryption

For a set of documents, the following is repeated once for each document. This should be done by the client, before uploading it to a remote, untrusted server.

Firstly, the input document is tokenised into a set of words, W. This tokenisation needs to still contain *all* of the input symbols, whilst separating words from punctuation. So, for example, a sentence such as "Something, something2!" would need to be transformed into the strings
{'`Something`', '`,<space>`', '`something2`', '`!`'}. The bundling of the first comma and space together into the same 'word' is probably acceptable, as it is unlikely that a user would want to search for ", ".

Once the document has been tokenised, the following process is performed on each word, W_i:

1. Three keys need to be generated, based on the private key given for encryption (e.g. a password). These keys are k', k'' and k''' and they are used at different stages in the process. These keys must be different and should be derived from the master private key in such a way that knowing k' doesn't reveal either k'' or k''' (as well as all other combinations). This allows us to reveal only one or two keys to an untrusted server, giving it enough information to perform a search, but not enough to decrypt the document or understand what we're searching for. Another reason for creating more keys based on one original key is to produce keys of the required bit lengths.

2. Word W_i is encrypted with a standard block cipher. This can be performed using either ECB mode, or CBC mode with a fixed IV. The key used for this block cipher (k'') is generated from the private

key in the previous step (probably a hash of the master private key which brings it to the size required by the block cipher being used).

$$X_i = E_{k''}(W_i)$$

3. The next step is to take x bits from the stream cipher. This cipher (G) is seeded on the key k'''. x must be less than the length of the encrypted word, X. I will refer to these bits as S_i. The choice of x should be pre-determined and be consistent throughout the system.

4. The encrypted word, X_i, is then split into left and right halves (L_i and R_i) where the length of L_i is x and the length of R_i is $length(X_i) - x$.

$$X_i = \langle L_i, R_i \rangle$$

5. A word specific key, k_i, is then created by combining the left half, L_i, with the key k' before hashing it.

$$k_i = f_{k'}(L_i)$$

6. S_i is then combined with this key (k_i – either through a process such as concatenation or `XOR`) before being hashed to produce a number of bits, equal in length to that of R_i.

$$F_{k_i}(S_i)$$

7. The final step towards producing the ciphertext is to perform a `XOR` between $\langle L_i, R_i \rangle$ and $\langle S_i, F_{k_i}(S_i) \rangle$.

$$Ciphertext = \langle L_i, R_i \rangle \oplus \langle S_i, F_{k_i}(S_i) \rangle$$

This process (depicted more elegantly in Figure 4.1) may seem convoluted, but each step serves a purpose that should become more clear when I discuss the decryption and search algorithms.

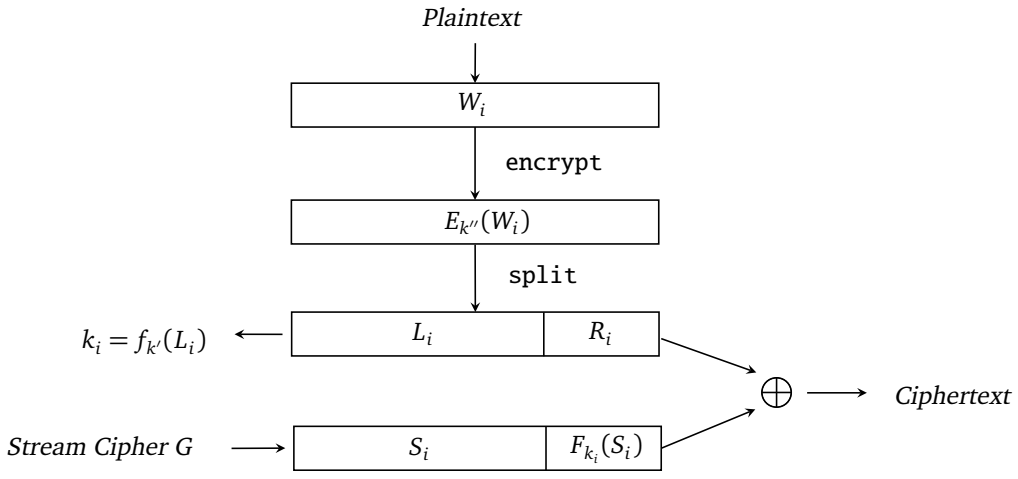

Figure 4.1: The encryption scheme proposed by Song et al.

Search

So, with a set of documents encrypted using the algorithm described previously uploaded to an untrusted server, how can we search for a given word and what information will this search leak to the server? Given a word W, the local, **trusted client** will perform the following:

1. Generate the same three keys as used in the encryption process, k', k'' and k''' (all derived from the master private key using the exact same processes).
2. Encrypt word W using the same block cipher and key (k'') as the encryption process to produce the encrypted word, X

$$X = E_{k''}(W)$$

3. The left part (L), consisting of the same x bits as used in the encryption process, is then extracted and used to generate the word-specific key, k (word-specific as it is derived from a *word* in the document). This is derived as in step 5 of the encryption process: by combining they key k' with the left part of X before hashing them.

$$k = f_{k'}(L)$$

The client can now send $\langle X,\ k \rangle$ to the **untrusted server**, which will search by performing the following algorithm on each document in its collection.

1. For each encrypted word block C in the document, `XOR` it with the encrypted word X. This will result in the $\langle S_i,\ F_k(S_i) \rangle$ pair.
2. Because the the length x is known (used in step 3 of the encryption process), we can now take this number of bits from the front of the $\langle S_i,\ F_k(S_i) \rangle$ pair to retrieve S_i, the bits that were taken from the stream cipher during the encryption phase.
3. The final stage is simple. Since both S_i and k (handed to the server by the client in order to search) are known, S_i can be combined with k and hashed using the same process as encryption (step 6) and compared to the right part of the pair - $F_k(S_i)$. If this matches, then the word is found and the current document can be added to a list of documents, ready to be returned to the client after the entire document set is inspected.

This process leaks *no* information about what word is being searched for to the server, whilst still allowing it to determine matching documents.

Decryption

Decryption should now be a fairly obvious process, similar to that of searching. To decrypt a document that has been downloaded to the local, trusted client, firstly the three keys k', k'' and k''' should be generated. After this, the client should iterate over each encrypted block, C, in the document as follows:

1. Take x bits from the stream cipher (seeded on key k''') to create S_i before `XOR`ing them with the first x bits of the ciphertext block, C. This will reveal the left part of the ciphertext word, L. Now we need to determine the right part, R.
2. Since we know S_i and L, we can generate the word specific key k as we did in the encryption process.

$$k = f_{k'}(L)$$

3. Using this key, k, we can now generate $F_k(S_i)$. which can be used to fully restore the encrypted word, X. Since we know the pair $\langle S_i,\ F_k(S_i) \rangle$ and we know the ciphertext, we can perform a `XOR` between these to retrieve X.
4. X was encrypted using a standard block cipher, to which we know the key (k''), so we can trivially retrieve the plaintext word.

Analysis

There are a number of key points in the above algorithms that should be reiterated for clarity. Firstly is the reasoning behind the need for splitting the encrypted word, X, into the $\langle L, R \rangle$ pair. The purpose of this split should be evident from steps 2 and 3 of the decryption algorithm. The stream cipher bits, S_i, are *combined with the key* k before being hashed to produce the second part of the pair $\langle S_i, F_k(S_i) \rangle$. This pair is used during the search stage to determine whether a given block matches or not. After the first stage of the decryption algorithm, all we know is S_i and the ciphertext. In order to produce the second part of the pair $\langle S_i, F_k(S_i) \rangle$, we have to produce the word-specific key. If we *hadn't* split the encrypted word X into an $\langle L, R \rangle$ pair, this key could have been produced by hashing the entire word: $k = f_{k'}(X)$. But on decryption we know only the ciphertext and the stream cipher bits S_i, which would not have been enough for us to reconstruct the key. Hence, X is split and the key k is generated based on only the left part, which can be reconstructed during the decryption stage by performing an `XOR` between S_i and the first x bits of the ciphertext C.

The choice of x is important. It should be a predetermined value that depends on the size of the blocks in use. The left part, L, of the encrypted word is hashed to produce R and as x is increased, the length of L increases and the length of R decreases. Because R is formed by computing a hash of L, as the space that R occupies shrinks, the available set of possible hash outputs also decreases. Because of this, larger values of x will result in the increased possibility of documents being returned that actually don't contain the search term due to hash collisions – these documents are known as false positives. Because of this, the client must take care to manually decrypt and search the returned documents in order to verify them. A good value of x should allow an R large enough to produce few false positives.

The iteration over output words in both the search and decryption algorithms relies on the input words being of fixed length. In the paper, Song et al propose choosing a block length that should be capable of containing most possible input words, with words shorter than the block being padded and words that are too long are split into multiple blocks (with possible padding on the last block). Another option is to allow for variable length input/output words. This could be combined with prepending the length of the following block to the block itself, but as the paper states, this could lead to possible statistical attacks. Variable length output ciphertext words *without* length information added would require the server to perform a search at every bit index, which would obviously greatly increase how computationally expensive the search operation is (as well as decryption). The determining factor between these two variable word length schemes is whether space is valued more than CPU time.

This scheme can also cause problems when the input document contains a large amount of punctuation, or characters that you do not want to be included in possible searches. As described earlier, a document containing the string "Something, something2!" should be returned when the user searches for "Something". Because of this, alphabetic characters clearly need to be separated from punctuation and spaces and encrypted into separate blocks. This means that a typical English document, such as one of Shakespeare's plays, would generate quite a large space overhead due to the necessary addition of a full encrypted block for every piece of punctuation[1].

Of the search considerations listed in Chapter 2, this scheme also suffers from not being capable of handling a number of them. Sub-matches are not supported due to nature of the whole-word encryption, since the block cipher will produce wildly different outputs for two words, even if one is the prefix of another. Since natural language support through word stemming relies on sub-matches, this is not supported either. Case insensitivity and regular expressions are also not supported for the same reason[2]. However, this scheme *is* capable of handling proximity searches which can be supported through multiple queries combined with primitive server-side logic.

[1] This, obviously, is assuming we choose the fixed block length over the variable length scheme described in the previous paragraph.

[2] Regular expressions could, potentially, have primitive support through manual explosion by the client, followed by a chain of search requests. For example, the regular expression `a|b` could be transformed into two searches by the client, one for `a` and one for `b`. Obviously, more complicated regular expressions will quickly become infeasible using this method.

Computationally, for a large data set, this scheme can be very expensive. It is linear in the size of each document and so quickly degrades when faced with large, real-life data. It was, however, the first real effort towards a searchable encryption scheme and has paved the way for further research, as described in the following sections.

4.2.2 Secure Indexes

In 2004, Eu–Jin Goh published *Secure Indexes* [9]. In this paper, he describes a scheme for generating a cryptographically secure pre-processed index for a given document and the associated search process. This scheme, due to it using an index, provides a constant time ($O(1)$) search of a document (so the final dataset search is linear in the number of documents, rather than their size). The index is generated by the trusted client from a plaintext document, before being uploaded alongside the document after it's encrypted with a *standard* block cipher (such as AES) to an untrusted server.

Bloom Filters

Goh's search scheme makes use of a datatype known as a Bloom filter [3]. This structure provides a fast set membership test, with possible false positives[3]. A Bloom filter is stored as an array of bits. Initially, a Bloom filter's internal array is set to all 0s. When an element is added to the set, a number of hashes are performed. The input to all of these hashes is the element being added, and the output from each is an index into the array (normally different for each hash). After these hashes are calculated, each bit in the filter at the indexes specified by the hash outputs is set to 1.

An example Bloom filter, using three hash algorithms is show in Figure 4.2. This figure also shows the addition of three different words, `x`, `y` and `z`. As this image shows, it is quite possible for two hash outputs to result in the same bit-index, in which case the bit that is already set to 1 is left as-is.

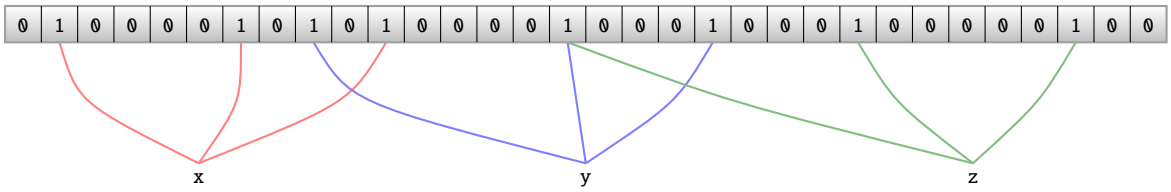

Figure 4.2: A 32-bit Bloom filter with three hashes (the number of lines from each word). The words `x`, `y` and `z` have been added as members.

To test an element for membership, it is hashed using the same algorithms and the conjunction of the resultant bits is returned (so only if they are all set to 1 is the element considered a member). Clearly, due to possible overlaps and an increased saturation level as progressive elements are added, false positives can occur when all the bits specified by the hash outputs are set to 1, even when the element wasn't originally added to the filter.

The different hash algorithms can be created in a number of different ways. One option is to use a hand-implemented set of distinct hash functions. This choice can be quite limiting, and quickly makes the Bloom filter complex to implement. Double and triple hashing trivially triple the number of distinct hashes, but this will still mean developing and implementing at least two unique hash functions, producing six hashes in total. The number of hash algorithms to produce an acceptable false-positive rate will vary dependent on the number of elements and will be explored more thoroughly later.

Understanding how a Bloom filter works, we can now look at Goh's secure index scheme in detail. I will start by describing the key generation process, before looking at encryption, then the search and decryption algorithms.

[3]That is, the test for membership might return true when the member being queried actually doesn't exist in the set. It is *not* possible, however, for false negatives to be returned (a test for membership of an entry that *does* exist in the set returning false).

Key Generation and Trapdoors

The key generation algorithm, `Keygen(s)` takes a security parameter (e.g. a password) and generates a master key, K_{priv} which is, in turn, comprised of a set of r keys. These will be used throughout the scheme. A specific implementation of this might be a 512-bit hash (maybe using SHA-512) of the input password to create the master private key, K_{priv}, which is then split into sixteen 4-byte keys ($r = 16$).

$$K_{priv} = (k_1, ..., k_r)$$

A *trapdoor* is a transformation of the term being searched for such that an untrusted server can find potential matches, without gaining knowledge of the plaintext. In this secure index scheme, a trapdoor is formed of the input word W, the private key K_{priv} and a series of hash functions. This algorithm is described as `Trapdoor(`K_{priv}`,` W`)` and given the necessary arguments, the trapdoor T_w is computed (where f is a suitable hash algorithm) as

$$T_w = (f_{k_1}(W), ..., f_{k_r}(W))$$

Encryption

The encryption process centres around generation of the index. `BuildIndex(`D`,` K_{priv}`)`, as it is termed in the literature, takes the private key and a document, D which consists of the plaintext and a unique identifier, D_{id} and returns a Bloom filter representing the document index. As I will explain, the document identifier D_{id} is used to stop two identical documents from generating the same index (or documents containing a large number of repeated words from generating similar indexes).

The client, given a document and the private key can then create the document's index as follows.

Firstly, the document is split into a set of words. Unlike Song's algorithm, we can actually *ignore* punctuation (unless we predict the user is likely to want to search for it in our particular application). This is thanks to the fact that the entire, unmodified document will be encrypted and uploaded to the server, whilst the index can merely contain words that the user is likely to search on[4].

For each word, W_i, the following algorithm is then performed.

1. A trapdoor is constructed from the word W_i and the private key K_{priv} using the `Trapdoor(`K_{priv}`,` W`)` algorithm described previously.

$$T_w = (f_{k_1}(W_i), ..., f_{k_r}(W_i))$$

2. A *codeword* is then constructed based on the trapdoor T_w. This simply takes each element of T_w and hashes it with the document ID.

$$C_w = (f_{D_{id}}(T_w \downarrow_1), ..., f_{D_{id}}(T_w \downarrow_r))$$

3. This codeword can now be added to the Bloom filter that represents the document index.

The index created through this step could now be used as the document index, however Goh goes on to introduce a process of *blinding* the Bloom filter with random noise in order to further discourage any potential statistical analysis.

The blinding procedure starts by calculating u as the number of tokens (one byte per token is suggested by Goh as a reasonable estimate) in the *encrypted* version of the document's plaintext. This can be calculated as the length of the plaintext, plus any necessary padding. v is then calculated as the number

[4]In this case, all of the words in the document are used, but for efficiency in certain applications, a list of key words could be used and only words found on this list could be included in the index. This would obviously decrease the number of terms that can be successfully searched for, but for applications where only a distinct set of search terms are possible, it would greatly decrease the size of the index and thus the space overhead.

of *unique* words in the document. Once this is done, $(u - v) * r$ 1 bits are inserted *at random* into the Bloom filter (where r is the the same as was used in the `Keygen` algorithm, and hence the number of tokens in the codeword C_w).

Once the Bloom filter has been blinded, it can be returned by the `BuildIndex` algorithm as the index $\mathscr{I}_{D_{id}}$ for the document D.

