LETTERKENNY INSTITUTE OF TECHNOLOGY

A thesis submitted in partial fulfilment of the requirements for the Master of Science in Computing in Systems and Software Security at Letterkenny Institute of Technology

SEARCHABLE SYMMETRIC ENCRYPTION (SSE): A MECHANISM FOR SEARCHING SYMMETRICALLY ENCRYPTED DATA STORED IN THE CLOUD

Author:	Supervisor:
Shaun Mc Brearty B.Sc.	Mr. William Farrelly M.Sc., B.Sc.

Submitted to Higher Education and Training Awards Council (HETAC) August 2015

Declaration

I hereby certify that the material, which I now submit for assessment on the programme of study leading to the award of Master of Science in Computing in Systems and Software Security, is entirely my own work and has not been taken form the work of others except to the extent that such work has been cited and acknowledged within the text of my own work. No portion of the work contained in this thesis has been submitted in support of an application for another degree or qualification to this or any other institution.

Signature of Candidate: Date:

_________________________ _________________

Acknowledgements

First and foremost, the author would like to thank supervisor Mr. Billy Farrelly for his advice and support throughout the duration of this dissertation. After my initial dissertation topic proved to be infeasible in the allocated time, Billy suggested a number of alternative topics to me, including the topic I ultimately opted for: Searchable Encryption. As a complete newcomer to the topic of Searchable Encryption (and Cryptography in general), I often struggled when it came to communicating my ideas and thoughts on the topic, particularly in my writings; however Billy's ability to interpret my ramblings and his ability to phrase these ideas in a manner much more clear and concise than my own was abundantly helpful.

Outside of this dissertation, I would like to place on record my sincere thanks to Billy for everything he has done for me in my career thus far. Billy was instrumental in helping me secure my first two employments after completing my undergraduate education, as well as pointing me in the direction of numerous other lucrative employment and educational opportunities in the last three years.

I would also like to thank my instructors on the taught portion of the course: Mr. Nigel McKelvey, Mr. John Mc Garvey, Mr. John O'Raw, Ms. Ruth Lennon and Dr Mark Leeney. They are five of the finest educators I have ever encountered and their devotion to their work and their willingness to help their students in any

manner necessary goes well beyond that which can reasonably be expected from people with such busy schedules.

Although I have never met the man, I will be forever grateful to Seny Kamara of Microsoft Research. Being completely new to the topic of Searchable Encryption, I struggled greatly with a lot of the initial material I encountered; however Seny's blog postings on Searchable Encryption[1] – written for those who are completely new to the concept – simplified this process greatly.

To my family – John (Dad), Tina (Mum) and Kevin (Brother) – I thank you from the bottom of my heart for all your help and support over the past three years. All three of you helped to provide me with a home environment that allowed me to focus on my studies almost exclusively. For the last three years I've managed to avoid brushing the dog, making my own bed, cooking a dinner and washing my own clothes – all in the name of attaining my Master's Degree. Now that this process is almost at an end, I have no doubt I will spend the next month trying to find a new and improved excuse to avoid same. In terms of finance, a massive thank you to both my parents for covering the costs associated with the course. Neither of you batted an eye lid or moaned and groaned whenever Bank Giro's arrived in the post at various times over the last three years, and for that I am eternally grateful. Although I don't say it very often, I love all three of you very much.

[1] www.outsourcedbits.org

I would also like to thank my partner Louise for her constant support, love and understanding over the past few years. Louise has had to endure me during many a dissertation/exam induced bad mood during our time together and for that I am very sorry. In addition, her understanding and acceptance of my excessive workload these past few years has been nothing short of remarkable. Not once did she make an issue about the numerous hours I spent with my head in a book/laptop, or the tiredness and bad moods that were associated with it. Thank you for putting up with me - I love you more than you will ever know. I would also like to take this opportunity to promise you that never again will I proceed to talk about Searchable Encryption during Keeping up with the Kardashians.

Lastly, I would like to thank my electric kettle and Tetley’s Tea Bags. Caffeine played an essential part in the making of this dissertation.

Glossary of Terms

Please note that the following definitions have been formulated in the context of text based Information Retrieval (IR) and Searchable Symmetric Encryption (SSE).

Collection: A set of Documents (Manning *et al.,* 2008, p.4). The term Corpus or Database is sometimes used as an alternative to Collection (Song *et al.,* 2000; Manning *et al.,* 2008, p.4).

Document: A digital text file. *For Example: PDF, DOC, DOCX, TXT, HTML*.

Document Relevance: A Document is considered Relevant if it contains information of value in relation to a person's Information Need (Manning *et al.,* 2008, p.5).

Dictionary: *'A […] resource containing […] the Words of a language, giving information about their meanings, pronunciations, etymologies, inflected forms, derived forms, etc.'* (Dictionary.com, 2015a).

Free Text Query: A Query expressed in a natural language (*For Example: English*) without the use of formal operators (*For Example: Boolean Operators)* (Manning *et al.,* 2008, p.14). A Free Text Query consists of one or more Terms.

Information Need: A topic about which a person wishes to learn/know more about (Manning *et al.,* 2008, p.5). An Information Need is typically conveyed as a Query.

Information Retrieval (IR): The act of locating Documents from within a Collection that satisfies a person's Information Need (Manning *et al.,* 2008, p.1).

Information Retrieval Operation: The act of using an Information Retrieval System.

Information Retrieval System: A software system that performs Information Retrieval (Manning *et al.,* 2008, p.2). In order to use an Information Retrieval System, a person must first input their Information Need in to the system in the form of a Query. In turn, the Query is utilised by the Information Retrieval System to determine the Relevance of the Documents contained within the Collection (with respect to the specified Query). Once the contents of the Collection have been examined, the set of Documents considered Relevant; *that is, the Search Results*, are returned to the user of the system. Internet Search Engines and Operating System File Search Functions are particularly prevalent example of Information Retrieval Systems (Manning *et al.,* 2008, p.5).

Query: One or more Terms chosen to convey a person's Information Need (Manning *et al.,* 2008, p.5), typically in the form of a Free Text Query (Manning *et al.,* 2008, p.14).

Query Term: An individual, unspecified Term that occurs within the text associated with a Free Text Query.

Search: See Information Retrieval. Used Interchangeably.

Searchable Encryption: Information Retrieval performed on a Collection consisting of encrypted text Documents. The Documents within the Collection remain encrypted at all times, while the Query specified by the user is encrypted prior to being utilised in the Information Retrieval Operation that follows (Song *et al.,* 2000).

Search Operation: See Information Retrieval Operation. Used Interchangeably.

Search Result: The set of Documents (from within a Collection) considered Relevant after Information Retrieval has been performed.

Search String: See Free Text Query. Used interchangeably.

Term: The individual space delimited text units that make up the contents of both Documents and Free Text Queries. Given that it is commonplace for Documents and Free Text Queries to contain text not classified as Words (*For Example: K-9, Blink-182*), Term is used instead to describe the individual space delimited text units that make up the contents of both Documents and Free Text Queries. It should be noted that all Words are classified as Terms, but not all Terms are classified as Words (Manning *et al.,* 2008, p.3-4). *For Example: 'the' is both a Word and a Term*; *'asdfg' is a Term, but not a Word.*

Word: An individual unit of a given language (Dictionary.com, 2015b) (*For Example: 'a', 'the', 'from', etc*.), listed as an entry in the Dictionary of the associated language (Manning *et al.,* 2008, p.3-4).