Once the index is constructed, the plaintext document is encrypted using a standard block cipher and the private key K_{priv}. The tuple containing this encrypted document, the document identifier D_{id} and the index can then be uploaded to the untrusted server.

$$Ciphertext = \langle D_{id}, \mathscr{I}_{D_{id}}, E_{K_{priv}}(D) \rangle$$

Search

When the user wants to perform a search for word W, the trapdoor T_w for the search term is generated using the `Trapdoor(`K_{priv}`,` W`)` algorithm. Once this is generated, it can be handed to the untrusted server which will then iterate over all of its stored documents and perform the following:

1. Generate the *codeword* from the trapdoor in the same manner as previously.
2. The document's Bloom filter index is then checked to see if this codeword is a member.
3. If the Bloom filter replies positively, the document is added to the set of documents to return to the user.

The trapdoor is used to hide the search term from the server, whilst the second-stage codeword is used to cleanly separate indexes of documents with similar content.

Analysis

As I have already said, the Bloom filter is capable of returning false positives. Because of this, the document set returned to the client must not be taken for granted – each document returned will need to be decrypted and manually searched client-side. This may sound like a huge pitfall in the scheme, but in reality, the false positive count is small. It's also a nice way of obfuscating the actual result set from the server, which would see more documents than necessary returned by a query, therefore reducing the amount of information available to it for cryptanalysis.

A simple extension of this that allows for server-side searches that reduces the possibility of false positives[5] is to use Song's algorithm to encrypt the document. First, the document index produced by Goh's algorithm is consulted and then before the documents are returned, Song's algorithm is used on the encrypted document body as a second check, enforcing the decision on whether the document contains the search term. This approach won't reduce computation time (unless the server is more powerful than the client), but it could potentially reduce network communication overhead.

One problem that should be obvious is the possibility of saturating a Bloom filter. If incorrect parameters are used, such as the size of the bit array being too small, or the number of hashes too large, a large document could potentially result in a Bloom filter comprised of all 1s. In this situation, the result set of *any* search query will include this document. Goh describes a way of determining an appropriate number of hash algorithms to use and bit array size in Section 5 of his paper [9]. Firstly an acceptable false positive rate (fp) must be settled on by the implementation. With this decided, the number of hashes is calculated as $r = \lceil -log_2(fp) \rceil$ and the size of the array necessary as $m = \lceil \frac{nr}{ln\,2} \rceil$ (where n is the number of unique words in the document *set*).

A nice extension of Goh's use of Bloom filters, given in an Appendix to his paper, is that of hierarchical search. He suggests that Bloom filters can be combined hierarchically (perhaps mirroring a folder

[5] This isn't necessarily a *good* thing, as false positives act as noise to further confuse the server over which documents contain the search term.

structure of a file system, with a Bloom filter for each directory node) through a disjunction. A tree of Bloom filters is produced with each successive layer providing more specific search results. If there are no documents within a folder structure that is decorated with this system that contain a certain term, searching for this term starting with the top-level directory will likely terminate faster than a flat, linear search of all child documents.

One problem with the usage of plaintext *document identifiers* is that these themeselves might contain information that should be kept secure – a lot could potentially be derived from the name of a document. A simple solution to this is to use a meaningless identifier (such as an incremented number, or a hash of the document's actual name), which can be mapped by the client application to its meaningful representation after the search results are returned.

In Chapter 2 I discussed possible search types and options. Although the secure index scheme can trivially support exact-match searches, it cannot handle sub-matches or regular expressions (except for the primitive explosion method described in Section 4.2.1). This is, again, due to the nature of the pseudo-random hash functions used in the trapdoor and codeword steps; an entirely different output will be generated by two different words, even if one is the prefix of another.

Case-insensitivity is also not directly supported, but it could be extended quite easily by inserting every word into the Bloom filter twice: firstly as it exists in the document, and secondly after being converted into lower case. A case-insensitive search could then be carried out by querying the server with the trapdoor of the lower case search term. Obviously, necessary modifications would have to be made to the Bloom filter parameters in order for it to handle the increased number of tokens. Another option is to store two Bloom filters with each document, one for exact matches and one containing all lower case codewords. This would, however, increase the disk space required for the scheme and increase the amount of information available to an adversary for cryptanalysis.

Although natural language searches are not natively supported, the above techniques could be employed for this too. The stem of each word in the document can be added to the Bloom filter (or a secondary index specifically holding word stems). A client-side search program could take a user entered query and derive the word stems, before generating the necessary trapdoor and querying the server.

A Prototype Implementation

As part of my background research, I created a prototype implementation of Goh's secure index scheme. It is written in C♯ and is architected as a pair of applications: a client and a server. Using this implementation, I was able to investigate the efficiency of the algorithm as well as its space overhead. I used a corpus consisting of 501 RFCs [48] (specifically those numbered 2001-2500) totalling 27.6MB (28,956,621 bytes). Using a machine with an Intel Core 2 Duo clocked at 2.33GHz, 2GB of RAM and a 7200rpm hard disk, I encrypted the corpus[6], which took a total of *12 minutes, 5 seconds*.

After encryption, the dataset weighed in at 30.3MB (31,858,514 bytes), giving a *9.1%* increase in size and a datarate for the encryption process of *39KB/s*. I then performed a number of searches over the encrypted data, detailed in Table 4.1.

Term	# Results	# False Positives	Avg. Search Time
"IMAP"	42	12	176ms
"octet"	182	6	291ms
"Ossifrage"	8	8	161ms
"TLS"	23	7	166ms
"test"	129	12	265ms

Table 4.1: A number of search queries and their respective statistics.

[6]The password used was `aE2\HUYq8sVzL5@g4tj6`

From this table, we can see that the search time is reasonably consistent, with, I suspect, the larger number of results being transmitted causing added network delay. The saturation levels of the encrypted files' Bloom filters are between 10% and 40% (of bits that were set to 1), which seems acceptable. I decided on a false positive rate of 10%. Clearly this implementation is slightly flawed, as the average false positive rate from these results (ignoring the *Ossifrage* search, which is anomalous due to it not existing in the source data) was just under 18%. This could be due to my choice of hash not distributing evenly, or perhaps it would average out to around 10% as more searches are performed.
My choice of algorithms for the implementation was as follows:

- **AES** was used as the block cipher. This is made use of to encrypt each full document.
- **SHA-512** was used to generate the private key. This is used to construct trapdoors and to encrypt the documents.
- An implementation of the **DJB Hash** [62] was used by the Bloom filter to generate bit-indexes from the codewords to be inserted. This is used over a stronger algorithm, such as SHA-1, due to its speed.
- My implementation of the **Bloom filter** makes use of a counter to generate r hash functions, rather than individual implementations of specific hashes. This is done by appending the counter to the word, prior to hashing it.

This implementation concentrated on code clarity over performance, which can clearly be seen by the long time taken to encrypt such a small dataset. This is obviously not good enough to be used by any serious industrial application.

4.2.3 Improved Secure Index Construction

In 2006, *Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions* [7] by Curtmola et al was published. In this paper, the authors review previous work in the area of searching over encrypted data and propose an improved scheme. The distinguishing factor of this scheme over the previous efforts is its speed. Where Song's algorithm is linear ($O(1)$) in the size of the documents and Goh's is linear in the *number* of documents, Curtmola's search scheme will execute in constant time ($O(1)$). They also go on to discuss a further extension to their own scheme that is capable of staying secure, even when faced with an *adaptive adversary*: one that is allowed to see the outcome of previous queries.

The construction is formed of a combination of a look-up table (`T`) and an array (`A`). Firstly, for each unique word in the document set, a linked list is created that contains the list of document *identifiers* in which that word is found. Once all of the linked lists for all words are created, they are flattened, encrypted (using a unique key for all elements of the list) and *scrambled* into the array `A`. Before being encrypted, each element is packaged with the key used to encrypt the next element in the list. The look-up table makes location and decryption of words in `A` possible by containing references to the elements that were, before being flattened, the heads of each linked list. This allows the server, given an entry in the look-up table and the key that was used to decrypt the first element of this list, to decrypt all of the elements in the list, resulting in a list of all documents in which the search term was found. The scheme, similarly to Goh's secure index scheme, consists of four algorithms:

- `Keygen`(k)

  This algorithm takes a security parameter, k and generates three random keys, s, y and z each of length k. These three keys are then combined to produce the the private key, K.

  $$K = (s, y, z)$$

- `BuildIndex`(K, $\mathcal{D}$)

  This algorithm builds the document index for a given document using the given key. Firstly, the set of all unique words that are to be made searchable is constructed, before a global counter `ctr` is initialised to 1. After this, the array `A` is constructed. The following process is performed on every unique word, w_i found in the document set.

  1. The list of documents in which the word was found is created ($D(w_i)$).
  2. An initial key, k_{i_0} is generated. This is the key used to encrypt the 0th element in the set of documents that contains word w_i (hence the i_0 subscript).
  3. For each successive element (N_j) in the set of documents that contain word w_i ($D(w_i)$):
     (a) A key for this element is generated as k_{i_j}. This will be used to encrypt the next node in the set.
     (b) Construct a tuple, consisting of the jth document identifier, D_{i_j} (the one we are currently iterating over), the key to the next tuple that we just generated (k_{i_j}) and the index in `A` of the next element in the list. This index is generated using a pseudo-random function (hash) and takes `ctr + 1` as its input. The output of this hash function will have to give a unique index into `A` for each element. If this element is the *last* element of the set of identifiers, the address to the next element is set to `null`.

     $$\langle \texttt{id}(D_{i_j}),\ k_{i_j},\ \texttt{address(ctr + 1)} \rangle$$

     (c) This tuple is then inserted into the array `A` at index `address(ctr)`, before `ctr` is incremented.

  If the array `A` contains any empty cells, they should be set to random values, generated using a pseudo-random function.

  The look-up table `T` is now constructed. For every word w_i in the document set, firstly a `value` is created. This is built by first creating a tuple containing the address in the array `A` of the first element of the linked list for word w_i and the key that was used to encrypt this element.

  $$\langle \texttt{addressof(A[}N_0\texttt{])},\ k_{i_0} \rangle$$

  This would be enough to perform a search, but is not secure. To secure this `value`, it is `XOR`ed with the result of a hash of the word w_i, with the key y (generated as part of the master key in the `Keygen` algorithm).

  $$\texttt{value} = \langle \texttt{addressof(A[}N_0\texttt{])},\ k_{i_0} \rangle \oplus f_y(w_i)$$

  The address that this value should be placed at in the look-up table must now be calculated. This is done using another pseudo-random hash function, F, combined with the key z (generated as part of the master private key K in the `Keygen` algorithm).

  $$\texttt{T[}F_z(w_i)\texttt{]} = \texttt{value}$$

- `Trapdoor`(w)

  The trapdoor for word w is generated as a tuple of two hashes of w, one with the F_z algorithm/key used for the address in the look-up table, and one using the f_y algorithm/key used for encryption of the `value` of this look-up table entry.

  $$T_w = (F_z(w), f_y(w))$$

- `Search(`$\mathscr{I}$`,` T_w`)`

  The all important search algorithm takes a document index, $\mathscr{I}$ and the trapdoor T_w to search for. The first element of this trapdoor specifies the key to the look-up table, so this is retrieved

  $$\texttt{value} = \texttt{T}[T_w \downarrow_1]$$

  The server can now `XOR` this value with the second part of the trapdoor to retrieve the tuple containing the index into `A` that contains the first element of the linked list describing the documents that contain word w

  $$\langle \texttt{addressof(A}[N_0]\texttt{)}, k_{i_0} \rangle = \texttt{value} \oplus (T_w \downarrow_2)$$

  This now gives the server enough information to iteratively decrypt all elements in the linked list whose head is stored in $\texttt{A}[N_0]$ and thus, enough information to construct a list of document identifiers (and therefore, documents) that contain the word w. This list can now be returned to the client.

Analysis

This scheme provides a constant time search algorithm, which, for a large dataset, could prove to be a major advantage over all the other schemes. The key separation and trapdoor provides control over the amount of information available to the server. Given a trapdoor, it can only access the single linked list which represents the word. If there is no corresponding entry in the look-up table, the word is determined to not exist. Also, because of the way the array entries are scrambled before being added to the array (through the use of a pseudo-random hash that produces an index), statistical analysis is made more challenging.

One rather large problem with this scheme is the need to update the array and trapdoor whenever a document is added or removed, as well as the fact the document index will increase in size linearly with the number of documents. This is worse than Goh's scheme because of its use of a Bloom filter which is a more space-efficient, non-linear storage structure. Also, as Goh's scheme uses per-document indexes, an update wouldn't require modification of any other document's index.

Looking at the search considerations in Chapter 2, this scheme can be seen to share the support for each with Goh's secure index scheme. Exact-match is the only natively supported option due to the nature of the pseudo-random functions used in generating the trapdoors and building the index's array and look-up table. As I suggested while discussing Goh's scheme (Section 4.2.2), an extension could be created to add support for case-insensitivity and natural language through the addition of extra indexes for each (i.e. one index contains exact-matches, one contains the list of unique *lower case* words from the document set and one contains the list of *stemmed* words). This would add a fair deal of space overhead (at most[7] tripling the space required by the index) but wouldn't leak a great deal of extra information, thanks to the pseudo-random scrambling of entries around the array `A` (each index will be scrambled differently).

4.2.4 Public-key Alternatives

So far, all of the schemes I have described have been symmetric – that is, they use a single private key for both encryption and decryption. Public key cryptography offers a solution to the key exchange problem that arises from symmetric cryptography. It is, however, a great deal slower process and so is typically used to exchange a key which is then used in a symmetric cipher. For this reason, the research into the area of creating a public key algorithm capable of performing search queries is less extensive and, in my opinion, less relevant to the potential areas of application. There has, however, been a number of efforts into this area [4, 8, 13] which I shall discuss now.

[7] This is at most, because some documents will likely contain a number of word variations that will resolve to one stem, or one lower case word.

PEKS

In 2004, *Public Key Encryption with Keyword Search* [4] was published by Boneh et al. In this paper, the authors describe a way of appending information to a message that has been encrypted with a standard public key cipher, such as RSA, which allows a server that doesn't have the private key necessary to decrypt the entire message to still be able to search for a certain set of keywords. For a keyword, a PEKS (Public-key Encryption with Keyword Search) value can be generated which will allow the server to perform a search using a trapdoor.

So, given a message M that contains the set of *keywords* $\Sigma = W_1, ..., W_i$, the ciphertext will consist of the following (where A_{pub} represents the *public* key of the intended recipient):

$$(E_{A_{pub}}(M), \texttt{PEKS}(A_{pub}, W_1), \ldots, \texttt{PEKS}(A_{pub}, W_i))$$

Similarly to the index schemes that I have already discussed, the scheme is split into four algorithms:

- `KeyGen(`Σ`)`

  The key generation algorithm, given the set of keywords Σ, will produce a pair of keys, A_{pub} and A_{priv} (the public and private keys, respectively). These keys are created by generating a public/private key pair for *every word* in Σ. The final private key (A_{priv}) is then constructed as the set of all the individual words' private keys and the final public key (A_{pub}) is constructed as the set of all the individual words' public keys.

- `PEKS(`A_{pub}`,` W`)`

  Firstly, a random number, M, is generated. The algorithm then returns the tuple containing this M and its value, encrypted with the public key associated with W (which is in the set A_{pub}).

  $$\langle M, E_{K^W_{pub}}(M) \rangle$$

- `Trapdoor(`A_{priv}`,` W`)`

  The trapdoor for a given (T_w) word W is simply its private key, K^W_{priv} as defined in the set A_{priv}.

- `Test(`A_{pub}`,`S`,`T_w`)`

  The test algorithm allows an untrusted server, given a trapdoor T_w, the set of public keys A_{pub} and an output from PEKS algorithm (S) to test to see if the word represented by the trapdoor matches the word used to generate the PEKS. This is done by taking the PEKS, S and attempting to decrypt the second element of the tuple with the trapdoor (the private key for the word). If the output of this decryption matches the first element of the tuple, the algorithm returns true, if not, false is returned.

  $$D_{T_w}(S \downarrow_2) \equiv S \downarrow_1$$

Analysis

This scheme provides a way for an untrusted server to test to see if a given message contains a certain keyword. One of its main flaws is clearly that the list of keywords has to be determined carefully in order to keep message size down (a low number of keywords) whilst allowing a variety of different searches (a large number of keywords). One of the applications that the paper uses as an example is that of a mail server. A user could use this scheme to send a message to a friend using their public key which would stop anyone, including the server, from reading the message. If the server attempts to do anything clever with messages it receives, such as moving all email containing the word "urgent" in the subject into a special folder, it will be unable to do so using a standard public key scheme. However, if the sender had made use of this scheme and appended `PEKS(`A_{pub}`,'urgent')` to the message, the

server, given the trapdoor `Trapdoor(`A_{priv}`, 'urgent')` by the person receiving the emails, could have determined successfully that the email sent did in fact contain the word "urgent".

Another problem is the scheme's performance. The public and private keys are generated as a set of other public/private keys which are in turn generated from a number of different input words. Public-key algorithms require large prime numbers to be calculated in order to generate usable keys, so this process is potentially very time consuming.