Abbreviations List

CSP: Cloud Service Provider

DS: Data Set

ESS: Encrypted Search String

FHE: Fully Homomorphic Encryption

IND-CKA2: Indistinguishable Against Adaptive Chosen Keyword Attacks

IR: Information Retrieval

ORAM: Oblivious RAM

RAM: Random Access Memory

SSE: Searchable Symmetric Encryption

TDF: Term-Document Frequency

Table of Contents

Table of Figures

1. Introduction

1.1 Purpose

This Document represents the author’s thesis submitted in partial fulfilment of the requirements for the Dissertation module of the Master of Science in Computing in Systems and Software Security at Letterkenny Institute of Technology.

1.2 Overview

The concept of Cloud computing is now an accepted philosophy for computing. As of 2014, 19% of all enterprises within the European Union utilise Cloud computing in some form or another (Eurostat, 2014), with industry forecasts indicating significant growth in the sector over the coming years (Columbus, 2015).

The benefits of Cloud computing are significant: reduced costs, high reliability, as well as the immediate availability of additional computing resources as and when needed. Despite such advantages, Cloud Service Provider (CSP) consumers need to be aware that the Clouds poses its own set of unique risks that are not typically associated with storing and processing one’s own data internally using privately owned infrastructure (Hashizume *et al.,* 2013).

Perhaps the most severe risk facing CSP consumers at present is the threat of data disclosure or data loss (OWASP, 2013). Recent years have seen a number of such incidents occur, whereby organisations customer data – hosted on the Cloud - has been leaked online (for hacktivism or vandalism purposes) or stolen for criminal purposes. Victims of such attacks include large organisations such as Sony (Sony, 2014), Adobe (Arkin, 2013) and Apple (Kerris and Muller, 2014).

Cloud computing is made possible through the use of many technologies, including internet access, virtualisation and third party data centres. As a result, Cloud computing has a much broader attack surface than that associated with storing and processing data internally using privately owned infrastructure. The storing of consumer data online makes such information – potentially - accessible to anyone with a web browser, while the use of virtualisation technology has the potential to allow CSP consumers to gain access to other CSP consumer's private data and/or applications (Zhang *et al.,* 2012; Hashizume *et al.,* 2013). In addition, the use of third party data centres poses a number of potential risks, including employees of the CSP (both current and former) gaining access to private consumer data (either physically or via software) (Claycomb and Nicoll, 2012; Hashizume *et al,.* 2013; Intermedia.net, 2014; Nguyen *et al.,* 2014).

As a countermeasure to such attacks, various access controls are utilised: In the case of online access to the CSP, such access controls typically take the form of usernames and passwords; In the case of virtualisation, such access controls typically take the form of logical data separation; and in the case of third party data centres, such access controls typically take the form of physical access controls (*For Example: Locks, Keypads*) (as well as software based access control) that prevent unauthorised CSP personnel from gaining access to user data (Hashizume *et al.,* 2013). In principle, all of the aforementioned access controls are sound; however in practice, such controls have been circumvented.

In the event that any of the aforementioned access controls are compromised maliciously, the chances of a data breach occurring are high. Should a data breach occur and the associated data is retrieved in encrypted form, the data is essentially useless to an attacker (unless the encryption algorithm utilised is weak and/or the attacker has some foreknowledge of the associated decryption key) (Zhang *et al.,* 2012; Hashizume *et al,* 2013); however, in the event that a data breach occurs and the associated data is retrieved in plaintext form, an organisations worst nightmare has become a reality. What follows is typically a slew of press releases, negative publicity, damaged business reputations, and fines under various data protection laws (ICO, 2014; ICO, 2015; Levick.com, 2015).

To reduce the impact of potential data breaches (and to provide privacy for CSP consumer data) CSPs typically employ the use of cryptography. In a Cloud environment, cryptography is typically utilised for two purposes: security while data is at rest; and security while data is in transit. Unfortunately the Cloud cannot guarantee the security of data during processing as the current limitations of cryptography prevent data from being processed in encrypted form.

Given the fact that data is processed in unencrypted form, it is quite common for attackers to target data in use, rather than targeting data which is encrypted during storage and transit (Hashizume *et al.,* 2013; OWASP, 2013).

At the time of writing[2], an entity wishing to store its data within the Cloud must choose one of the following options:

1. Store Data in Encrypted Form (Two Options Exist)
    - A. Disclose Decryption Key(s) to Cloud Service Provider (CSP) <u>OR</u>
    - B. Keep Decryption Key(s) Private
2. Store Data in Unencrypted Form

Option 1A requires encrypted data owners to disclose their decryption key(s) to CSPs. This is due to the fact that data cannot be searched or operated on while in encrypted form. In order to provide CSP customers with such functionality, CSPs require access to the necessary decryption key(s).

Option 1B (Keeping Decryption Key(s) Private) represents the most secure sub-option; however, as previously mentioned, CSP customers lose the ability to search or operate on their data while it is in encrypted form. In order to utilise such functionality using Option 1B, CSP customers must download their data, decrypt it, and only then can it be searched and/or operated on. While this approach may be fine for small amounts of data, it becomes increasingly inefficient and unwieldy as the amount of data increases. In addition, should any changes be made to the data after it has been downloaded; the customer must then re-encrypt and re-uploaded the entire dataset to the Cloud.

[2] September 14th 2015

Option 2 avoids the use of encryption for data security. Rather than relying on cryptography for data security; *that is, the traditional approach to data security*, this approach utilises the aforementioned approach of logically separating data (Mather *et al.,* 2009, p.62).

Evidently, none of the options available at present provide an adequate balance of data security and functionality. Option 1A and Option 2 offer full functionality at the expense of data security, while Option 1B provides data security at the expense of any and all functionality (Mather *et al.,* 2009, p.66).

The ideal solution to achieving an optimal balance of data security and functionality within the Cloud involves the CSP having the ability to search and operate on data while it is in encrypted form – without having any knowledge of the associated decryption key(s), or the associated plaintext(s) (Mather *et al.,* 2009, p.62-63, 69).

Two forms of encryption do in fact exist at present that make the above a reality. The first, known as Fully-Homomorphic Encryption (FHE) allows data to be *operated on*[3] while in encrypted form (Gentry, 2009). The second, known as Searchable

[3] Given two single digit binary numbers N1 and N2, *that is, 0 or 1*, an encryption scheme is said to be Fully Homomorphic if it satisfies the following property: $N1 \oplus N2 = N3$; $E^{PK}(N1) \oplus E^{PK}(N2) = E^{PK}(N3)$; $D^{SK}(E^{PK}(N3)) = N3$, where $\oplus$ denotes both binary multiplication **and** binary addition, *i.e. mod 2*.

An encryption scheme that satisfies the above property can be used to derive a software based NAND Logic Gate (which can in turn be used to derive the set of all other Boolean Logic Gates). Given the full set of Boolean Logic Gates, any Boolean Logic Circuit can be derived; therefore allowing encrypted data to be arbitrarily operated on in the exact same manner as plaintext data (albeit plaintext data is operated on at a hardware level; not a software level as is the case with Fully Homomorphic Encryption).

Encryption, allows for data to be *searched* while in encrypted form (see Information Retrieval (Page 5)) (Song *et al.,* 2000).

While being impressive in terms of its functionality and capabilities, FHE remains extremely slow when implemented in software (Gentry *et al.,* 2015). As such, its mass deployment and usage within the Cloud appears to be some way off.

Searchable Encryption on the other hand has been shown to be sufficiently efficient on the few occasions that it has been implemented in software. Despite being a relatively obscure form of Cryptography, a number of researchers in the area hold the opinion that Searchable Encryption is now at the point that it can be deployed and used within the Cloud (Kamara *et al.,* 2012; Cash *et al,.* 2013).

Given its superior efficiency, the author has chosen to focus on the topic of Searchable Encryption for this dissertation. The author's decision to do so is motivated by the Clouds lack of support for true data security using encryption, as well as the Clouds lack of support for searching encrypted data.

Used in the Cloud, Searchable Encryption has the ability to allow CSP customers to store their data in encrypted form, while retaining the ability to search that data without disclosing the associated decryption key(s) to CSPs (Song *et al.,* 2000), *that is, without compromising data security on the Server*.

Searchable Encryption is a diverse subject that exists in many forms. While there are several methods of carrying out Searchable Encryption, two general techniques dominate the literature: Searchable Symmetric Encryption (SSE); *that is, Searchable Encryption using Symmetric Key Cryptography* and Public Key Encryption with

Keyword Search (PEKS); *that is, Searchable Encryption using Public Key Cryptography (Boneh et al., 2004; Curtmola et al,. 2006; Bosch et al., 2014)*. Neither SSE nor PEKS natively supports Searchable Encryption as it was originally envisioned by Song *et al.* (2000) (see Section 2.1). Instead, the literature has focussed on adapting various forms of Indexes; *that is, Data Structures that support efficient searching by pre-computing and mapping Search Terms to the Documents they occur in (and vice versa)*, for use with Information Retrieval (IR) over encrypted Documents (Bosch *et al.,* 2014).

Two forms of Indexes are discussed in the Searchable Encryption literature: Forward Indexes and Inverted Indexes (see Section 2.2). While both Indexes store the exact same information, each Index is optimised for different forms of searching. In the case of the Forward Index, it is optimised for searching *specific Documents* for the presence of Search Strings (Luenberger, 2006, p.285), while the Inverted Index is optimised for searching *an entire Document Collection* for the presence of Search Strings (Luenberger, 2006; p.286; Manning *et al.,* 2008, p.6). Early work on the topic of Searchable Encryption focussed on the use of Forward Indexes almost exclusively (Goh, 2003; Chang and Mitzenmacher, 2005); however, subsequent work on the topic has focussed on the use of Inverted Indexes (due to its ability to efficiently search an entire Document Collection, as opposed to specific Documents) (Curtmola *et al.,* 2006; Van Liesdonk *et al.,* 2010; Kamara *et al.,* 2012; Cash *et al.,* 2013).

Aside from SSE and PEKS, two other forms of encryption exist at present that support Searchable Encryption: Fully-Homomorphic Encryption (FHE) (Gentry, 2009) and Oblivious RAM (ORAM) (Goldreich and Ostrovsky, 1992).

As mentioned previously, Fully-Homomorphic Encryption supports computations over data in encrypted form, including Searchable Encryption as it was originally envisioned by Song et al (2000); nonetheless efficient Fully-Homomorphic Encryption remains someway off (Gentry *et al.,* 2015).

Used in isolation, ORAM does not support Searchable Encryption. Essentially, ORAM is a Client-Server communication protocol designed to obfuscate memory access patterns on the Server side of a given transaction. In its simplest form, ORAM consists of two operations: The Client storing data on the Server; *that is, writing*, and the Client retrieving Data from the Server; *that is, reading*. In an effort to obfuscate memory access patterns on the Server, each Write operation is also accompanied by an associated Read operation, and each Read operation is accompanies by an associated Write operation. In addition, each Read/Write operation accesses numerous memory locations on the Server instead of just a single memory location (in an effort to further obfuscate memory access patterns; *that is, false positives*) (Goldreich and Ostrovsky, 1992). In the context of Searchable Encryption, ORAM is typically combined with SSE and PEKS Searchable Encryption schemes to improve their security. SSE and PEKS Searchable Encryption schemes Leak Information to the Server a number of ways (see Section 2.3.1). By combining such schemes with ORAM, such Information Leakage can be eradicated; nonetheless, the search efficiency of schemes utilising ORAM is severely hindered due to the amount of work involved in obfuscating memory access patterns using ORAM (Stefanov *et al.,* 2013; Hahn and Kerschbaum, 2014).

In relation to search efficiency, both SSE and PEKS achieve optimal search time when used in conjunction with an Inverted Index; *that is, search time is linear in the number of Documents matching the Search String*; however in terms of security, SSE is vastly superior to PEKS (Bosch *et al.,* 2014). Given that PEKS is a form of Public Key encryption, an adversary can easily mount an attack on such a Searchable Encryption scheme given the associated Public Key and a dictionary of chosen Terms (Boneh *et al.,* 2004). In the case of SSE, all associated keys are kept private (Curtmola *et al.,* 2006).

As part of this dissertation, the author has chosen to focus on the topic of SSE using the Inverted Index. The author's decision to do so is due to the fact that SSE represents the most efficient form of Searchable Encryption (see Figure 1) at the time of writing[4]. The author readily acknowledges that SSE is rather unorthodox in its working, particularly when compared to other Searchable Encryption solutions; nonetheless, SSE represents one of the few forms of Searchable Encryption that is achievable using established standardised encryption algorithms. Alternative forms of Searchable Encryption require the use of non-standardised, special purpose encryption algorithms (Gentry, 2009). As regards security, the author also

[4] September 14th 2015.

acknowledges that SSE is considered one of the least secure forms of Searchable Encryption (see Figure 1); primarily due to Information Leakage. Solutions exist to eradicate and obfuscate all forms of Information Leakage in SSE; however existing solutions have a significant effect on the search efficiency of SSE (Stefanov *et al.,* 2013; Hahn and Kerschbaum, 2014). Evidently, the challenge for researchers is to improve the security of SSE while maintaining its superior search efficiency.