As with the majority of the schemes I have discussed, the search options (described in Chapter 2) available in this scheme are limited to exact-match. Case-insensitivity and natural language could be added in much the same way as I have described previously: by adding the lower case, or stemmed words, respectively, as PEKS to the end of the message (along with adding the necessary trapdoors to the server). This will obviously increase the size of the message.

PIR

The above PEKS scheme is really only suited to automated searches with a relatively small keyword list, as described in the previous example. A keyword list that contains all of the unique words in a document, simply appended to the end of the message would bloat the message greatly. Boneh continues his work on public-key search schemes by helping to author the paper *Public Key Encryption that Allows PIR Queries* [8], which uses a technique involving Bloom filters to help lower the space overhead and allows Private Information Retrieval (PIR).

4.2.5 Extension Work

A number of papers have been published that extend research that I have already discussed. I will now give an overview of these systems.

Efficient Tree Search in Encrypted Data

In 2004, *Efficient tree search in encrypted data* [15] by Brinkman et al was published. In this paper, the authors develop an extension to Song et al's work [19], implementing tree search over XML data along with support for XPath queries. Since their work is based on [19], its search time is linear in the size of each document, however they improve performance by taking the structure of the XML into account.

Their work also derives from previous research into storing XML in a relational database and improving the efficiency of XPath queries. Using this background, they made minor modifications to Song's algorithm in order to preserve the structure of the XML – each individual attribute name, value, tag name and contained text is treated as a variable length block. This, combined with the XPath speed-ups results in a fast encrypted XPath scheme that, unlike the standard Song algorithm when applied to XML, requires only a subset of the stored information to be searched. Depending on the structure of the XML, this can yield impressive performance increases.

Rank-Ordered Search

In 2007, *Confidentiality-Preserving Rank-Ordered Search* [20] by Swaminathan et al was published. This paper looks at the practicality of building a ranking system into a secure index based search scheme. It defines methods for ranking documents based on relevance whilst maintaining security, thereby allowing the user to remotely search over encrypted data without revealing the search term or the private key used to the server.

The encryption used on each individual document, similarly to Goh's secure index scheme, is a standard block cipher, whilst the secure index is generated prior to the document encryption phase. Initially, a ranking table of *term frequencies* is generated from the set of documents that are to be uploaded. This table is of size $T \times D$, where the T **rows** represent the *unique* terms (after being *stemmed* to produce the word from which they are derived) in the entire document set and the D **columns** represent documents. This table will allow a number to be stored for each term indicating how frequently the term appears in each document.

In order for the frequency table to be stored on the server securely, the key used to identify a given word's row in the table is encrypted (using a key derived from the word itself) and subsequently hashed. At this stage, only the words are represented by secure hashes – their occurrences within documents are not. Because of this, each row of the term frequency table is then encrypted using a key derived from the plaintext word associated with the row (this means that if one row is compromised, others are not).

Since the frequency table generated is likely to be quite sparsely populated, a further compression stage is included to reduce the space overhead required of the server. Once this is done, all the documents are encrypted using a standard block cipher and uploaded to the untrusted server along with the encrypted term frequency table.

When a user wants to query the server, firstly the query is split into individual terms which are then stemmed. Then, the stems are encrypted using a key derived from themselves. These terms are then hashed to produced possible keys to the term frequency table that is stored on the server. One at a time, these terms are transmitted to the server which looks up the relevant row in the frequency table and returns it, if found. The client can then decrypt the returned row before inspecting the frequencies. A standard method for determining relevance can then be employed, from simply ordering the results by the frequency of the term, to using a more sophisticated ranking algorithm. Swaminathan et al suggest using the Okapi [17] relevance score as a possible example. This score, however, requires a *collection frequency weight* per word, which would not be available if the client was only given access to a single row, as is the case. Because of this, the paper suggests computing this value during the encryption process and encrypting them with the same word key as used by the frequency table. This set of frequency weights could then be stored separately to the frequency table on the server.

This scheme, although fairly simple, provides a high level of security whilst allowing the sorts of relevance based queries that are being increasingly expected by users, thanks to the efforts of a number of internet search engines. The space overhead should be relatively small, with a compression algorithm being employed to squeeze it further. Since only a single row is returned for each query, the communication overhead is also fairly low, although higher than most others that don't require any communication other than the initial query and subsequent return of matching documents.

Since this is an index based scheme, the search options available are limited. The stemming process will allow for a form of natural language queries, and is necessary for effective ranking. Exact matches, however, are only supported through stemming, so there is no way to force the scheme to search for an exact query. The more complicated query types, such as sub match and regular expressions are not supported, due to the fact that whole, stemmed words are encrypted and hashed.

Others

There are a number of other papers in this field of research that I shall not cover in detail, but that I feel deserve mention.

- *Privacy Preserving Keyword Searches on Remote Encrypted Data* by Chang and Mitzenmacher [22] (2005) offers another symmetric key approach using secure indexes.
- *Public Key Encryption with Keyword Search Revisited* by Baek et al [13] (2005) takes a second look at [4] and considers a number of issues that the original paper had not addressed (such as multiple-keyword search).
- *Using Secret Sharing for Searching in Encrypted Data* by Brinkman et al [16] (2004) proposes using randomly generated polynomials and their differences to secure information on a remote server. A pseudo-random generator, seeded on a private key is used by the client machine to build a tree of polynomials which are then compared with those stored on the server in order to encode a tree of XML elements.
- *Executing SQL over Encrypted Data in the Database-Service-Provider Model* by Mehrotra et al [11] (2002) is focused on the issues involved in creating a database that is able to store encrypted

content, whilst being capable of handling SQL statements. Published prior to work in the area of secure indexes, their scheme is not overly efficient and is highly specialised.

4.3 Conclusion

Each scheme discussed has varying advantages and disadvantages. Some can be seen to be more useful than others in certain scenarios and some also have a larger potential for a number of different search options (as discussed in Chapter 2). Tables 4.2 and 4.3 give details of the different schemes' features and ability to perform different search options, respectively.

Scheme	**Search Complexity**[8]	**Search Type**	**Insert Requires Recalculation?**
Song [19]	$O(n)$	Linear	no
Goh [9]	$O(d)$	Pre-processed Index	no
Improved Index [7]	$O(1)$	Pre-processed Index	yes
PEKS [4]	$O(n)$	Linear	no
Ranked [20]	$O(d)$	Pre-processed Index	yes

Table 4.2: A summary of search schemes and their key features.

Scheme	**Exact Match**	**Sub-match**	**Case Insensitivity**	**Regex**	**Proximity**	**Stemming**
Song [19]	yes	no	no	no	yes	no
Goh [9]	yes	maybe	maybe	no	no	maybe
Improved Index [7]	yes	maybe	maybe	no	no	maybe
PEKS [4]	yes	maybe	maybe	no	no	maybe
Ranked [20]	no	no	yes	no	no	yes

Table 4.3: A summary of search schemes and their ability to perform certain search options.

The entries marked "maybe" describe the fact that the scheme doesn't natively support the search option, but through a number of techniques, such as adding more words to the index perhaps, these search options can be supported.

The clear winners in terms of computational complexity of searches[9] are Goh's Secure Index [9], the Improved Index Construction [7] and the Rank-Ordered Search [20] schemes. These are all either constant time in the number of words in each document (Goh's Secure Index and the Ranked-Ordered Search), or constant time in the number of documents in the set too (Improved Index Construction). The problem with the latter two is that an upload (e.g. an SQL INSERT) of a new document requires recalculation of the *entire* document set index. This is because the indexes generated by both schemes are used to index the entire document *set*, compared to Goh's secure index scheme, which will generate an index for each document. Because of this, an insert using Goh's scheme doesn't require modification to any other document's index.

As such, the usefulness of an implementation of either the Improved Index Construction or the Rank-Ordered Search scheme in a database management system like PostgreSQL is questionable, as it is an environment in which INSERTs could potentially occur frequently. Because of this, and the fact that it offers the possibility for a reasonable number of different search options, Goh's scheme was chosen to be implemented.

[8] Note that d is the number of documents in the document set and n is the number of words in a document.

[9] The computational complexity of *encryption* is $O(n)$ for all schemes, as they are all required to parse each document word-by-word.

5 Implementation

In Section 3.6, the PostgreSQL database system was chosen as the target application to be extended. Also, in Section 4.3, the encrypted search scheme chosen was the Secure Index scheme developed by Goh [9]. In this chapter I will detail the design and implementation of Goh's Secure Index scheme[1] within the PostgreSQL 8.3 database management system.

5.1 Preliminary Decisions

Prior to being able to begin implementation of the system, a number of decisions had to be made. As well as the architectural design of the system and choices relating to the underlying libraries to be used (described in the following sections), SVN was chosen as the version control system, and Eclipse CDT was used as an IDE.

5.1.1 System Architecture

Database systems are typically split into two main components: the client and the server. PostgreSQL is no different. It consists of a server application (`postgres`) and a client terminal application (`psql`). The latter is effectively a CLI wrapper around PostgreSQL's `libpq` library, which is the primary C API for developing client applications. In order to implement the search scheme, both the server and the client-side code had to be modified.

On the server-side, a new datatype needed to be introduced and an operator had to be defined that takes the new datatype as one of its operands. The datatype is used to store the encrypted documents and their indexes and the operator compares a document against a trapdoor. If the document index is found to contain the codeword that is derived from the given trapdoor, the operator will return `true`, indicating that the document was a successful match.

On the client-side, it is necessary for the search query, along with a key, to be transformed into a secure trapdoor (as the plaintext query and key cannot be sent to the server). Initially, it was planned to make modifications to `libpq`'s source code in order to perform this transformation of queries. The API offered by `libpq` is somewhat flat – by this I mean that its main purpose is to simply forward strings containing SQL commands to the server; it does very little processing of its own.

The primary function offered is `PGresult *PQexec(PGconn *conn, const char *command)`, which takes a connection and a command string. Because of this, the code necessary to transform queries, if implemented within `libpq`, would have needed to perform something akin to the following regular expression over each command string: "`SELECT [^ ]+ FROM [^ ]+ WHERE [^ ]+ ENCLIKE ([^ ]+) WITHKEY ([^ ]+)`". In this example, `ENCLIKE` would be the chosen name of the operator for searching an encrypted datatype, and `WITHKEY` would be used to specify the private key to use. If a query like this is found by some regular expression, implemented within `libpq`'s `PQexec` function, then code to

[1] The specifics of the scheme will not be gone over again, they can be found in the Background chapter, in Section 4.2.2.

transform the query, along with the key, into a trapdoor could be executed, and then this modified query (containing the trapdoor) could be issued to the backend server, rather than the initial query presented to the client library.

The main benefit of this system is that the transformation of the query/key pair into the trapdoor is completely transparent to the specific client application that is making use of `libpq`. This is a nice feature, as developers that are implementing client applications with `libpq` don't have to know anything about how the secure index scheme works – only that the SQL syntax for `SELECT` has been extended with `ENCLIKE` X `WITHKEY` Y, which will cause the search to be performed securely on the server-side.

The problem with this approach is that it requires the extra overhead of processing every SQL query issued through `libpq`. As well as this inefficiency, it would not be at all modular. As I will discuss in Section 5.3, PostgreSQL's system for adding datatypes and operators is highly modular, allowing them to be packaged and distributed separately to PostgreSQL and then installed into existing PostgreSQL instances. Any modifications to `libpq` would require distribution of the entire `libpq` source, which is tightly coupled with PostgreSQL backend code. This means that if implemented within `libpq`, someone wanting to make use of the secure index scheme would have to recompile their entire PostgreSQL installation.

Because of this, it was decided that modifications to `libpq` were a bad idea and that instead, a *common library* would be implemented. This would contain an implementation of the secure index scheme, as well as all of the other necessary bits and pieces in order to make use of such a system. This library would then be utilised by both the server-side code and any client applications wishing to make use of the system. An example client application was also developed to demonstrate the usage of the library and allow for a quantitative evaluation (Chapter 6). Figure 5.1 shows the key components and some of their responsibilities.

Figure 5.1: A brief system architecture overview showing each component's key responsibilities.

Because PostgreSQL is developed entirely in C, it was logical to develop the encrypted search extensions in C too, in order to avoid incompatibilities between different languages, allowing for easy integration with PostgreSQL. Also, most languages have support for binding to libraries developed in C (using JNI in Java, or natively through .Net languages' interop services), meaning that the common library could easily be utilised by a wide variety of applications if developed in C.

5.1.2 Cryptography Library Choices

A multitude of cryptography libraries exist, supporting a variety of cryptographic algorithms. These come in all sorts of different shapes and sizes, target different languages and are published under different licences. The following list is a brief evaluation of some of these libraries.

Botan [25] is a cryptographic library written in C++, is currently at version 1.8.0 and is released under the BSD licence. It provides support for a large number of algorithms, including Triple-DES, AES (both with support for ECB, CBC and further modes), SHA-1, SHA-256/512, WHIRLPOOL, RSA and a number of stream ciphers. Environments that are supported include Linux, Mac OS X, and Microsoft Windows (all capable of running on a variety of different architectures, including x86, x86-64 and IA-64). The large range of supported algorithms and environments as well as its liberal licence make it a good candidate. Unfortunately, it is written in C++ and PostgreSQL is entirely C, so if it was to be used, a wrapper would have to be developed around it allowing the key functions to be exported to C code.

Crypto++ [26] is an alternative to Botan that also targets C++ applications (currently at version 5.5.2). As such, it is written in C++ and is (mostly) released into the public domain. Similarly to Botan, a large variety of different cryptographic algorithms are supported, including AES (in ECB, CBC and others), Triple-DES, Blowfish, SHA-1 (and other SHA- variants), RSA and ElGamal. It is supported on a number of different systems (including Windows, Linux and Mac OS X) and architectures. Again, if it was to be used, a wrapper would be required to allow usage with C code.

Microsoft CAPI [33] is a cryptography API built into different versions of the Microsoft Windows operating system (the Win32 API). This API supports a number of different cryptographic algorithms, such as AES, Triple-DES, RC5 and SHA-1 (as well as its derivatives, such as SHA-512). As this API is built into Windows, it is not at all portable and the source code is not available. If this was to be used with PostgreSQL, it would limit the secure index implementation to PostgreSQL servers installed on Windows platforms.

GNU Libgcrypt [30] is the GNU basic cryptography library, currently at version 1.4.4. It is written entirely in C and is released under the GNU GPL. A wide variety of different algorithms are supported, including AES, Blowfish, Triple-DES, MD5 and SHA-1 (and other bit length SHAs, like SHA-512). As this library is implemented in C, its integration with PostgreSQL would be far easier than its C++ competitors discussed previously.

Other systems exist, but some aren't as mature as these options, and some offer far *too much* functionality – for example, full implementations of protocols and systems such as SSL and X.509 certificates – that would only prove to add bloat to the system. Of the libraries detailed, the only system that could be easily integrated with the PostgreSQL server-side system is Libgcrypt. It is developed in C and as such wouldn't require any extra wrapper around its functionality. Because of this, and that it supports most of the cryptographic primitives that would be necessary for the implementation of the system, it was chosen to be used.

5.2 Common Library

The common library is the central component and is linked into both the client application and the server-side code. As such, I will describe its implementation first, as the other system components depend on it.

5.2.1 Implementation of Cryptographic Primitives

With Libgcrypt chosen as the cryptography library to be used, its API needed to be extended slightly. The two central cryptographic primitives that are necessary to the implementation of the secure index scheme are a block cipher and a one-way hash. While Libgcrypt does offer a number of implementations of both of these, its API isn't very concise and certain functionality is missing.

By concise, I mean, for example, that there are no top-level functions that take a `char* data` and a `char* password` and produce a `char*` pointing to the ciphertext. There are instead, separate functions for creating a cipher handle, creating a hash handle for hashing the password, initialising the cipher with the key and an Initialisation Vector (IV) and functions that take cipher or hash handles and an input buffer and fill an output buffer. Because of this, the first task was to create two wrappers around the block cipher and one-way hash functionality of Libgcrypt. The functions that were implemented to wrap the Libgcrypt API are shown in Listing 5.1.

Also, certain functionality wasn't available in Libgcrypt, namely the implementation of a padding algorithm. As discussed in Section 4.1.1, a form of padding is necessary when the length of input to a block cipher is not a multiple of the cipher's block size. Because of this, the PKCS#7 [2] padding algorithm was implemented (in blockcipher.c). The `pkcs7_pad` function calculates the necessary number of padding bytes and appends them to the plaintext to bring it up to the required length. Similarly, after decryption, the `pkcs7_remove_pad` function checks the plaintext for padding bytes and these are

removed. The block cipher is used to encrypt the full document contents after the index is created and it uses the Cipher Block Chaining (CBC) mode.

Listing 5.1: An outline of the Libgcrypt wrapper and extension functions.

```
typedef struct
{
    int length;
    char* data;
} encrypted_data, decrypted_data;

/* Encrypts the input data (of the specified length) using
 * the password given. The resulting ciphertext will be
 * stored in output. The function returns 0 if it failed,
 * or non-zero if it succeeded. */
int encrypt_data( char* input,
                  int length,
                  char* password,
                  encrypted_data** output );

/* Decrypts the input data (of the specified length) using
 * the password given. The resulting plaintext will be
 * stored in output. The function returns 0 if it failed,
 * or non-zero if it succeeded. However, it might 'succeed'
 * to decrypt something with an incorrect password. */
int decrypt_data( char* input,
                  int length,
                  char* password,
                  decrypted_data** output );

/* Add and remove padding, respectively. */
static char* pkcs7_pad( char* data, int* length );
static int pkcs7_remove_pad( char* data, int* length, char** output );

/* Initialise a cipher handle by hashing the password and
 * setting the key and IV */
static gcry_cipher_hd_t initialise_block_cipher( char* password );
```

Similarly to the block cipher, a wrapper was created around Libgcrypt's hash API. This resulted in a single function which takes a `char*` and a length and returns a `char*` pointing to the output of the hash. This uses the SHA-512 algorithm and is utilised when hashing the password to generate a key and an IV. Specifically, the password is hashed to produce a 512 bit output. When AES is used as the cipher, the first 256 bits of the hash are used as the key and the subsequent 128 are used for the IV. If the algorithm changes, the cipher initialisation function will use the new algorithm's specification to extract the right number of bits for both the key and the IV.

Since Libgcrypt offers many different algorithms, both the block cipher and the hash algorithm were implemented in a way that would allow changes to the underlying algorithms in use. The block cipher algorithm to use is specified in `blockcipher.h` (currently as `#define BLOCK_CIPHER_ALGO GCRY_CIPHER_AES256`, meaning AES with a 256-bit key is used). The block cipher initialisation function, `initialise_block_cipher(char* password)`, will use the Libgcrypt API to determine the necessary block size and key length when it generates these parameters. This means it can easily be changed to a different algorithm if, for example, the level of security offered by AES-256 is over-the-top for a certain application (although this algorithm is extremely fast anyway).

5.2.2 A Bloom Filter

The key component of the secure index scheme is the Bloom filter [3] (explained in Section 4.2.2). This space efficient data structure allows for fast membership tests and can return false positives, meaning the server, or an eavesdropper, knows even less about the results of the query than they would if no false positives were returned, as they can't be sure which documents (ciphertexts) matched against the trapdoor.

There are a number of implementations of Bloom filters available, but there isn't a large selection to choose from for C. As such, I opted to develop my own. It was implemented in `bloomfilter.c` and offers a number of public functions. When using the Bloom filter, the first function that would typically be called is `bloom_calculate_params`. This function takes the required *false positive rate* and the number of *unique words* it is to store and calculates the number of hash algorithms that the Bloom filter should use and the size of the underlying array in order to maintain the given false positive rate. Listing 5.2 shows this calculation.

Listing 5.2: The Bloom filter parameter calculation function.

```
1 bloom_filter_params* params =
2                     (bloom_filter_params*) allocate(sizeof(bloom_filter_params));
3 params->hash_count = ceil(-(log(false_positive_rate) / log(2)));
4 params->array_length_bytes = ceil(ceil((inflate_factor *
5                                         (double)unique_words *
6                                         (double)params->hash_count)/log(2))/8.0);
```

As well as the maths used to calculate the required array length and the number of hash algorithms necessary (`params->array_length_bytes` and `params->hash_count`, respectively – discussed in Section 4.2.2), there are two things to note about this algorithm, the first being the use of the `allocate` function. This is important because of the way this common library is used in both any client applications and the PostgreSQL backend. As I will explain in Section 5.3.1, PostgreSQL uses its own memory allocation functions (namely `palloc` and `pfree`). As such, it is important that when used in the server environment, all memory allocation (and deallocation) is done through PostgreSQL's own system. Because of this, the `allocate` and `freemem` functions are declared as `extern` in the common library and are expected to be implemented by the application/environment in which it is used (client applications, for example, can simply define them to call `malloc` and `free`).

The second point of note is the `inflate_factor` variable. As I will explain in 5.2.3, there are different ways that the number of unique words that the Bloom filter is expected to be able to hold without straying far from the given false positive rate is calculated. Two of these techniques work best when the Bloom filter's storage array is inflated by a constant factor. Because this isn't related specifically to the implementation of the Bloom filter, and is actually implemented within the *client application*, I will leave it until that discussion to elaborate. It is enough to know that there exists the ability to inflate the storage capacity of the Bloom filter by a constant factor.

Once the filter's parameters are calculated, the second function that would typically be called is `bloom_init`. This takes the calculated parameters returned by `bloom_calculate_params` and returns a pointer to an initialised Bloom filter. The function simply allocates the required amount of memory and initialises the filter's storage array to 0s.

The two access functions are `void bloom_add(bloom_filter* bloom, char* string, int length)` and `int bloom_contains(bloom_filter* bloom, char* string, int length)`, which add a string[2] to the filter and check to see if a string is a member of the filter, respectively. In order to both add and check for membership, these functions need to generate a set of *indexes* into the storage array that are to be set to 1 (in the case of `bloom_add`) or checked (in the case of `bloom_contains`). This is done by the `generate_indexes` function.

[2] That is a string of bytes, *not* necessarily terminated with a null byte.

Defined as `static unsigned int* generate_indexes(char* data, int data_length, int hash_count)`, this function decides on the set of index positions by performing a number of hashes of the input data. As these hashes will be performed thousands of times when encrypting documents of a reasonable length, they need to be very efficient. As such, the standard cryptographic hash algorithms such as SHA-1 are far too slow to be used. Because of this, a set of existing, fast hash algorithms were used. These were taken from [29] and they are published under the Common Public Licence (CPL). Eight algorithms were used in total and they are tied together in an array of function pointers. The `generate_indexes` function uses these hashes on the input word a number of times (determined by the parameters passed to `bloom_init`) and returns the resulting set of hashes, modulo the length of the filter's storage array. `bloom_add` then simply sets the bits at these indexes to 1 and `bloom_contains` only returns 1 (true) if the bits at these indexes are all 1s.

5.2.3 Secure Index Implementation

With the Bloom filter implemented, the core algorithms involved in the secure index scheme could be implemented. These are the `goh_trapdoor` and `goh_codeword` functions, implemented in `goh.c`. Figure 5.2 shows the process through which the trapdoor (T_w) and the codeword (C_w) are calculated (more details were given in Section 4.2.2).

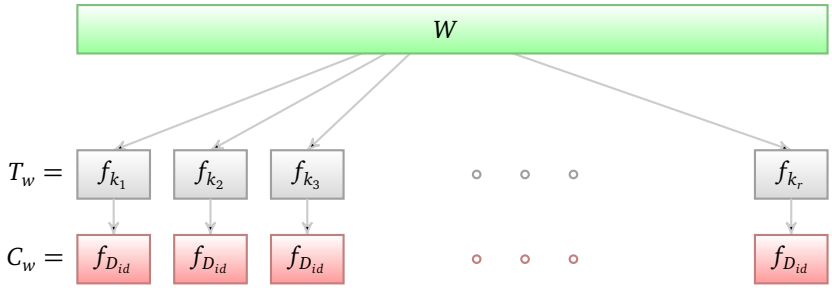

Figure 5.2: The generation of the trapdoor and codeword from a given input word W, using a key $k = (k_1 ... k_r)$, for document with ID D_{id}

As this diagram illustrates, the trapdoor and codeword for word W are defined as

$$T_w = (f_{k_1}(W), \ldots, f_{k_r}(W))$$
$$C_w = (f_{D_{id}}(T_w \downarrow_1), \ldots, f_{D_{id}}(T_w \downarrow_r))$$

The secure index scheme specifies *how* the algorithms work, but doesn't pinpoint specific implementation details, such as the bit-lengths of the individual components. The implementation of f, along with the way the key is split into $k_{\{1...r\}}$ and how it is combined with the data to be hashed is left up to the implementer. On of the most important choices is the length of f's output, which will determine the length of the trapdoor, which will need to be transmitted over the network. Too small and this will generate a large number of false positives by increasing the number of collisions, too large and it will add more network overhead. Similarly for the codeword; the specifics of how the document ID is combined with the element of the trapdoor to be hashed are not defined.

For the *trapdoor* implementation, the key k was chosen to be 512 bits in length. This is quite easy to produce using an existing 512 bit hash algorithm. SHA-512 was used for this purpose. With the 512 bit key (64 bytes), the individual elements of $(k_1, \ldots, k_r)$ were chosen to be 32 bits in length (4 bytes, meaning $r = 16$). The output of f is kept to 2 bytes, resulting in a 32 byte trapdoor. The key data is combined with the input word through concatenation (the key block is appended to the word, prior to it being hashed).

For the *codeword* implementation, the same hash algorithm, f, was used. The document ID is combined with the trapdoor block through concatenation (the trapdoor block is appended to the document ID prior to it being hashed).

Both of these algorithms use the same underlying hash algorithm, f. Since this operation will be performed a large number of times when encrypting a document, performance is important. Attempts to used a cryptographically secure algorithm such as SHA-1 proved to be far too slow. Even algorithms that result in a far smaller output, such as a CRC-32 with a 4 byte output were still too slow. Because of this, it was decided to use a hash algorithm from the Bloom filter set (the FNVHash algorithm, to be precise). If this algorithm proves to not give a normal distribution of output for a random set of input, then this could be used to derive patterns and could potentially be used as an attack vector on the system. Ideally a secure algorithm should be used, but the performance penalties in practice are far too drastic.

Document Index Generation

With the trapdoor and codeword algorithms in place, the next step is to generate a document index for a given plaintext. This goal is achieved by the `goh_generate_index` function.

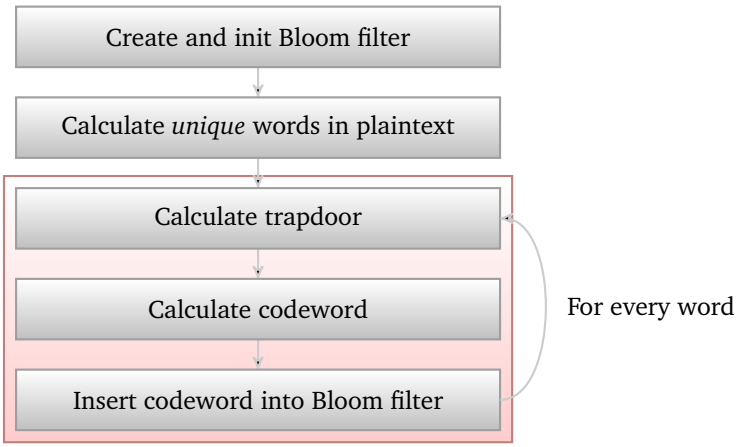

Figure 5.3: Algorithm for creating a document index.

The first step simply creates and initialises a new Bloom filter to represent the document's index. After that, the set of unique "words" in the document needs to be calculated. This calculation depends on what is to be considered a 'word', which can vary depending on the different search scheme. As stated in Section 1.3, there were four different search types implemented, each with a different definition of a 'word.'

Exact-match. This search option uses the standard definition – words in a document are extracted by splitting the document on a pre-determined set of punctuation characters (defined by `DOCUMENT_WORD_DELIMTERS` in `goh.c`). The resulting chunks are used as words and their codewords are calculated and inserted into the Bloom filter index.

Case insensitive. With this search option, the same technique as exact-match is used to generate the document index, except that before the codeword for each word is calculated, it is transformed into its lowercase equivalent. Its codeword is then inserted into the filter. When a search is performed, the search term must also be transformed into lowercase before the search takes place.

Left-most match. This option allows the searcher to use wildcards on the right-hand-side of their search term. This means that one could search for *"success"* and documents containing *"successfully"* would also be returned as positive matches. This is achieved by inserting a number of entries into the document index for each word, as determined by the exact-match approach. Specifically, each substring that starts from the beginning of the word to be inserted is also inserted, with an asterisk appended. For example, when generating the index, if the word "successfully" is to be inserted, then the words "s*", "su*", "suc*", "succ*", etc, will all be added. When a search is performed, a search for a wild-carded term like "suc*" will be found in the index.

'Natural language' search (word stemming). This option makes use of Porter's stemming algorithm [14] to break words (as derived from the document through the exact-match approach) into their base 'lexemes'. These are terms which form the basis of words – for example, "successfully" and "successful" will both be transformed into the lexeme "success". This process is performed on each word prior to its insertion into the document index. Also, terms that are known as 'stop words' are removed. These are words which don't serve to add meaning to a sentence, but are merely there to form a complete phrase. These include words like "the," "a," and "because" (the complete list of stop words used in the implementation is available at [32]). When the document index is created, these words are ignored. When a search is performed, the search term is transformed into the base lexeme and this is used to perform the search. If the search term is a stop word, an error message should be shown by the client application prior to searching. This approach is also a superset of the case insensitive option, as all lexemes are lowercase. It is also highly language dependant. This implementation uses stop words and stemming rules that are specific to the *English* language, although there are similar algorithms for other languages that could easily be used instead.

Each of these approaches produces different size indexes. This has an effect on the efficiency of searches and the encryption process. Left-most matches inflate the document indexes by quite a lot, whilst the word stemming approach actually reduces the index size, as it removes unnecessary 'stop words'. The differences in the performance of these options will be discussed in depth in Chapter 6.

Options to enable these different search types are made available by the common library's API. As they each build different document indexes and expect the search queries to be presented in different ways, they cannot be used interchangeably. That is, in order to perform a case insensitive search of a document set, that document set must have been encrypted with the specific case insensitive option, and if this is the case then a case *sensitive* exact-match search is not possible. It would be possible to have multiple indexes for each document, one for each search scheme, allowing different search types on the same dataset, but this would require a considerable amount of potentially unnecessary space. The different choice are also application specific, so the implementation of a generic system that covers all bases adds too much unnecessary overhead.

The algorithm to extract unique words from a plaintext is defined in `unique_words.c`. This walks over the plaintext and inserts each word it finds into a hashtable, unless it's already stored in the hashtable. The 'word' inserted is dependant on the search options as discussed previously. The left-match approach may result in multiple 'words' being inserted at each iteration for example. Once this hashtable is fully populated, the words it contains are inserted one by one into the document index (after being transformed into their codewords).

As well as the different search options, there are a number of ways in which the *sizes* of the indexes in a document set can be calculated. The original paper [9] specifies using the total number of (unique) words in the *entire* document set. This value is then passed to the `bloom_calculate_params` function to determine the size of the Bloom filter necessary and the number of hashes. These parameters are then used for *all* of the documents in the document set. In practice, this approach is extremely slow and space-inefficient (this will be made clear in Chapter 6). As such, a number of other techniques for calculating document index sizes were experimented with. Because of this, the common library API takes an integer argument to its `goh_encrypt_document` function (used by client applications to encrypt a document) which defines the number of "unique" words the filter that gets created should be expected to hold. The different ways of calculating this value are implemented in the client application and as such are discussed in Section 5.4.

Compression

Document indexes (depending on the false positive rate) can be sparsely populated, resulting in a block of memory that is amenable to standard compression techniques. The common library was extended with the ability to perform compression and decompression of document indexes during the encryption (after the index is created) and search processes (decompression of each document index prior to searching), respectively, in order to experiment with the advantages this offers. The compression algorithm used is FastLZ [27], an LZW variant, which is implemented in C and distributed under the MIT licence. Details on the performance impact and the amount of compression achieved can be found in Chapter 6.

Client API

The main client API offers three functions which are detailed in Listing 5.3. These functions are intended for use with a client application and should be the only calls necessary to the common library in order to support a secure index scheme. However, the implementation of the PostgreSQL backend datatype and operator described in Section 5.3 make direct use of some more of the internals of the common library in order to operate.

Listing 5.3: The primary client API functions.

```
/* Encrypts the given document contents using Goh's secure index scheme.
 * The key is expected to be something akin to a password (i.e. it doesn't
 * have to be of fixed length). The memory returned will be consistent
 * with the server-side document datatype. */
char* goh_encrypt_document(char* document_id,
                           int document_id_length,
                           char* document_contents,
                           char* key,
                           int* out_length,
                           int unique_word_count,
                           int left_match_support,
                           int case_insensitive);

/* Creates a binary trapdoor for the given search term/key combination.
 * The search_term parameter might be modified. */
char* prepare_query_trapdoor(char** search_term,
                             char* key,
                             int left_match_support,
                             int case_insensitive);

/* Confirms that a result does in fact contain the search term. */
int confirm_result(char* document_contents,
                   char* search_term,
                   int left_match_support,
                   int case_insensitive);
```

5.3 A Server-side Datatype and Operator

PostgreSQL offers a modular system for extensions, allowing developers to create their own custom datatypes and operators which can then be installed into existing PostgreSQL servers without the need to recompile its source [41]. This is achieved by the PostgreSQL catalogue system, which stores information about tables and columns as well as information about datatypes and operators. The API specifies that new 'base types' are developed in a language compatible with C, compiled into a dynamically linkable library and are loaded at runtime by the server when necessary. There are other sorts of types which can be created, but only 'base types' are implemented in C and let developers specify their own input/output functions, as well as their in-memory layout.