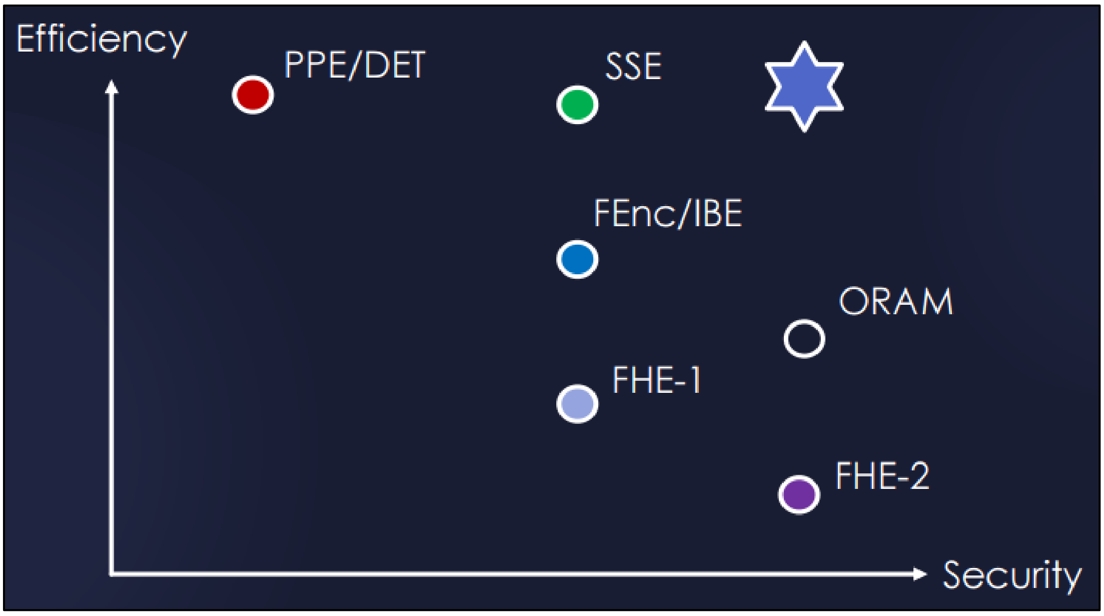


Figure 1: Efficiency Vs. Security Trade-off For Various Searchable Encryption Schemes (Kamara, 2013).

Figure 1[5] lists all known solutions to the problem of searching on encrypted data; *that is, symmetrically encrypted data, as well as public key encrypted data*. The y-axis of Figure 1 lists all Searchable Encryption solutions with respect to their efficiency, while the x-axis lists all solutions with respect to security. As regards

[5] FEnc/IBE represent extensions of PEKS Searchable Encryption, while PPE/DET refers to Searchable Encryption schemes that utilise Symmetric Key Encryption or Public Key Encryption exclusively (SSE utilises a combination of Symmetric Key Encryption and Hash Functions; hence SSE being classified as a separate form of Searchable Encryption).

efficiency, the SSE literature defines efficiency as the time-complexity associated with finding a given Encrypted Search String (ESS) within a body of encrypted data (expressed in Big O Notation). In terms of security, the SSE literature defines security as the amount of Information Leakage associated with using a given Searchable Encryption scheme; *that is, what the Server learns (or can deduce) about the ciphertext by searching over it* (expressed in Terms of the numerous categories of Information Leakage) (Kamara, 2013).

1.3 Outline of Dissertation

Chapter 2 constitutes the Literature Review conducted as part of this dissertation.

The Literature Review begins with a brief overview of the concept of Searchable Encryption (Section 2.1). As part of this overview, the author identifies the major reasons why symmetrically encrypted ciphertext cannot be searched in the same manner as plaintext data, before briefly discussing Searchable Symmetric Encryption (SSE) – the solution to searching same.

In its most basic form, SSE is nothing more than an Inverted Index – a mechanism that has been used in plaintext Information Retrieval (IR) for decades - that has been modified and adapted for use with ciphertext.

To familiarise the reader with the concept and operation of an Inverted Index, Section 2.2 provides an in-depth discussion of the Inverted Index as used in plaintext IR. This is followed by a discussion of the how the Inverted Index has been adapted and modified for use with ciphertext; *that is, SSE*, in Section 2.3, as well as a brief discussion and critical analysis of two closed source implementations of SSE.

At the conclusion of the Literature Review, the following Research Questions are identified:

- RQ1: *How Efficient Is Searchable Symmetric Encryption (SSE) When Implemented And Deployed In A Cloud Environment?*
- RQ2: *What Is The Performance Cost Of Preserving Data/Query Privacy Using Searchable Symmetric Encryption (SSE) When Compared To Plaintext Information Retrieval (IR)?*

With a view to providing an answer to the aforementioned Research Questions, the author developed two software artefacts: ‘PlainTXT Stroage and Search Engine’ – an implementation of a plaintext IR System, and ‘CipherTXT Storage and Search Engine’ – an implementation of an SSE system.

Chapter 3 outlines the Software Requirements Specification for both software artefacts produced as part of this dissertation; Chapter 4 outlines the Design details of same, while Chapter 5 outlines the Implementation details of same.

Chapter 6 comprises the Test Results obtained from both software artefacts, while Chapter 7 constitutes an evaluation of both the software artefacts and their associated Test Results.

Chapter 8 comprises the set of Conclusions derived from this dissertation, as well as areas of potential future Research.

2. Literature Review

This Chapter introduces the concept of Searchable Encryption in the context of a Literature Review.

Searchable Symmetric Encryption (SSE) has its roots in plaintext searching, although symmetrically encrypted ciphertext cannot be searched in the same manner; nonetheless, many of the principles that apply to plaintext searching also apply to SSE.

In its most basic form, SSE is nothing more than an Inverted Index – a mechanism that has been used in plaintext Information Retrieval (IR) for decades - that has been modified and adapted for use with ciphertext.

Initial discussions of the Inverted Index centre on an explanation of its application to plaintext Information Retrieval. This is followed by a discussion of the how the Inverted Index has been adapted and modified for use with ciphertext.

Consider the following example application in the context of plaintext:

A Client has outsourced storage of a Document Collection to a Server. Should the Client wish to retrieve those Documents from the Server that contain a specific Search String, the Client simply forwards the Search String in question to the Server. In turn, the Server processes the Search String against the Document Collection and responds to the Client with those Documents that contain the specified Search String; *that is, Search Results.*

2.1 Searchable Encryption: An Overview

Fifteen years ago, Song *et al.* (2000) first proposed the concept of Searchable Encryption. When explaining the basic operation of Searchable Encryption, Song *et al.* (2000) did so using an example whereby the content of symmetrically encrypted Documents were sequentially searched; *that is, character-by-character, word-by-word*, for the presence of a user specified Search String. Prior to the Search taking place, the Search String specified by the user was first encrypted using the same key used to encrypt the Documents being searched, with the resulting value – referred to as the Encrypted Search String (ESS) – being the value Searched for within the encrypted Documents. Those encrypted Documents deemed to contain the ESS were then returned to the user as part of the subsequent Search Results (see Figure 2).

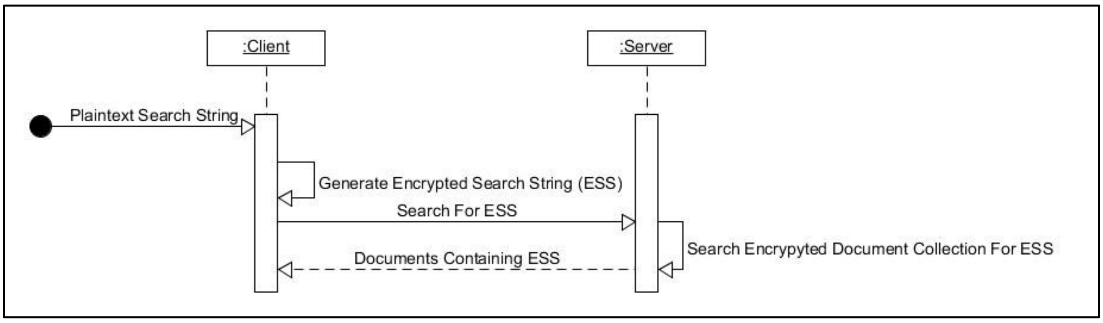

:Client
:Server
Plaintext Search String
Generate Encrypted Search String (ESS)
Search For ESS
Search Encrypyted Document Collection For ESS
Documents Containing ESS

Figure 2: Original Description of Searchable Encryption

While this explanation successfully communicated the basic premise of Searchable Encryption – in a manner relatively similar to plaintext Information Retrieval (IR); *that is, plaintext searching* - it nonetheless ignored the fact that modern symmetric ciphers do not support Searchable Encryption as described by Song *et al.* (2000).