The secure index scheme required both a new datatype and an operator. The datatype represents a document that has been encrypted with the secure index scheme. This means that it contains both the document's ciphertext and the Bloom filter representing the document index. The binary operator takes one of these documents and a binary string representing a trapdoor. It returns true if and only if the trapdoor is found in the document's index.

5.3.1 PostgreSQL Internals

There are two main types of datatype that can be created in PostgreSQL: fixed and variable length. A fixed length type has a known internal length and can be represented by C `struct`. A variable length type allows different instances to be of different lengths. This was necessary when developing the document datatype, as different documents and indexes would vary in length. Variable length datatypes are still defined as C `structs`, but they all start with a four byte block which is used to determine their length at runtime. They also make use of a commonly used C 'hack', where an array is used as the last element of the struct, but it is declared as one element in length. When creating an instance of this struct, more memory than `sizeof` specifies is allocated, in order to allow enough room for the particular data that is to be stored. This structure can then be accessed past the 'end,' thanks the C's lack of bounds checking. This technique is necessary so that the a datatype's memory is contiguous, allowing PostgreSQL to move it around in a single block. To do this, PostgreSQL inspects the initial four bytes to determine its actual length when it needs to move it in memory or persist it to disk.

As well as datatypes being of variable length, they can also be TOASTable. TOAST (The Oversized-Attribute Storage Technique) [46] is used to store data that is too large to fit within a single PostgreSQL 'page'. It allows variable length datatypes to be split into separate blocks which are then stored separately to the table in which they are referenced. It can also apply compression to the data. Because documents that are being encrypted are likely to exceed a single page size (which is very small, at roughly 8kB for a default installation), the document datatype was made TOASTable.

As previously mentioned, PostgreSQL uses its own memory allocation functions – `palloc` and `pfree`. Calls to these functions instead of the standard glibc `malloc` and `free` allow PostgreSQL to maintain a list of memory allocations that were made during each transaction. After a transaction ends, it will check that these blocks of memory had been freed and if they haven't, it will automatically free them, helping to prevent memory leaks.

5.3.2 Implementation

The datatype and operator are both implemented in a separate C library which depends on the common library. In order to be loaded by the PostgreSQL server, this library is built as a shared object. Along with this library, a file containing a set of SQL statements is used to initialise a database with the necessary information to be able to utilise the it.

Datatype

The datatype that represents documents encrypted with the secure index scheme is depicted in Figure 5.4. This type is a variable length base type, so it defines a four byte header which stores its runtime length. It also maintains the byte lengths of the document ID (a `char*` used to identify the document by name), the length of the Bloom filter and the length of the document ciphertext.

The use of `int`s for the lengths gives both the index and ciphertext maximum lengths of just under 2GB, however the 4 byte datatype header restricts this further, to 2GB for the entire structure. The common library's client API supports generation of instances of this data structure through its `goh_encrypt_document` function.

The main components of a PostgreSQL datatype are its input/output functions. For the document datatype, there are four of these functions:

`goh_document_in` takes an *escaped* string and converts it into an instance of the document datatype. This function is used by the client application `psql` to convert a string into a datatype which can then be saved and manipulated by the PostgreSQL backend. An SQL command such as "`INSERT 'data'::gohdocument INTO mytable`" will cause this function to be invoked, causing the *data* string to be converted into a `gohdocument` (the SQL name for the document datatype). This conversion is caused by the cast operator – '::' – but might be performed implicitly in some circumstances in which the type is unambiguous. The binary-to-ASCII conversion routines used in

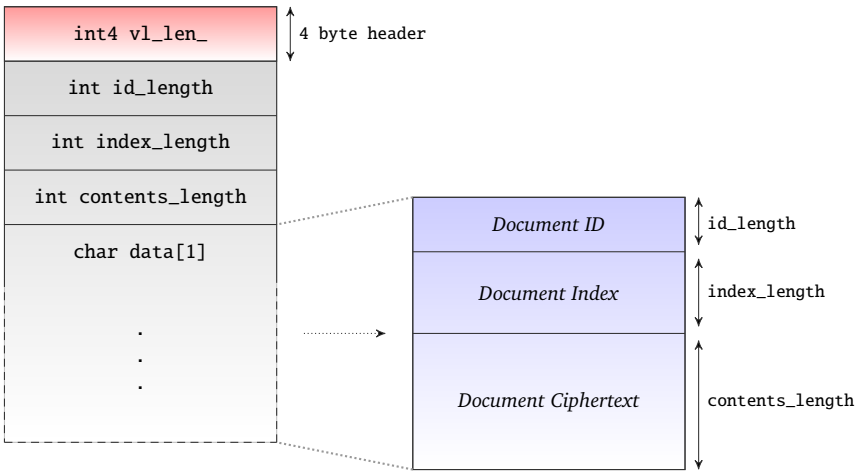

Figure 5.4: In-memory layout of the document datatype.

both this function and `goh_document_out` are defined in the PostgreSQL backend source and have been reused.

`goh_document_out` takes a pointer to an existing document in memory and converts it into an *escaped* string. This function is used by the client application `psql`, as well as the `pg_dump` application, when they want to convert a memory representation into a string which can be presented to the user, or written out to a backup file, respectively.

`goh_document_send` takes a pointer to an existing document in memory and pushes it into a buffer. This function is used when returning results to a client application, through `libpq`. This function doesn't have to perform much conversion of data (other than the standard host-to-network byte ordering), so it is fast.

`goh_document_recv` takes a buffer, as constructed by the `goh_document_send` function, and converts its contents into an instance of the document datatype. This function is used by client applications through calls to `libpq`'s `PQexecParams` function. This allows binary streaming and so the `recv` function is used over the text-converting `in` function by the client application for efficiency, as little conversion is needed.

These functions are all implemented within the server-side library and are each defined to take a `PG_FUNCTION_ARGS` and return a `Datum`. These are PostgreSQL constructs and are part of the 'version 1 calling conventions'. The individual arguments are all wrapped in a `PG_FUNCTION_ARGS` and they are extracted through a number of macros. These macros help to encapsulate the different datatypes, performing modifications such as "de-TOASTing," amongst other things. This is necessary on TOASTed datatypes and causes the split-up memory which is stored in the TOAST table to be converted into a standard, contiguous block.

Each of these input/output functions is then tied together into a single SQL datatype using a number of SQL commands, one of which is shown in Listing 5.4.

The use of the value `VARIABLE` for the type's `internallength` indicates that it should conform to the 4-byte-header system and the use of `extended` for the `storage` mechanism forces the datatype to be TOASTed (rudimentary compression is also used with this option).

Operator

The operator takes an instance of the document datatype, along with a block of binary data (known as a `bytea` in PostgreSQL terminology) that represents a trapdoor and returns a Boolean representing the

Listing 5.4: Part of the SQL needed to create the document datatype.

```
CREATE TYPE gohdocument (
    internallength = VARIABLE,
    input          = goh_document_in,
    output         = goh_document_out,
    send           = goh_document_send,
    receive        = goh_document_recv,
    storage        = extended
);
```

result of the search. The operator is defined as a single function, `goh_document_search`. This uses the same version 1 calling conventions as the datatype, wrapping the arguments in a `PG_FUNCTION_ARGS`. The arguments that are passed to this function are the operands (the document and the trapdoor). This function then simply accesses the document's index and uses the common library to create a codeword from the trapdoor and perform a search of the index. If the index shows the codeword as a member, the operator returns true. For the SQL side, the operator was defined (using the SQL as shown in Listing 5.5) as the symbols '<=', so that queries such as

```
SELECT * FROM myTable WHERE myEncryptedColumn <= 'trapdoor'
```

can be used to search a column in a table. The result set of this query can still contain false positives, which must be filtered out by the client application manually.

Listing 5.5: SQL command for creating the operator.

```
CREATE OPERATOR <= (
    leftarg = gohdocument,
    rightarg = bytea,
    procedure = goh_document_search,
    commutator = <=
);
```

5.4 A Client Application

An example (CLI) client application was developed in C to allow for benchmarking as well as acting as a demonstration of how to use the common library. Essentially, it is an interface to `libpq`, along with a number of calls to the common library's client API in order to encrypt documents and also to prepare a search query (generate trapdoors). It is build with the ability to batch encrypt a given directory and upload it to a PostgreSQL server (with the secure index system installed), as well as being able to search this server and filter the results for false positives. As stated in Section 5.2.3, this client application also allows for a number of different *index size* calculations.

5.4.1 Index Sizes

In Goh's original paper [9], the suggested method for generating an index involves calculating the total number of words in the *entire* document set. This number is then used to derive the size for every document in that set. The advantages of this technique are twofold:

- The number of false positives returned should be held at the rate required
- Each document index is the same size. If they are of different sizes, information is leaked to the server about the number of words the document contains, relative to the other documents.

After implementing the system in this way, it became apparent that it wasn't particularly efficient (detailed in Chapter 6). This is because it requires an initial scan of the document set in order to calculate the total number of unique words (this can't be done at the same time as the encryption and upload of documents, as the total unique word count is required before anything can be encrypted) and also is very space-inefficient, creating large indexes for all documents. Because of this, a number of other approaches were experimented with.

Largest file. This technique scans the document set for the largest number of unique words in any document. This value is then used to create the document indexes for all documents in the set. This keeps the index sizes the same for all documents and as such leaks no information that way. It does, however, increase the false positive rate due to the smaller indexes. But, as I will show in Chapter 6, it is far more computationally efficient to filter through a large number of false positives than it is to search through very large indexes. The increased amount of false positives also help to confuse the server, leaving it unsure of which documents returned actually contained the trapdoor.

Average unique word count. This approach scans the document set and calculates the average number of words in every document. This value is then used to compute the index sizes for all documents. Because a number of files will contain *more* unique words than the average of all documents, this value has to be inflated by a constant factor before it is used (as mentioned in Section 5.2.2). This technique generates same-size indexes for all documents, but similarly to the largest document approach, does increase the number of false positives returned.

Per-document index calculation. This option uses the unique word count of a document to calculate that document's index size. This approach results in different index sizes for each document, meaning a certain amount of information leaked to the server, however it is faster (to both encrypt and search) than the others as it doesn't require an initial pre-scan of the document set. The amount of information leaked to the server by the index sizes is also debatable, as the ciphertext stored for each document could be used to determine the potential length of each document, which would allow for a reasonable guess as to its unique word count anyway.

All of these options are made available by the client application through different command line switches (extracted internally through the `getopt` API). The different search options will also affect the index sizes, with support for left-match searches (using any of the above size calculations), for example, resulting in larger indexes than exact-match (using the same index size calculation from above).

False positives are eliminated by the client application after the result set is returned by the server. To do this, the ciphertext of each document is decrypted and the resulting plaintext is inspected. If the search used is a standard exact-match or a left-match, then the search term is known to exist in its entirety in the plaintext and so, the C `strstr` function is used – if this function's result is non-null then the document is a successful match. In the case of case insensitive and natural language search, `strstr` cannot be used, as the search term might not be contained entirely in the document's plaintext. As such, in these cases the document is walked over and each individual word (separated through the use of `strtok`) is converted into the form in which the search term is represented – it is converted to lower case, and possibly also stemmed. This is then compared against the search term in order to determine a match.

As well as being able to make use of the searchable encryption scheme, this application is also capable of uploading plaintext to the server, in order to benchmark results against a plaintext search (it can also be used to create an index on a plaintext column). It can also be built with support for index compression, as described in Section 5.2.3, as well as the different search options (case insensitive, left-match and word stemming) described in Section 5.2.3.

5.5 Build Environment

The build system allows compilation of all the components, as well as linking against the correct PostgreSQL installation. There exists a system for PostgreSQL that allows external modules to be built and installed smoothly (called PGXS [39]), but this wasn't used, as it offered too much functionality, the majority of which was unnecessary and would just require further work for no gain. The build system expects an environment variable, `PGSQL_INSTALL` to point to the location on the system where the PostgreSQL server installation lives. This is then used to link against, as well as installing the server-side datatype and operator's `.so` (shared object) file into.

Because the server-side library has to be built as a .so file, the common library was *statically* linked with it. This means that the common library is compiled into object files, which are then archived using the `ar` program, to produce a `.a` file. This file is then embedded within the server-side `.so` file, allowing for a single installation target.

The common library depends on Libgcrypt, which is expected to be installed on the target system. The Libgcrypt library is dynamically linked at runtime by both the server-side code and the client application.

5.6 Summary

In this chapter, the implementation of a secure, pre-processed index approach to searching encrypted data was detailed. This consists of a *portable* common library, which contains the core components (a Bloom filter implementation, the necessary cryptographic primitives and an implementation of Goh's indexing scheme) necessary to make use of this system. Since this implementation is modular in its design, it could quite easily be reused if attempts were made to add support for a searchable encryption scheme to another application. It also supports four different search types, detailed in Section 5.2.3.

Along with this library, the PostgreSQL database management system was taken as the application to extend. A datatype and an operator were implemented, allowing a column in a table to contain ciphertext that is amenable to search and to compare this ciphertext against a trapdoor in order to determine a match, respectively.

A CLI client application was also developed that supports interaction with a PostgreSQL server with this secure search scheme installed. This client is capable of batch-encrypting a given directory and uploading the ciphertexts to the PostgreSQL server. This database can then be searched by issuing queries through the client application. The results are filtered for false positives before the list of matching documents are presented to the user. Using this application, a number of different approaches to index size calculation were experimented with. In Chapter 6, I will show that these different options result in a notable increase in performance over the original system, as suggested by Goh in his paper [9].

6 Evaluation

In this chapter, I will evaluate the success of the project. This can be measured using the performance of the implementation, both in terms of time (for encryption and searches) and space overheads.

6.1 Corpus Selection

In order to test the system, a corpus (or corpora) had to be chosen. The system could then be used to encrypt and search over this dataset in order to quantitatively evaluate the implementation. Choosing a suitable corpus is important. A number of properties should be considered. The *size on disk* should be reasonable, capable of being stored in less than a few gigabytes of space in order to keep testing times down, but it should be able to reproduce the sort of dataset size that would be expected in a real-world environment, causing the search scheme implementation to be thoroughly tested. The *word variety* contained within the corpus should be considered; the possible number of *results* from queries must fall within a wide range, from as low as 1 result, to the majority of documents being returned. The set of test queries used to evaluate the system can then incorporate these edge cases which will test the memory usage of the system and other factors when the result count is large. Also, the *number of individual documents* should be considered. This number should be fairly large (greater than 1,000 for example) in order to mimic a real world situation. The number of documents will also depend on the size of each individual document (as previously described).

A number of choices for corpora exist, each with different values for the properties outlined above.

The Enron Corpus [5] is comprised of a collection of emails that were made public after the Enron Corporation was investigated because of suspected fraud. This corpus consists of *200,399* email messages from *158* different users and totals *400MB* in compressed form.

The RFCs [47] is a database of Request For Comments (RFCs). At the time of writing, the RFC database contains *5,407* (5,371 in plain text) entries and totals *277MB*. This corpus contains a large number of technical terms, a lot of which are unique to the document in which they are discussed. This will allow for a large range of possible results from search queries.

The Collected Works of Shakespeare [57, 58] is the complete collection of William Shakespeare's work. The plain text version (available at [57]) is particularly small in size (at *5.03MB*) and contains 42 documents. However, one of the advantages of this corpus is that it contains a wide variety of different words, allowing for queries that will return a small number of results.

A USENET corpus (2005-2008) [60] was built from public newsgroup postings between 2005 and 2008. This corpus is by far the largest discussed, reaching *22GB compressed* and containing in excess of *18 billion words*. The contained documents are between 500 and 500,000 words in length. A dataset of this size might well be too large, but it would certainly stress test the modified application. The documents also each contain MIME headers, which would have to be removed or else the results would be affected by this added 'noise'.

The set of RFCs was chosen as the corpus to use to evaluate the system. This is of usable size, compared to something like the USENET corpus which is far too large to realistically test with, and the works of Shakespeare which are far too small. It provides a wide range of possible queries due to its technical content and contains a large number of separate documents.

6.2 Evaluation Decisions

There are a number of different options available for use in the system. These include the way in which the size of document indexes are calculated (the implementation of which was discussed in Section 5.4.1), the usage of compression on the indexes (described in Section 5.2.3) as well as the different search types that can be performed (discussed in Section 5.2.3). Each of these affect both the space and time overheads of encryption and the time necessary for searches. The time and space requirements for the encryption process are evaluated in Section 6.3 and requirements for search are evaluated in Section 6.4.

These tests were conducted on a machine with an Intel Core2 Duo CPU (E6550) running at 2.33GHz, 2GB of RAM and a 7200rpm disk drive. The operating system used was 32-bit Ubuntu 8.10 with kernel version `2.6.27-14-generic`. Both the client and server application were run on the same machine, as the network communication overhead was assumed to be negligible. The system is configured to have a false positive rate of 10% and it uses the AES cipher (with a 256-bit key) to encrypt the document contents.