Specifically, modern symmetric ciphers implement Shannon's Confusion and Diffusion principles (through the use of Substitution-Permutation networks) to counter cryptanalysis (Stallings, 2014, p.66-67). As a consequence, Searchable Encryption - as described by Song *et al.* (2000) - is not feasible[6].

The description of Searchable Encryption provided by Song *et al.* (2000) operates on the assumption that a given Term - whether in plaintext form or encrypted form - is located in the same position in both the plaintext version of the Document and the encrypted version of the same Document. *For Example: Given a plaintext Document beginning with the Term 'The', the description provided by Song et al. (2000) assumes that the first three characters of both the plaintext version of the Document and the encrypted version of the Document correspond to the Term 'The'*. Essentially this description assumes that symmetric ciphers encrypt data one character at a time, when in reality, this is not the case.

[6] In their paper, Song *et al.* (2000) demonstrate that Searchable Encryption can in fact be achieved as originally described; albeit only in the case of a single highly unsecure scenario. The scenario in question requires that data be encrypted using Electronic Code Book mode (the use of ECB mode is highly discouraged due to its susceptibility to cryptanalysis) and that the author of the Document being searched limit the maximum length of plaintext Terms in the Document to the Block Size of the associated cipher (8 characters in the case of DES, 16 characters in the case of AES). In addition, the scenario also requires that the author of the Documents ensure that only a single Term is contained within each ciphertext Block.

For Example, consider a Document due to be encrypted using AES (16 byte Block Size). Should the Document in question begin with the word 'The' (3 characters), the author of the Document would have to ensure that 13 whitespaces appeared after the word 'The' in order to ensure that only a single Term was contained within the first ciphertext block within the Document (16 characters in total).

Modern symmetric ciphers encrypt data in blocks of a fixed size, rather than character by character (Stallings, 2014, p.62). The effect of using such ciphers is that the ciphertext associated with a given plaintext Term is spread across the entire ciphertext block, rather than appearing in the same position as the plaintext Term; thus preventing traditional Sequential Searching (Stallings, 2014, p.63). In addition, modern symmetric ciphers typically operate using advanced block cipher modes (another mechanism to counter cryptanalysis) which 'chain' the ciphertext of previously encrypted blocks to the current plaintext block (by means of a bitwise XOR operation) (Stallings, 2014, p.183-198); thus further complicating the problem of searching ciphertext for the presence of an encrypted version of a plaintext Search String[6].

Recognising the inherent difficulty in achieving Searchable Encryption as originally described by Song *et al.* (2000), subsequent work in the area focussed on developing solutions to the problem as originally conceived; albeit without actually using Sequential Searching (Goh, 2003). Specifically, researchers focussed on adapting the Inverted Index – a mechanism that has been used in plaintext Information Retrieval for decades – for use in Searchable Encryption (Curtmola *et al.,* 2006).

In its most basic form, an Inverted Index is a Data Structure that maps Terms to the Document(s) they occur in; therefore eradicating the need to Sequentially Search Documents (Luenberger, 2006, p.285; Manning *et al.,* 2008, p.6). When adapted for use with an encrypted Document Collection, the resulting Inverted Index is titled

Searchable Symmetric Encryption (SSE) (Curtmola *et al.,* 2006)- the topic of focus for this dissertation.

2.2 Information Retrieval and the Inverted Index

Unlike searching a Collection of encrypted Documents, searching a Collection of plaintext Documents for the presence of a user specified Search String is a trivial process.

The most basic method of doing so, known as Sequential Searching, involves examining each Document within a Collection on a Term by Term basis. As each Term within the Document being examined is encountered, the Term in question is simply compared to the user specified Search String for equality (assuming the Search String in question consists of a single Term). In the event that a Document Term matches the user specified Search String, the associated Document is then returned to the user as part of the ensuing Search Results (Manning *et al.,* 2008, p.3).

While Sequential Searching functions effectively, its search efficiency is poor: Sequential Searching suffers from the fact that each Document in the Collection must be examined; therefore making its search time linear in the number of Documents contained within the Collection. As such, the time taken to search the Collection increases as the number of Documents in the Collection expands.

The poor performance of Sequential Searching can be directly attributed to the fact that the set of Terms contained within each Document must be determined at run time; *that is, while the Search is being conducted*. In addition, the set of Documents that a Search String occurs in must also be determined at run-time; hence why each Document within the Collection must be examined (Manning *et al.,* 2008, p.3-4).

In an effort to expedite the process of plaintext Information Retrieval (IR), the Inverted Index was developed. Just like a Database Index is designed to speed up data retrieval without searching each row of a Database Table, the Inverted Index is designed to speed up IR without having to search each Document within a Collection.

In its most basic form, an Inverted Index is a Data Structure that maps each Term within a Collection to the Document(s) it occurs in (Luenberger, 2006, p.6).

The Inverted Index attempts to overcome the shortcomings of Sequential Searching by pre-computing the list of all Terms contained within a Document Collection, as well as each pre-computing what Document(s) each Term occurs in; *that is, in advance of a search occurring*. The purpose of pre-computing this list of Terms – commonly referred to as the *Lexicon* of the Collection – is that the list of Terms is searched for the presence of the user specified Search String, instead of the Document Collection; thus making the search time linear in the number of Terms contained within the Collection.

For improved search efficiency, it is common for the Lexicon to be stored using Data Structures that expedite searching, such as Hash Tables (*O(1) Search Complexity*), Binary Search Trees (*O(Log N) Search Complexity*), B-Trees (*O(Log N) Search Complexity*) or Word Tries (*O(Log N) Search Complexity*).

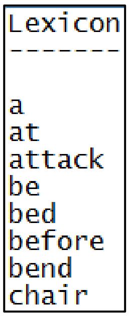
Section 2.2.1 outlines how the Inverted Index is constructed for usage in plaintext IR, while Section 2.2.2 outlines how same is queried.

2.2.1 Inverted Index Construction

Construction of an Inverted Index first requires a Document Collection from which the Inverted Index will be built; *that is, the Document Collection to be searched*. Inverted Index construction begins with each Document within the Collection being sequentially scanned by the Server[7], and a note being made of each Term that occurs within each Document. This process is typically referred to as *Document Tokenisation* (Manning *et al.,* 2008, p.22)).

Each and every Term encountered during Document Tokenisation is added to a list known as the *Lexicon* (see Figure 3). Essentially, the Lexicon is the list of all Terms that occur in a given Document Collection (Luenberger, 2006, p.285; Manning *et al.,* 2008, p.6). In the event of the same Term occurring multiple times in a single Document, or the same Term occurring in multiple Documents within the Collection– both of which are inevitable - the Term in question appears only once in the Lexicon; therefore, each Term contained within the Lexicon is unique (Luenberger, 2006, p.286-287; Manning *et al.,* 2008, p.22).