6.3 Encryption

In this section, the overheads associated with the encryption process will be detailed. Firstly, the *time* (or the *speed*) will be considered, before the *space* overheads are looked at. The system (and the different index size calculation options) will be evaluated both with and without compression enabled, followed by looking at the different search options and their associated costs. As previously stated, the corpus used was the set of all 5371 RFCs [48] (277MB), split into 20%, 40%, 60%, 80% and the full data set, in order to measure the various options against varying dataset sizes.

6.3.1 Time

Figure 6.1 shows that the original specification of the scheme (document index sizes are calculated by taking the total number of unique words in the entire document set) takes considerably more time compared to the other available choices (using the largest document, for example). This original specification has a data rate of roughly 1.37MB/s, compared to using the *average* number of unique words in every document to calculate the document index size, which has a data rate of around 2.21MB/s.

Figure 6.2 shows the same encryption process on the same data, but with index compression enabled. The time taken to encrypt using the total word count algorithm is reduced (bringing the data rate up to 2.18MB/s – an increase of 0.81MB/s), which clearly illustrates the fact that a large amount of the overhead with the encryption and upload process is caused by the sheer size of the documents. The compression algorithm used (detailed in Section 5.2.3) was chosen for its speed, so it adds little overhead. The speed of the encryption process is actually increased by the use of compression for all of the different options (the average count algorithm for example, increases 0.01MB/s to a data rate of 2.22MB/s), although the original scheme is affected the most.

Figure 6.3 shows the overhead added by the different search options. Quite clearly, encrypting the dataset with support for left-most matches increases the time taken dramatically. This is to be expected, as the support for left-most matches is achieved through inserting all left-most substrings of every word into the document index. The other search options take roughly the same amount of time, with word stemming coming out fastest. This is because of the stop-word removal process, which will filter out words, resulting in less to hash and therefore smaller document indexes.

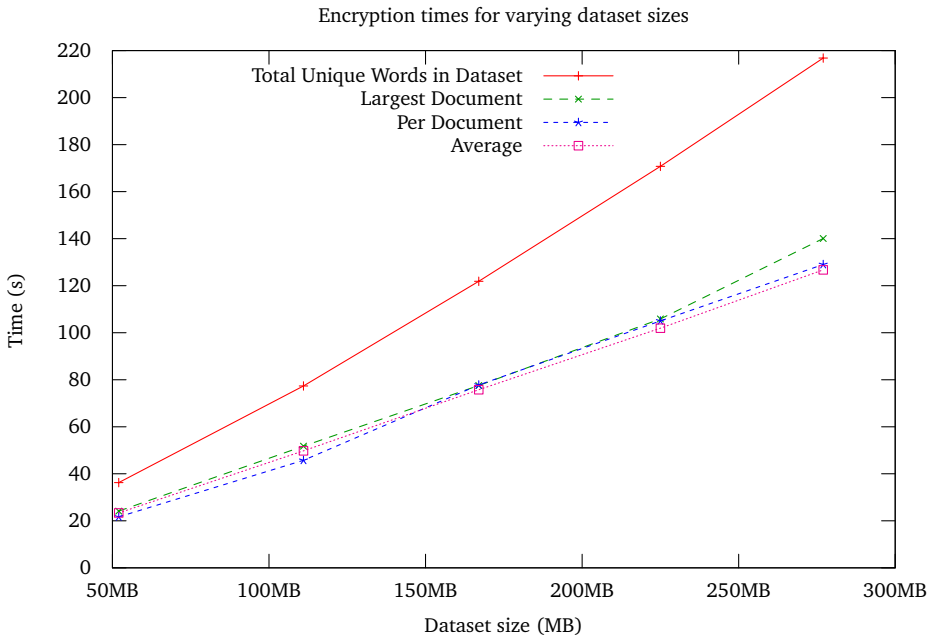

Figure 6.1: Graph showing encryption times against corpus size with compression *disabled*.

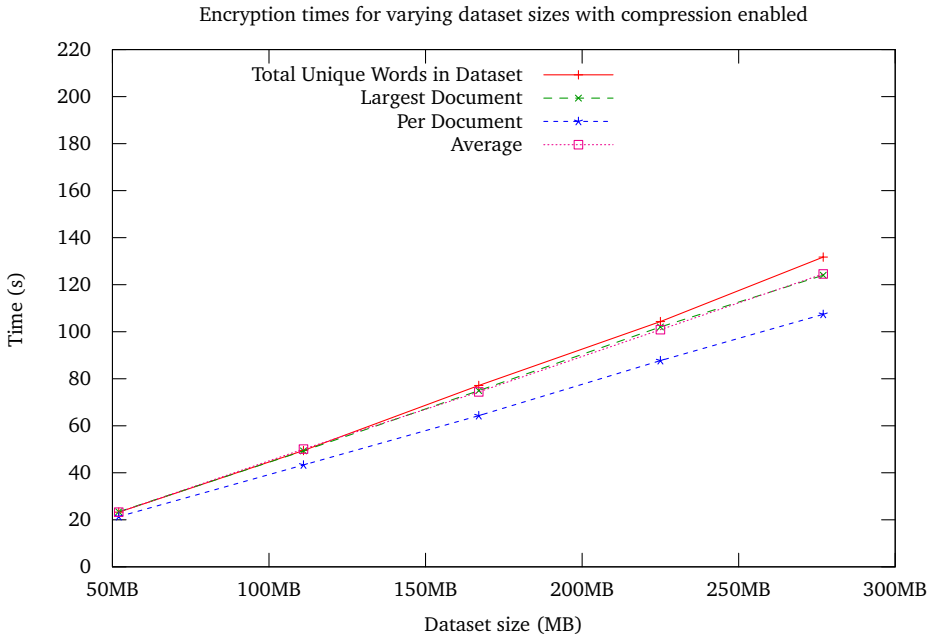

Figure 6.2: Graph showing encryption times against corpus size with compression *enabled*.

The choice of index size calculation used for Figure 6.3 was per-document, meaning the document index size varies from one document to another. This was chosen as it removes the overhead added by the other search options, allowing for a clean analysis of just the differences in search options.

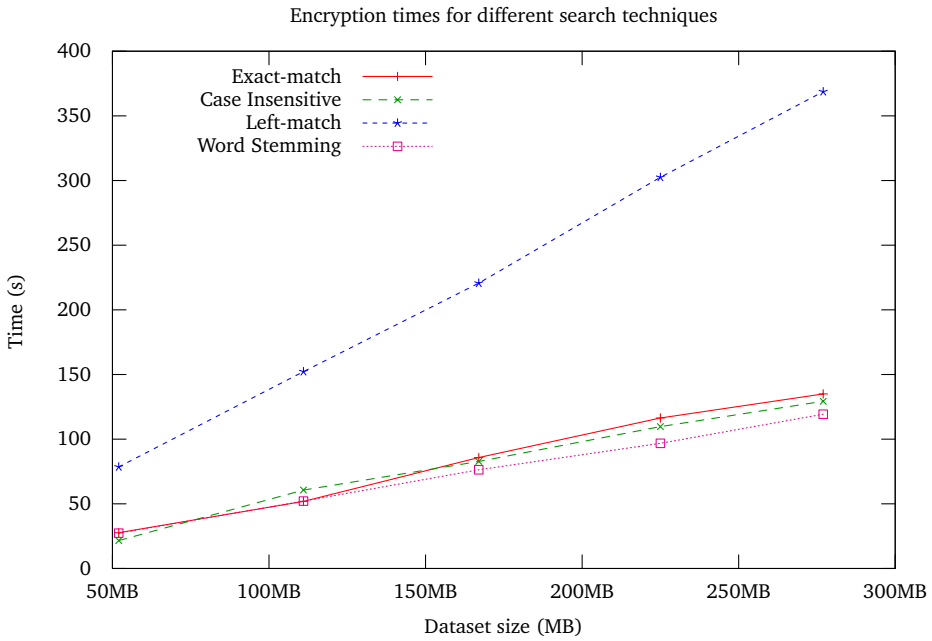

Figure 6.3: Graph showing encryption times for different search techniques (compression *disabled* and index sizes are calculated using the *per-document* option).

6.3.2 Space

The following figures were calculated using the space *on disk* of the PostgreSQL data directory. When the server is initialised, a directory for storing data is specified. This is then used by the server to store tables, relations and everything else specific to the database. This is stored in PostgreSQL's own binary format, also containing log messages and other details. Another way of measuring the space required is to use the `pg_dump` application to dump the database to an ASCII text file. This removes the overhead added by the internal log messages, etc, but it also requires binary data to be converted to ASCII (done by calling the `goh_document_out` function of the document datatype). The specific algorithm used in this conversion is internal to PostgreSQL. It converts all bytes that fall outside the normal ASCII range into an escaped octal representation (for example, the byte represented by decimal '04' would be converted into the string "`\004`"). Clearly this technique is not the most efficient technique possible (at worst resulting in an ASCII representation that is four times larger than the equivalent binary), with lots of other systems for binary to text conversion existing (such as yEnc or uuencode) which achieve better conversion ratios, but it didn't seem necessary to invest time in implementing one of these systems, simply to reduce the space required for a database dump. As such, evaluation results taken from `pg_dump` output are heavily inflated and so have not been used.

Figures 6.4 and 6.5 show the space overhead of the encryption algorithm. Clearly the compression algorithm has little effect on the overall size of the encrypted dataset. This is both because the indexes are overshadowed in size by the document ciphertexts, as well as the fact that PostgreSQL performs its own rudimentary compression on TOASTed datatypes. The original (total word count) scheme has a plaintext to ciphertext space ratio (uncompressed) of around 2.3 and the average word count scheme has a ratio of 2.1 – however these ratios decreases as the dataset size increases.

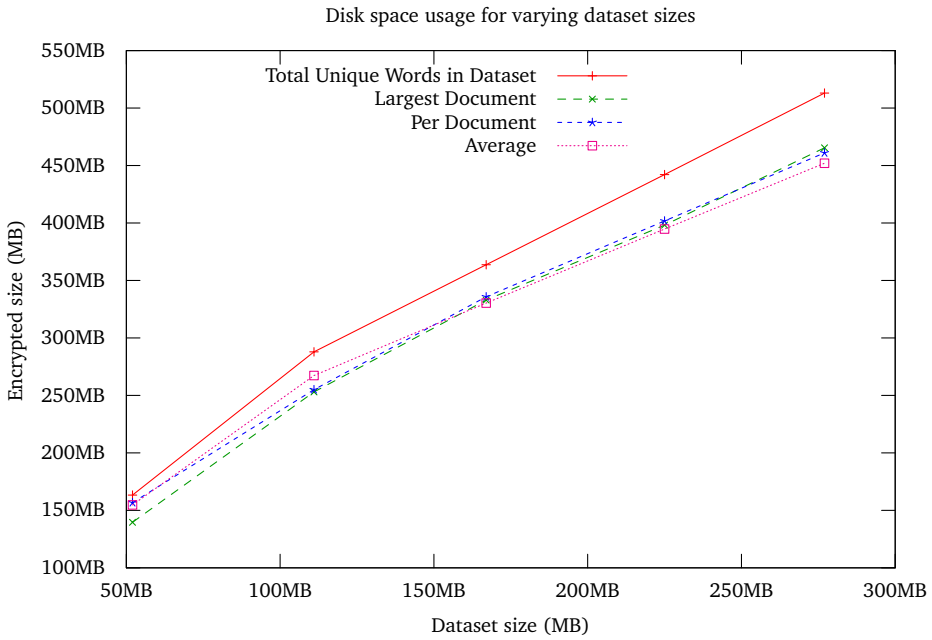

Figure 6.4: Graph showing disk space usage against corpus size with compression *disabled*.

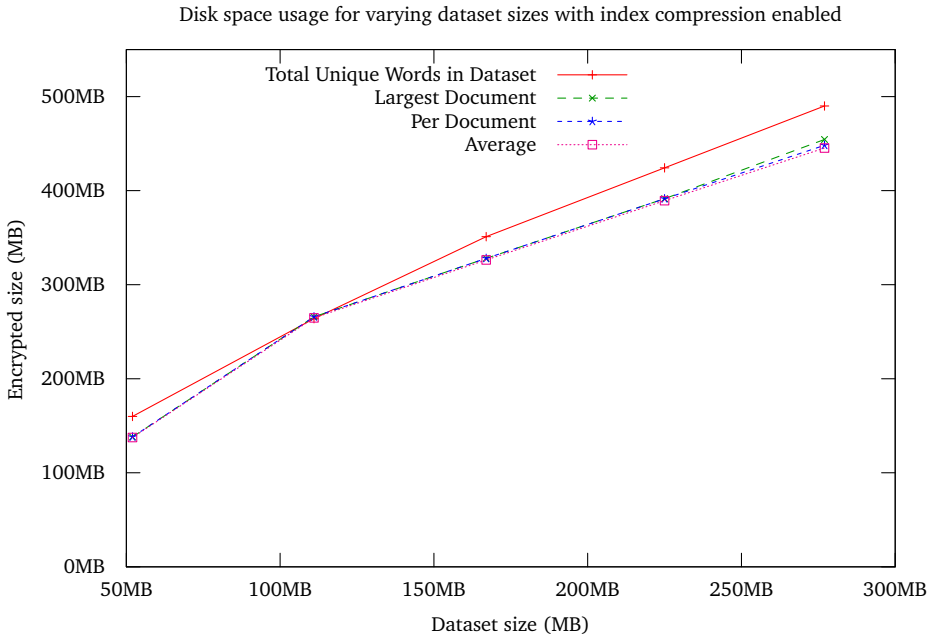

Figure 6.5: Graph showing disk space usage against corpus size with compression *enabled*.

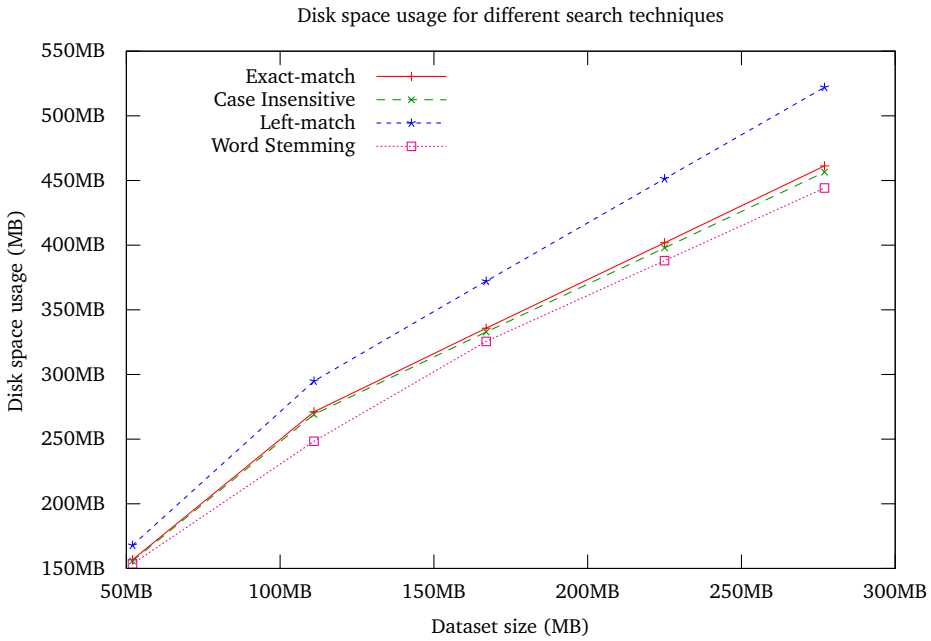

Figure 6.6: Graph showing disk space used for different search techniques (compression *disabled* and index sizes are calculated using the *per-document* option).

Figure 6.6 shows the space used for different search options. Similarly to the time taken to encrypt, the left-most match support adds the greatest amount of overhead, although not by as greater margin as with time. Unsurprisingly, the word stemming approach comes out with the smallest disk footprint due to its removal of stop words, but since the dataset used (the RFCs) contains a large amount of technical terminology, the amount of words found to be stop words isn't that large so the reduction is only small. Used on a different dataset, one would expect to see greater separation of this option from the others, in terms of both space and time overheads.

6.4 Search

The secure index scheme [9] has (in theory) a constant-time search in the length of documents (linear in the *number* of documents). This means that searches for different terms should take the same amount of time to complete. As such, a single search term was used to calculate the times shown in Figures 6.7 and 6.8 (although multiple searches were performed for each dataset size in order to calculate an average). The specific search term used was "Khanna" – the surname of an author of a number of RFCs found in the collection.

It should be noted that these figures are all taken from *warm-starts*. That is, a query has previously been run to cause the database to be loaded into memory. This is standard practice for large databases, which are typically kept entirely in memory, unless they become too large. Plaintext searches of a database require the same process – cold starts are far slower than warm (taking about 20 seconds to load the entire 277MB database into memory). Because these cold-starts don't tell us anything about the efficiency of the system, they have been ignored.

In Figure 6.7, we can see that the use of the total number of words in the dataset to calculate the index sizes (as proposed by [9]) is far *slower* to search over than the other techniques. The datarate of the search process on the full dataset with indexes calculated with the *average* word count is 281.8MB/s. This is compared to what is achieved by the search process on the dataset created using the *total word count* – a datarate of 70.8MB/s – nearly four times slower. The speed of search averages to around 300MB/s for the different dataset sizes.