[7] In SSE, the Client is responsible for constructing the Inverted Index; **not** the Server as is the case with plaintext Information Retrieval (IR).

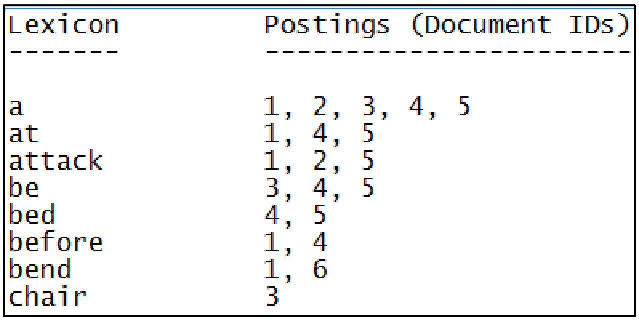


```
Lexicon
-------

a
at
attack
be
bed
before
bend
chair
```

Figure 3: Sample Lexicon.

Throughout the process of Document Tokenisation, each and every Document that a given Term occurs in is also noted; *that is, the Document ID is noted*[8] (see Figure 4). The noting of a given Term occurring in a given Document is referred to as a *Posting*, while the list of all Documents; *that is, Document ID's*, where a given Term occurs is referred to as a *Posting List* (Manning *et al.,* 2008, p.6).

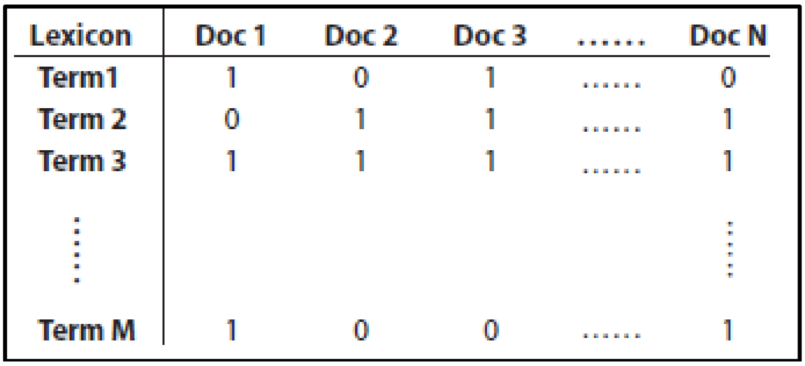


```
Lexicon          Postings (Document IDs)
-------          -----------------------

a                1, 2, 3, 4, 5
at               1, 4, 5
attack           1, 2, 5
be               3, 4, 5
bed              4, 5
before           1, 4
bend             1, 6
chair            3
```

Figure 4: Sample Lexicon (Including Postings/Posting Lists).

[8] As part of the process of Inverted Index Construction, each Document within the Collection is assigned a unique identifier known as a Document ID.



Lexicon	Doc 1	Doc 2	Doc 3	......	Doc N
Term1	1	0	1	......	0
Term 2	0	1	1	......	1
Term 3	1	1	1	......	1
⋮					⋮
Term M	1	0	0	......	1

Figure 5: Tabular Visualisation of an Inverted Index (Luenberger, 2006, p.287).

Figure 5 depicts a simple tabular visualisation of an Inverted Index. The Lexicon for the Document Collection is listed in the left most column of the table, while the list of Documents within the Collection; *that is, Documents IDs* is listed along the top row of the table. The intersection of each row and column contains a value denoting whether or not the Term associated with the row in question occurs within the Document associated with the column in question, with '1' denoting the occurrence of the Term within the Document; *that is, a Posting*, and '0' denoting the absence of the Term from the Document[9] (Luenberger, 2006, p.285-286; Manning *et al.,* 2008, p.3-4).

[9] In reality, a fully functioning IR System would only record the occurrence of a Term within a Document, and not the non-occurrence of a Term (doing so would be hugely wasteful in terms of memory). The inclusion of non-occurrence information in the table in Figure 5 is merely for explanatory purposes.

Regarding implementation details, an Inverted Index can be implemented in a number of ways:

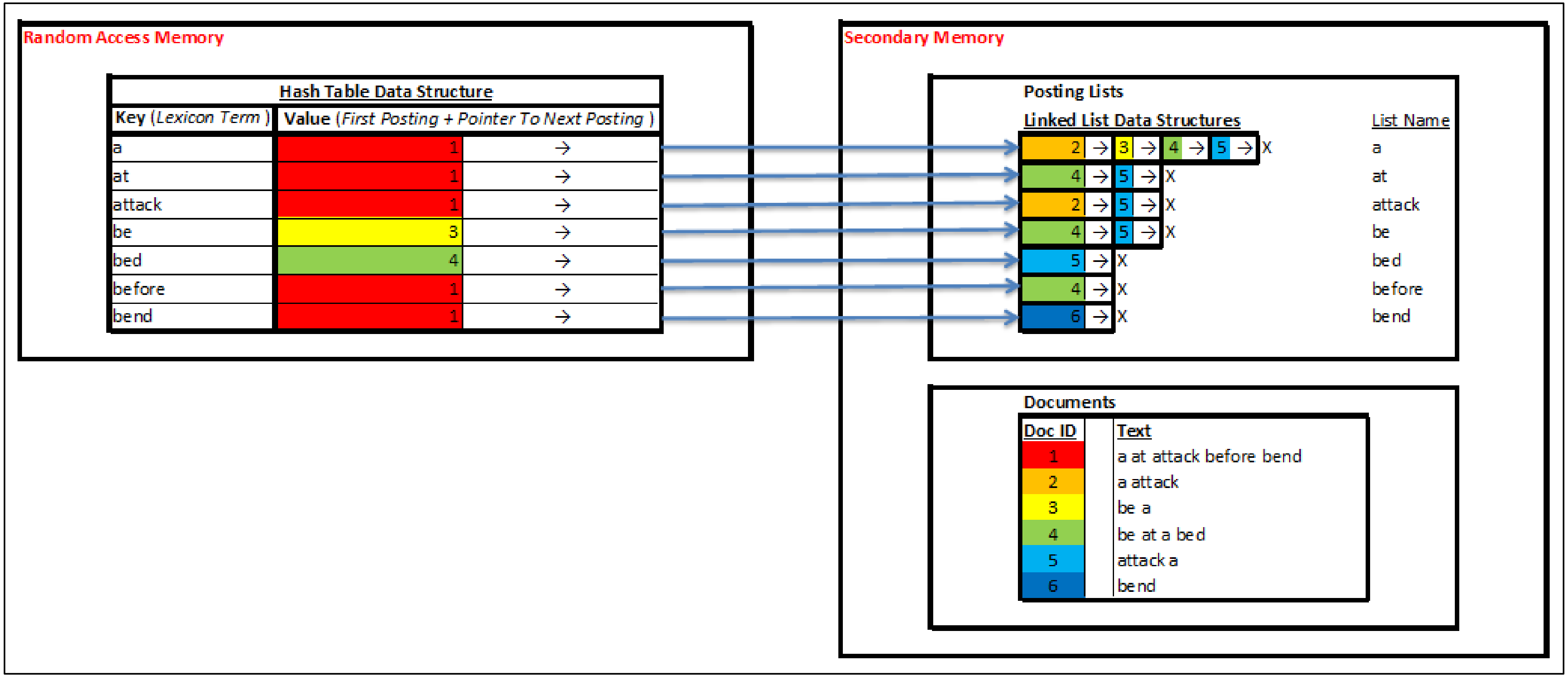
- As mentioned previously, the Lexicon can be implemented and stored using a number of different Data Structures. This dissertation assumes that the Data Structure used is a Hash Table (due to its efficiency; *that is, O(1)*).
- Due to their list-like nature, Posting Lists are typically implemented using a Linked List Data Structure. Alternative Data Structures, such as Arrays, can be used; however the dynamic nature of Posting Lists often makes the Linked List Data Structure the preferred choice.
- In relation to Document storage, Documents can be stored in a number of ways. Primarily, Documents are either stored using the native file system of the Server they are stored on, or alternatively, as Rows within a Database Table (with their designated Document ID acting as their Primary Key value).

In term of memory management, the Lexicon of an Inverted Index is typically loaded into Random Access Memory (RAM) at all times. Given that the Lexicon contains the information to be searched whenever an Inverted Index is queried; it is therefore common that a significant amount of RAM be allocated to same.

Regarding memory management for Posting Lists and Documents, both sets of information are typically stored in secondary memory. This is due to the fact that both Posting Lists and Documents are only ever retrieved whenever their associated Terms are searched for.

For improved performance, it is common for the first Link of a Postings List Linked List to be stored alongside its associated Term in the Lexicon Data Structure; *that is, in RAM*. This is due to the fact that the first Link in a Linked List is required to access all subsequent Links in the Linked List; *that is, all subsequent Postings* (stored in secondary memory) (Luenberger, 2006, p.289; Manning *et al.,* 2008, p.7).

Figure 6 provides a simple visualisation of the Inverted Index as typically utilised in plaintext Information Retrieval (IR). Incorporated within the Figure is the Data Structures commonly used to store the three data sets that make up the Inverted Index, as well as what form of computer memory is typically used to store each Data Structure.



Hash Table Data Structure

Key (*Lexicon Term*)	Value (*First Posting + Pointer To Next Posting*)	
a	1	→
at	1	→
attack	1	→
be	3	→
bed	4	→
before	1	→
bend	1	→

Linked List Data Structures	List Name
2 → 3 → 4 → 5 → X	a
4 → 5 → X	at
2 → 5 → X	attack
4 → 5 → X	be
5 → X	bed
4 → X	before
6 → X	bend

Doc ID	Text
1	a at attack before bend
2	a attack
3	be a
4	be at a bed
5	attack a
6	bend

Figure 6: Inverted Index Visualisation (Including Data Structures and Memory Management).

2.2.2 Querying an Inverted Index

Performing a Query against the Inverted Index structure described in Section 2.2.1 is relatively simple.

Given a Search String, the Lexicon Data Structure is examined to determine the presence or absence of the Search String within the Lexicon.

In the event that the Search String is present in the Lexicon, the first Posting associated with the matching Term is retrieved from RAM. In turn, this Link is then used to retrieve all subsequent Links in the Linked List (stored in secondary memory); thus retrieving all Postings for the Search String in question. Once the Posting List has been retrieved in full, the associated Document IDs are then used to retrieve the actual Documents – from secondary memory – that contain the Search String. Once all Documents are retrieved, they are then forwarded to the Client; *that is, Search Results*.

In the event that a Search String is not present in the Lexicon, this denotes that the Search String in question is not present in any Documents contained within the Collection (Luenberger, 2006, p.293; Manning *et al.,* 2008, p.10).

2.3 Searchable Symmetric Encryption (SSE)

To ensure clarity, the author refers to the Inverted Index structure used in plaintext Information Retrieval (IR) as the IR Inverted Index, while its SSE counterpart is referred to as the SSE Inverted Index.

As the name suggests, the SSE Inverted Index borrows heavily from the IR Inverted Index. All information presented previously in relation to the IR Inverted Index remains true for the SSE Inverted Index; however the reader should be aware that SSE and the SSE Inverted Index differ from IR and the IR Inverted Index in the following ways:

The topic of Information Leakage forms an Integral part of SSE. When the idea of Searchable Encryption was first proposed, one of its founding principles was the assumption that the Server storing the encrypted Document Collection is an adversary that is actively working on subverting the security of the Document Collection it possesses (with the ultimate goal of gaining access to the Document Collection in plaintext form) (Song *et al.,* 2000). As such, the SSE Inverted Index is constructed and operates in a manner that takes significant steps to reduce the Leakage of potentially useful Information to the Server. In practice, this involves the use of encryption for the Document Collection, the Lexicon, Posting Lists and Search Strings; as well as the use of Data Structures that hinder the Servers efforts in achieving its malicious goals (Goh, 2003; Chang and Mitzenmacher, 2005; Curtmola *et al.,* 2006).

Responsibility for creating the SSE Inverted Index is offloaded to the Client. In order for the Server to construct the SSE Inverted Index, decryption keys must be disclosed to the Server (as mentioned previously, this is undesirable from a data security perspective). Rather than reveal sensitive information to the Server, SSE delegates responsibility of constructing the SSE Inverted Index to the Client. Given that the Client is responsible for constructing the SSE Inverted Index, it is therefore expected that the Client forwards the SSE Inverted Index to the Server along with the encrypted Document Collection whenever the latter is forwarded to the Server for storage (Goh, 2003).

Subsection 2.3.2 provides an overview of how the SSE Inverted Index is constructed, while Subsection 2.3.3 provides an overview of how same is queried[10]. Prior to the discussing both, a brief overview of Information Leakage is first given in Subsection 2.3.1.

2.3.1 Information Leakage

2.3.1.1 A Basic Overview

A significant portion of the Searchable Encryption literature has focussed on determining what Information Leakage results from a) The Server being in

[10] The SSE scheme discussed throughout this dissertation is that presented in Kamara *et al.* (2012), with the exception that support for updates to the Document Collection and the associated SSE Inverted Index are not included in this dissertation.

possession of the encrypted Document Collection, and b) The Server carrying out searches on same; *that is, a Client ordering the Server to perform a Search, or the Server itself carrying out searches covertly*. The purpose of studying such Information Leakage is to determine whether or not any and all Information Leaked by various Searchable Encryption schemes is useful to the Server in terms of achieving its malicious goal(s)[11].

Ideally, no Information Leakage should occur as a result of utilising Searchable Encryption; however, like all ideal scenarios, realising it is not without its challenges. The two most secure forms of Searchable Encryption at present; *that is, Oblivious RAM (RAM) and Fully Homomorphic Encryption-2 (FHE-2)* (see Figure 1), achieve zero Information Leakage; however both do so at the expense of efficiency. In both solutions, this poor efficiency can be directly attributed to the Information Leakage countermeasures utilised (Chunsheng, 2011; Stefanov *et al.,* 2013; Hahn and Kerschbaum, 2014).