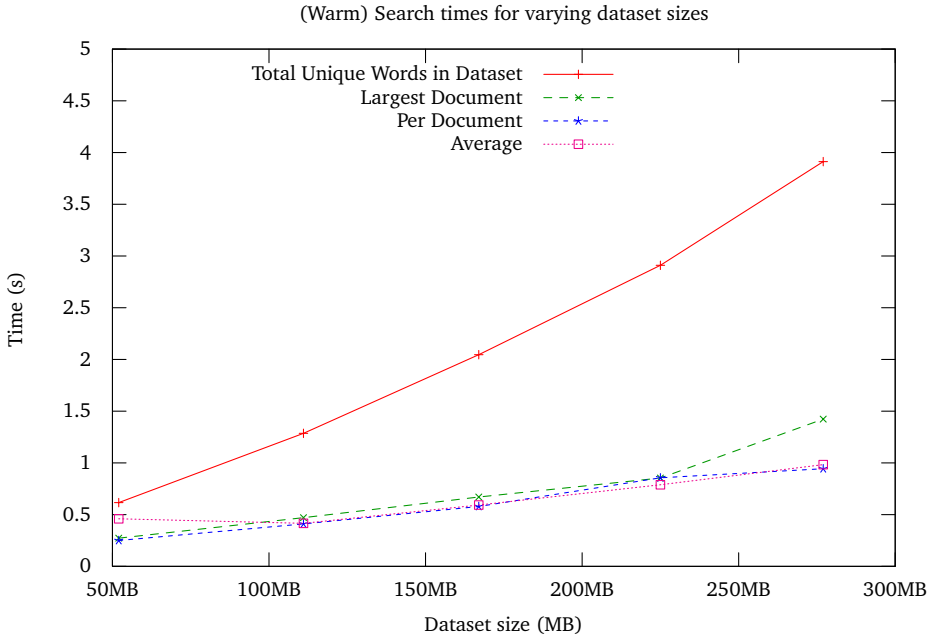

Figure 6.7: Graph showing (warm start) search times against corpus size with compression *disabled*.

Figure 6.8, compared to 6.7 shows that the compression used on the indexes has had a *negative* effect on the search performance. This is the opposite of what was seen in the encryption process, where compression of document indexes quite drastically reduced the time to encrypt and upload. The search process, however, runs just under 1s slower when searching the full dataset with compression enabled than with it disabled. This is because for each document, in order to determine if it was a match for the trapdoor specified, the document index must be decompressed before it is able to be queried. The decompression algorithm that is part of FastLZ is slower than the compression algorithm, which, combined with the fact that the search process is considerably faster than the encryption process anyway (and so the ratio between time spent in compression/decompression algorithms against time spent performing encryption/search is far larger), results in a large amount of extra overhead. Quite clearly, in an environment in which searches occur more frequently than document updates, compression of indexes should not be used as it offers little advantages for that environment. Since this is the environment that the secure index scheme itself is also tailored towards, compression should probably never be used.

Figure 6.9 details the difference between the various search options. Again, left-match search times are the clear outliers, with the other techniques staying relatively faithful to each other. The increased search time for left-matches is caused by the size of the indexes, as larger indexes cause PostgreSQL to take longer to find the document in its TOAST table (due to there being more pages to search), extract it (decompress it if the `extended` option is used, as is the case here) and for it to be de-TOASTed prior to the search being performed.

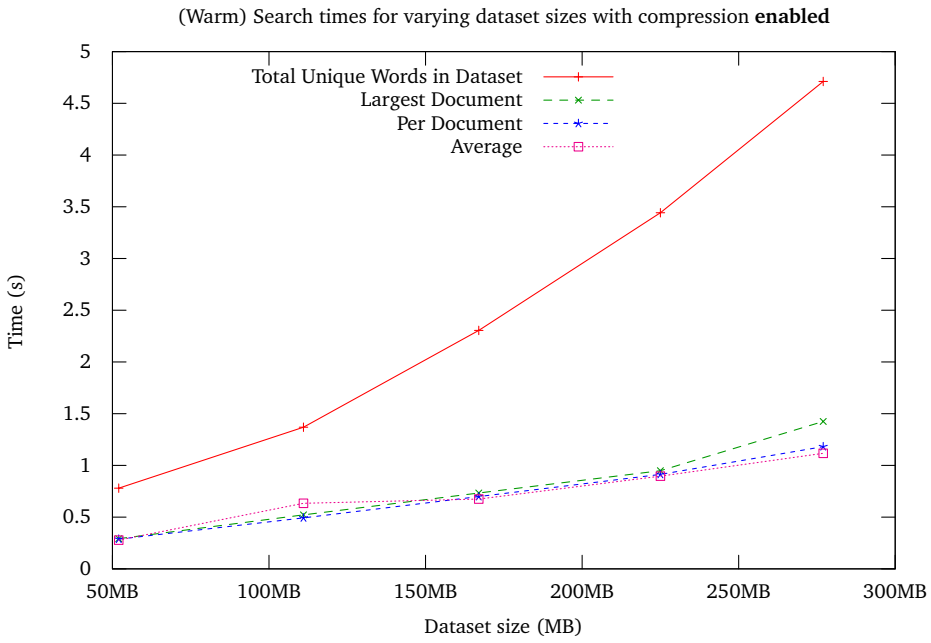

Figure 6.8: Graph showing (warm start) search times against corpus size with compression *enabled*.

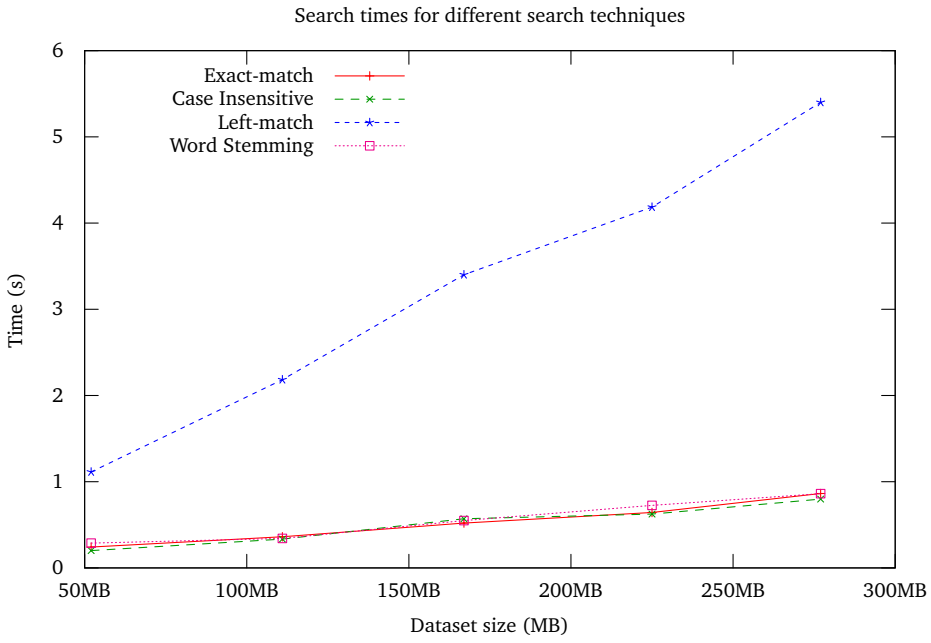

Figure 6.9: Graph showing search times for different search techniques (compression *disabled* and index sizes are calculated using the *per-document* option).

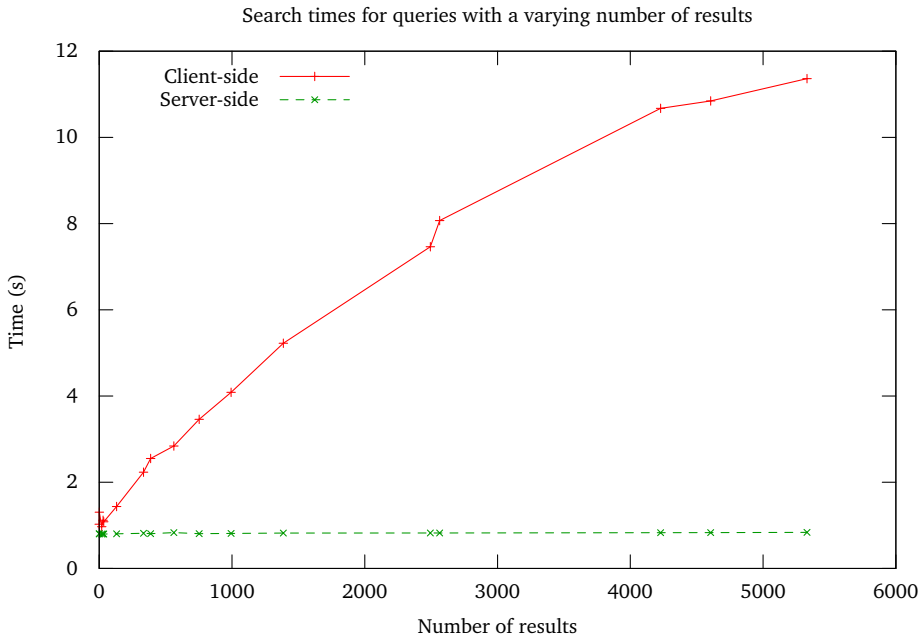

Figure 6.10: Graph showing search times for queries with a different number of results.

Although the secure index scheme in theory has a constant-time search algorithm, Figure 6.10 shows that it isn't quite so constant in practice. This graph shows that as the number of results (from the same corpus) returned increases, the time taken on the *client-side* increases. This is in contrary to the server-side search time which stays perfectly consistent. The client-side search time increases both because of the larger number of results that have to be checked for *false positives* (not necessarily more false positives, just more results to check), as well as any extra processing performed by the client (in the case of the client used here, this just consists of `printf`s to output the name of the document found).

6.5 Plaintext Comparison

An evaluation of the system implemented against an *insecure* database, storing information in plaintext, shows the *cost* of the added security it provides. There are two forms of plaintext search that are supported by PostgreSQL which will be investigated here. These are linear search and using a pre-processed index.

Figures 6.11, 6.12 and 6.13 illustrate these comparisons. PostgreSQL's Full Text Search (FTS) feature was used here to generate the plaintext index. Specifically, the relevant dataset was uploaded into the `data` column of a table called `plain` and the index was created using the following SQL statement:

```
CREATE INDEX plain_index ON plain USING gin(to_tsvector('english', data))
```

The index created by this statement is a Generalised Inverted Index (GIN). PostgreSQL offers an alternative to this, a Generalised Search Tree (GiST), but the latter type of index can return false positives (which have to be filtered by PostgreSQL prior to being returned). Because this would add extra overhead, the GIN index was used for comparisons in order to not artificially inflate the times for plaintext searches. GIN indexes are also specified for use in environments in which searches are more frequent than updates (in comparison to GiST indexes, which are better suited to applications with more updates), which is similar to that of the secure index scheme. Further information on PostgreSQL index support can be found at [45].

For a standard, linear search the SQL command executed simply uses the `LIKE` operator (with no wildcards). To utilise the FTS index, the query makes use of the `tsvector` and `tsquery` constructs (`$1` is replaced by the term to search for):

```
SELECT id FROM plain WHERE to_tsvector('english', data) @@ to_tsquery($1)
```

The encryption uses the per-document index size calculation algorithm and is only built with exact-match support.

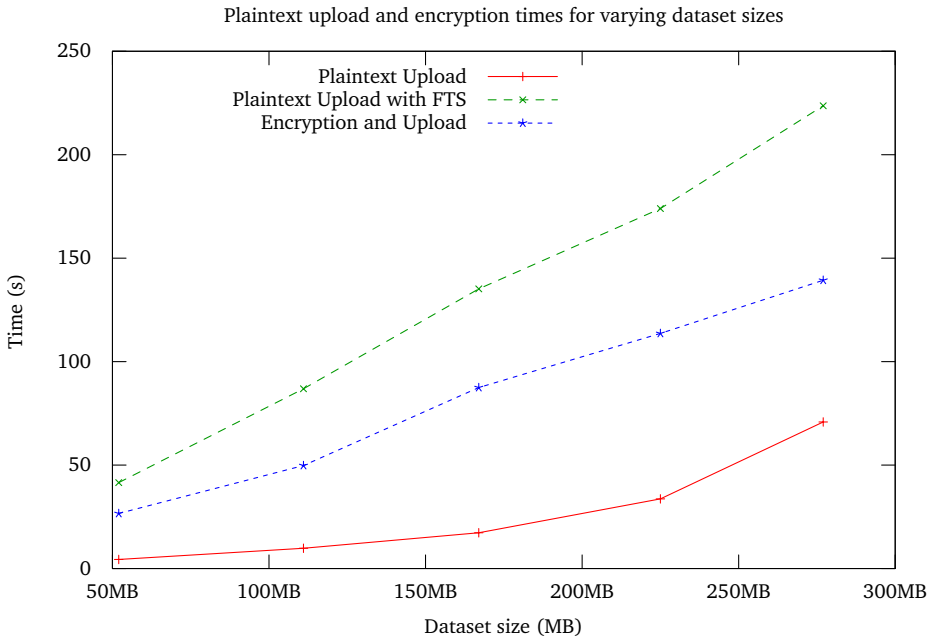

Figure 6.11: Graph showing both plaintext upload times (both with and without pre-processed indexes) along with encryption times for varying dataset sizes.

Figure 6.11's comparison of encryption time shows the secure index encryption process lying in between the two plaintext upload options. The plaintext index takes considerably longer to compute than the secure index, but a simple plaintext upload is significantly quicker than both the secure index encryption and the FTS index creation. This is to be expected, as no processing is taking place other than simple transfer of data.

Figure 6.12 demonstrates the additional size of the document indexes created by the secure index scheme, compared to those created by the standard plaintext GIN index created by PostgreSQL. The padding used as part of the encryption algorithm will also add to this overhead, but only minimally (a maximum of 16 bytes per document – 84KB for the entire document set).

Figure 6.13 shows the secure index search falling between the standard plaintext search and searches over GIN-indexed data. The index created by PostgreSQL on the plaintext is exceptionally fast to search, not straying far from 0.08s, regardless of dataset size. In comparison, the supposedly linear plaintext search time *without* an index seems to increase exponentially.

In my opinion, these figures demonstrate that the secure index encryption scheme can compete with plaintext searches in terms of both upload and search times. Databases found in industry are not typically configured with support for text indexes over large text fields unless they are tailored to scenarios in which fast data access times are imperative (such as web applications). As such, the introduction of a secure search scheme wouldn't introduce impossibly long waits in situations in which a standard plaintext search was previously used.

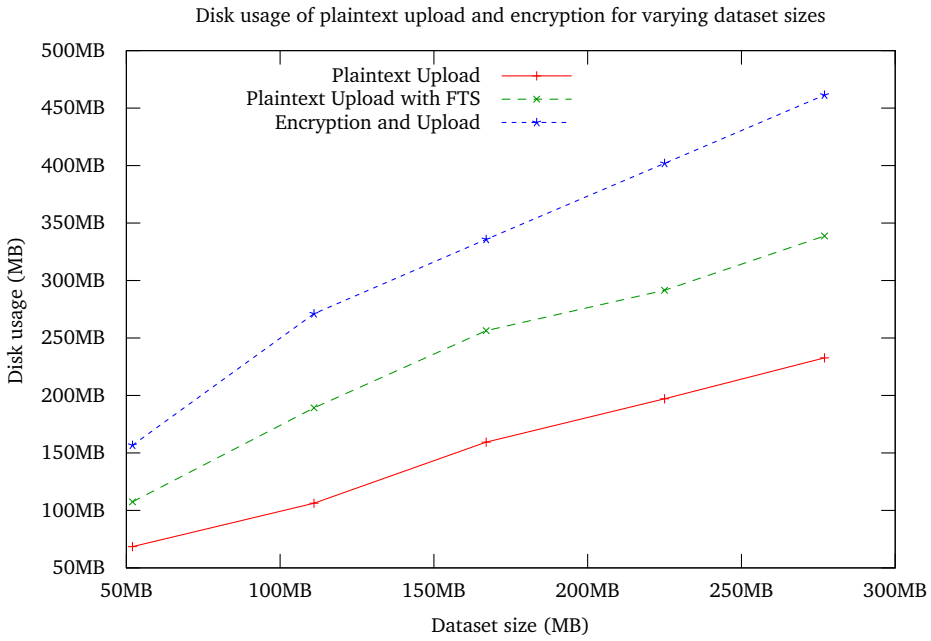

Figure 6.12: Graph showing the disk space usage of both plaintext upload (with and without pre-processed indexes) along with encryption for varying dataset sizes.

Figure 6.13: Graph showing the search times over both plaintext (with and without pre-processed indexes) and ciphertext for varying dataset sizes.

However, the secure search scheme is aimed at a situation in which updates are less frequent than queries, a specification that is shared by PostgreSQL's GIN indexes. Clearly there is a penalty associated with the introduction of this secure scheme, but it is not unbearable. If it was to be used in an update-heavy environment, the penalty would be considerably higher.

6.5.1 Summary

From these figures, we can see that the overhead of the encryption process *decreases* as the size of the dataset increases compared to the *unindexed* plaintext upload (from being around 6 times slower, to being about 2 times slower). However, when compared to the indexed upload, it is consistently around 1.5 times *faster*.

The space requirements of the encryption scheme are roughly twice that of the unindexed plaintext. Compared to the indexed plaintext however, the overhead of the encryption scheme drops to around 38%.

For the search process, the performance of the encrypted search *increases* compared to the unindexed plaintext search, from being around 1.2 times faster, to being over 2.2 times faster. The comparison against indexed plaintext search however, is not so favourable, with encrypted searches taking between 3.6 and 19 times longer to execute.

6.6 Potential Attacks

No evaluation of a security system would be complete without a discussion from the point of view of an attacker. There are a number of 'classic' attack models, some of which will now be discussed in relation to the system implemented in this project.

In a **ciphertext-only** attack, the attacker only has access to the *ciphertexts*. For example, a datacentre administrator that has access to the backend PostgreSQL server (with the secure index scheme installed) would have access to these ciphertexts. For each secure document, the attacker has available to them the document ID, the document index, the document ciphertext and their respective lengths.

The length of the document index will leak information about how many unique words are stored in the document set (depending on the index size calculation algorithm used) and the length of the ciphertext will be roughly equivalent to the length of the plaintext document. As discussed in 4.2.2, document IDs for each document should be *meaningless*, otherwise these will also leak information. If the index size calculation algorithm (the possible choices are detailed in Section 5.4.1) is anything other than per-document, then these index sizes will be the same for each document, reducing the amount of information available to an attacker when trying to differentiate between indexes.

One potential attack would be to take multiple documents and compare their indexes, however this avenue is blocked by the use of document identifiers in the creation of codewords. Identical documents with different IDs will produce different indexes, removing the possibility of statistical analysis, such as finding patterns of bits that are frequent in a number of document indexes, possibly identifying words that are common to the language (such as "the" and "because").

Because the hashes used to generate trapdoors are *keyed*, there is no way of generating a dictionary of hashes that could be used to quickly find trapdoors for words. The key information would have to be deduced too, which would require a technique such as brute-forcing, which would be equally applicable to the AES-output document ciphertext and is too computationally expensive to be deemed a viable attack.

An active, **known-plaintext** attack gives the attacker access to a number of known plaintext/ciphertext pairs. Firstly, a concrete example of how this might occur: a datacentre administrator sends an email to the company that is storing their data in his datacentre. This email might end up being stored securely there (if they back up their emails to secure, searchable storage). In this case, the rogue administrator

knows the contents of the plaintext email and if he can find the respective ciphertext then he has more information in order to determine the key. If the company has an automatic email backup system, then finding the ciphertext that corresponds to the email he sent would be trivial – he can just inspect the database logs that correspond to the time of his email.

With both the plaintext and ciphertext available, an attacker could attempt to work 'backwards' from the document index in order to determine trapdoors for words that are found in the plaintext he has. Since the index contains nothing but seemingly random bits, this is a difficult process. Again, the attacker could take a word in the plaintext and use a brute-force approach to generating trapdoors (by iterating over all possible keys) before comparing the resultant codeword with the document index. As the key used by the implementation is 512-bits long, this gives 2^{512} possible keys, which is obviously far too large to be iterated fully.

There is potential to exploit the strength of the hash algorithm used for Bloom filter insertions. These hash algorithms were chosen for speed and are not cryptographically secure (although the hashes used for key derivation are) and as such might not distribute evenly. If a pattern could be found between the input and output of the hash functions used, then there is a possibility that codewords could be constructed from the Bloom filter. These, however, are all combined with key information that was derived using a secure hash function, meaning the attacker would still have to break that. Since the key is derived from a password, then a number of standard attacks would be possible, such as a dictionary attack, but this is not a shortcoming of the scheme itself.

One of the advantages offered by the use of a Bloom filter in this scheme (as well as the performance it provides in terms of storage space) is that it is 'one-way'. Since there is a possibility that entries into the filter will map to the same bit-indexes, there is no definite way to infer the input from the output.

As mentioned in Chapter 1, the security of a system requires public scrutiny and so whether or not the system is 'secure' cannot be definitively answered. However, there are no obvious holes in the system and the original paper in which it is detailed [9] was published in 2004. Since no damning evidence has arisen since then to indicate any flaws, it could be seen as relatively secure.

7 Conclusion

This project centres on the development of a searchable encryption scheme for the PostgreSQL 8.3 database management system. Through this implementation, it is possible to host a PostgreSQL database on an untrusted host (perhaps in a datacentre), to store information on this host securely and to still be capable of performing *server-side* searches of the data without needing to trust the host, or download large amounts of encrypted data before decrypting and searching it locally. In Chapter 6, the system was shown to be robust enough to be used within industry and that the performance overheads associated with the encryption are not impractical. The system was shown to be comparable to *plaintext* search schemes, although it does come with certain performance penalties.

The developed system was used as a basis for experimentation for a number of different search techniques – namely exact matches, case insensitive search, left-most matches (right-most wildcarding) and a primitive 'natural language' search that makes use of word stemming. Searches of each type will still be performed securely by the server and leak little information. This demonstrates the extensibility of the system and shows that with further efforts, the potential for search query variety and complexity could verge on that of plaintext search.

In Chapter 6, the implementation was evaluated and figures detailing the performance of the system were produced. The set of all RFCs were encrypted using the system, totalling 277MB and taking just over 2 minutes (using the average word count). As such, the datarate achieved by the encryption algorithm using the *average* word count was around 2.21MB/s. This was found to be *faster* than the plaintext Full Text Search (FTS) index creation process.

The space overheads that the encryption scheme adds were found to be around 38% more than indexed plaintext, but around twice as much as unindexed plaintext. This could be seen to be overly high and hopefully, with further work these figures could be lowered.

The all important encrypted *search* algorithm was shown to be between 1.2 and 2.2 times *faster* than the linear plaintext search, taking under 1 second to search the entire set of RFCs – a datarate of roughly 300MB/s. But this was unfortunately outdone by the FTS indexed plaintext search, taking between 3.6 and 19 times longer to execute.

Experimentation showed that even though compression shaved a considerable amount of time off the *encryption* process, it should not be used, as it adds far too much overhead to the *search* process to be practical.

7.1 Future Work

There are a number of areas in which, with further time and efforts, this project could be extended.

Numerical Search

The ability to search for strings is important, but in a large number of applications (particularly databases), the need to search for *numbers*, or more specifically, to perform a check to see if a number falls within a certain *range*, is important. If these numbers have a meaning which the data owners are not prepared to divulge to the data host, a way of *encrypting* this data whilst maintaining the ability to perform server-side range checks would be of great interest. Research into this area does exist and an extension to the implementation presented in this project that makes use of this would be highly practical.

Hybrid Schemes

The secure index scheme implemented here could potentially be *combined* with another search scheme, Song's [19] linear search scheme for example. This particular combination would allow for a wider variety of searches – namely the ability to perform proximity and occurrence searches – without impeding the performance noticeably. This could be done by using the index scheme to first determine which documents (might) contain a match, and then using Song's scheme to subsequently search the document contents *linearly* for the search term.

Table Indexes

The current implementation stores document indexes in-line with their respective document. An extension to this that could potentially improve the performance of the system quite dramatically is the used of per-table indexes. This could be used to store the indexes in a quickly accessible form, with references to the documents they represent. This could be implemented within PostgreSQL through the use of 'operator classes' [44].

Geographic Data and Location Based Services

Services allowing users to search *geographic* data are becoming increasingly fashionable. These include applications such as Google Maps and Geonames.org which both relate search queries to specific locations. Searches over this type of data can be particularly revealing of potentially private information. For example, if a user searches for "*pubs in South Kensington*" it could quite easily be inferred by the server administrator that this person has a high probability of being in or around South Kensington. This sort of information could potentially be *encrypted* using a scheme similar to the one developed here, with server-side information being stored securely and search queries not leaking information about the actual search, or the *locations* that match the query, thus providing a level of privacy to the users of the service. This type of application requires more rigorous key management processes, as well as a scheme more specifically intended for shorter textual data (the implementation presented in this project is most efficient when dealing with large text fields).

Integration with Another System

As shown in Chapter 3, there are a number of possible application areas for this type of system. Further refinement of the common library implemented as part of this project and integration with another one of these applications would demonstrate its portability and help to evaluate it in a different scenario.

7.2 Final Remarks

The research area of producing searchable ciphertext is currently an immature field. The key problem lies in the fact that *patterns* are required in order to search, and patterns are exactly what you *don't* want when producing ciphertext, since they give an adversary a possible method of attack.

The indexing scheme implemented in this project shows that it is possible to add a number of different *search types* to the scheme, but it also shows that the addition of each different type requires a highly *specific* technique. There is no way of implementing a "generic" search system (perhaps something like a regular expression language that allows you to formulate a number of different queries) as each different search type makes use of a very specific 'trick' or implementation detail. Because of this I don't feel that progress in this area will advance considerably until this problem is overcome. Each different type of search, over different datatypes, combined in different manners (such as aggregation operators in SQL – `count`, `sum`, etc) all need techniques that are tailored specifically to their individual requirements.

Overall I am pleased with the outcome and success of the project and I feel that I have achieved what I had set out to at the beginning. The implementation of Goh's scheme is the first I am aware of, with his paper only detailing the theory behind it. The performance of the theoretical approach is also improved, with datarate of the search algorithm increased from around 70MB/s to roughly 300MB/s. The implementation of a left-match search scheme has also not been tried before, although the results achieved with this technique were not entirely acceptable. With more time, it should be possible to lower the overheads of the system further, reducing both the disk space and the search time required.

Bibliography

[1] Announcing the Advanced Encryption Standard (AES), November 2001. Federal Information Processing Standards Publication 197.

[2] An RSA Laboratories Technical Note. PKCS #7: Cryptographic Message Syntax Standard. pages 20–21, 1993.

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

[4] Dan Boneh, Giovanni Di Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano. Public Key Encryption with Keyword Search. In *proceedings of Eurocrypt 2004*, 2004.

[5] Bryan Klimt and Yiming Yang. Introducing the Enron Corpus. In *Conf. on Email and Anti-Spam (CEAS)*, Mountain View, CA, 2004.

[6] Clayton Donley. *LDAP Programming, Management and Integration*. Manning, 2002.

[7] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions, 2006.

[8] Dan Boneh and Eyal Kushilevitz and Rafail Ostrovsky and William E. Skeith III. Public-key encryption that allows PIR queries. Unpublished Manuscript. Technical report, In Proc. of CRYPTO '07, 2006.

[9] Eu–Jin Goh. Secure indexes. In *the Cryptology ePrint Archive, Report 2003/216*, March 2004.

[10] Gordon Good. The LDAP Data Interchange Format (LDIF) - Technical Specification, 2000.

[11] Hakan Hacigümüş, Bala Iyer, Chen Li, and Sharad Mehrotra. Executing SQL over Encrypted Data in the Database-Service-Provider Model. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 216–227, New York, NY, USA, 2002. ACM.

[12] Heinz Johner, Larry Brown, Franz-Stefan Hinner, Wolfgang Reis, and Johan Westman. *Understanding LDAP*. IBM Corp., Riverton, NJ, USA, 1998.

[13] Joonsang Baek and Reihaneh Safiavi-naini and Willy Susilo. Public Key Encryption with Keyword Search Revisited. Cryptology ePrint Archive, Report 2005/191, 2005.

[14] M. F. Porter. An algorithm for suffix stripping. In *Readings in information retrieval*, pages 313–316, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[15] R. Brinkman and L. Feng and J. Doumen and P. H. Hartel and W. Jonker. Efficient tree search in encrypted data. *Information Systems Security Journal*, 13:14–21, 2004.

[16] Richard Brinkman and Jeroen Doumen and Willem Jonker. Using Secret Sharing for Searching in Encrypted Data, 2004.

[17] S. E. Robertson and K. Sparck Jones. Simple Proven Approaches to Text Retrieval. Technical report, Cambridge Univ. Computer Laboratory, 1997.

[18] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition*. Wiley, October 1995.

[19] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical Techniques for Searches on Encrypted Data. In *proceedings of IEEE Symposium on Security and Privacy*, May 2000.

[20] Ashwin Swaminathan, Yinian Mao, Guan-Ming Su, Hongmei Gou, Avinash L. Varna, Shan He, Min Wu, and Douglas W. Oard. Confidentiality-preserving rank-ordered search. In *StorageSS '07: Proceedings of the 2007 ACM workshop on Storage security and survivability*, pages 7–12, New York, NY, USA, 2007. ACM.

[21] Xiaoyun Wang and Hongbo Yu. How to break MD5 and other hash functions. In *In EUROCRYPT*. Springer-Verlag, 2005.

[22] Yan-cheng Chang and Michael Mitzenmacher. Privacy Preserving Keyword Searches on Remote Encrypted Data. In *In Proc. of 3rd Applied Cryptography and Network Security Conference (ACNS)*, pages 442–455, 2005.

Web References

[23] Apache Directory Server. http://directory.apache.org/apacheds/1.5/.

[24] Beagle Project. http://www.beagle-project.org/.

[25] Botan. http://botan.randombit.net/.

[26] Crypto++. http://www.cryptopp.com/.

[27] FastLZ Website. http://www.fastlz.org/.

[28] FUSE. http://fuse.sourceforge.net/.

[29] General Purpose Hash Algorithms. http://www.partow.net/programming/hashfunctions.

[30] GNU Libgcrypt. http://www.gnupg.org/download/#libgcrypt.

[31] Google Desktop Homepage. http://desktop.google.com/.

[32] List of English Stop Words. http://jmlr.csail.mit.edu/papers/volume5/lewis04a/a11-smart-stop-list/english.stop.

[33] Microsoft CAPI. http://msdn.microsoft.com/en-us/library/aa380256(VS.85).aspx.

[34] MySQL Case Sensitivity. http://dev.mysql.com/doc/refman/5.0/en/case-sensitivity.html.

[35] MySQL DBMS. http://www.mysql.com/.

[36] OpenFTS. http://openfts.sourceforge.net/.

[37] OpenFTS Introduction. http://openfts.sourceforge.net/primer.html.

[38] OpenLDAP. http://www.openldap.org/.

[39] PGXS Extension Documentation. http://www.postgresql.org/docs/8.3/interactive/xfunc-c.html#XFUNC-C-PGXS.

[40] PostgreSQL DBMS. http://www.postgresql.org/.

[41] PostgreSQL Extensibility Documentation. http://www.postgresql.org/docs/8.3/interactive/extend.html.

[42] PostgreSQL Full Text Search. http://www.postgresql.org/docs/8.3/static/textsearch-intro.html.

[43] PostgreSQL License. http://www.postgresql.org/about/licence.

[44] PostgreSQL Operator Classes.
http://www.postgresql.org/docs/8.3/static/sql-createopclass.html.

[45] PostgreSQL Plaintext Indexes.
http://www.postgresql.org/docs/8.3/static/textsearch-indexes.html.

[46] PostgreSQL TOAST Documentation.
http://www.postgresql.org/docs/8.3/interactive/storage-toast.html.

[47] Request For Comments Database. http://www.ietf.org/rfc.html.

[48] RFCs, 2001-2500 (GZip archive).
ftp://ftp.rfc-editor.org/in-notes/tar/RFCs2001-2500.tar.gz.

[49] Samba Homepage. http://www.samba.org/.

[50] Spotlight. http://www.apple.com/macosx/features/300.html#spotlight.

[51] SQLite DBMS. http://www.sqlite.org/.

[52] SQLite Deployment Statistics. http://www.sqlite.org/mostdeployed.html.

[53] SQLite Expressions. http://www.sqlite.org/lang_expr.html.

[54] SQLite Full Text Search. http://www.sqlite.org/cvstrac/wiki?p=FullTextIndex.

[55] SQLite Licence. http://www.sqlite.org/copyright.html.

[56] SSHFS. http://fuse.sourceforge.net/sshfs.html.

[57] The Collected Works of Shakespeare. http://www.cs.su.oz.au/~matty/Shakespeare/.

[58] The Complete Works of Shakespeare (HTML). http://shakespeare.mit.edu/.

[59] University of Washington IMAP Toolkit. http://www.washington.edu/imap/.

[60] USENET Corpus (2005-2008).
http://www.psych.ualberta.ca/~westburylab/downloads/usenetcorpus.download.html.

[61] M. Crispin. RFC 3501: Internet Message Access Protocol, Version 4rev1, March 2003.
http://www.ietf.org/rfc/rfc3501.txt.

[62] Dan Bernstein. DJB Hash.
http://groups.google.co.uk/group/comp.lang.c/browse_thread/thread/9522965e2b8d3809/380a6124dd59cbce.

[63] J. Myers and M. Rose. RFC 1939: Post Office Protocol, Version 3, May 1996.
http://www.ietf.org/rfc/rfc1939.txt.

These addresses were last checked on the 15^{th} June, 2009.