In an effort to improve the overall efficiency of Searchable Encryption, several researchers have examined the prospect of relaxing the zero-tolerance approach to Information Leakage in Searchable Encryption (Goh, 2003; Chang and Mitzenmacher, 2005; Curtmola *et al.,* 2006; Kamara *et al.,* 2012; Cash *et al.,* 2013). Specifically, researchers have attempted to determine what Information Leakage is

[11] All SSE schemes – with the exception of Song et al. (2000) – included full discussions of Information Leakage, as well as full Proofs of Security (Bosch, 2014).

acceptable in Searchable Encryption (sometimes referred to as Trivial Information Leakage); *that is, Information that in no ways aids the Server in achieving its goal of subverting the encrypted Document Collection*. Evidently, the goal of this Research was to identify which Information Leakage countermeasures are absolutely necessary in Searchable Encryption; therefore allowing researchers to focus on creating search efficient schemes that conform to this baseline measure of Information Leakage (at a minimum) .

2.3.1.2 Information Leakage in Searchable Symmetric Encryption

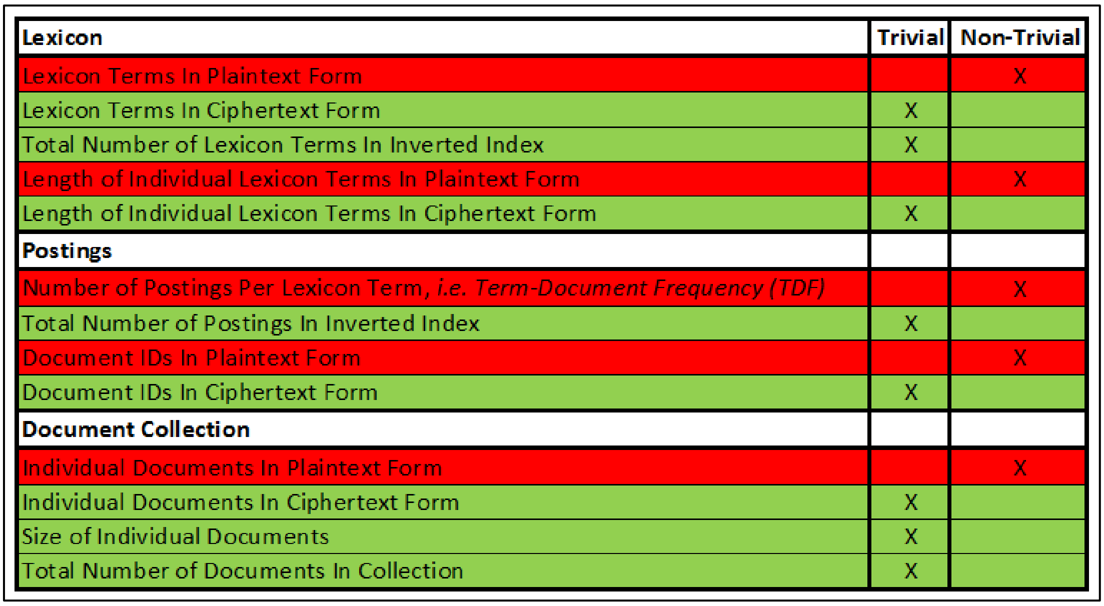
In the case of Searchable Symmetric Encryption (SSE), it is the SSE Inverted Index that is searched by the Server, instead of the encrypted Document Collection. As such, the SSE literature instead focusses on determining what Information Leakage results from the Server being in possession of the SSE Inverted Index, as well as what Information Leakage results from the Client (or the Server itself) querying same.

In terms of Information Leakage in SSE, such Information Leakage is typically broken into three categories: Storage Leakage; *that is, what the Server can learn (or deduce) from the SSE Inverted Index by simply storing it (that is, without the SSE Inverted Index actually being queried)*, Query Leakage; *that is, what the Server can learn (or deduce) from the SSE Inverted Index by querying it itself (covertly) , or observing the SSE Inverted Index being queried by Client(s)*, and Update Leakage; *that is, what the Server can learn (or deduce) whenever the SSE Inverted Index is updated (such as when Documents within the Collection are edited/deleted, or* *when new Documents are added to the Collection)* (Curtmola *et al.,* 2006; Chase and Kamara, 2010; Kamara *et al.,* 2012; Kamara, 2013).

As part of this dissertation, the author has attempted to produce an all-encompassing list of Information that can potentially be Leaked by SSE, as well as highlighting what Information Leakage is classified as Trivial and Non-Trivial by the Searchable Encryption literature (Curtmola *et al.* 2006, Cash *et al.* 2013). Figure 7 presents potential Storage Leakage by an SSE scheme (including an indication of whether or not such Information is classified as Trivial Information Leakage or Non-Trivial Information Leakage), while Figure 8 presents potential Query Leakage by an SSE scheme.

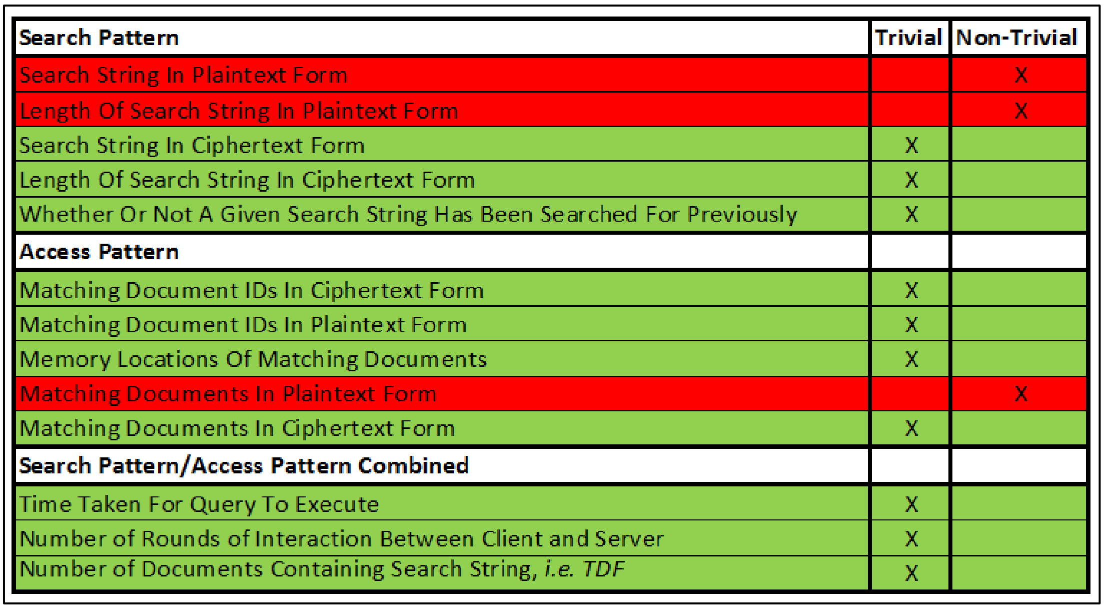
Note that potential Storage Leakage is presented in terms of the three inter-related data sets that make up an SSE Inverted Index; *that is, the Lexicon, Postings and the associated encrypted Document Collection*, while potential Query Leakage is presented in terms of the Search Pattern; *that is, the Encrypted Search String (ESS) received by the Server (and whether or not the ESS was utilised before)*, and the Access Pattern; *that is, those encrypted Document(s) deemed to contain the ESS, their associated memory location(s) and their Document ID(s)*.

In the description of SSE that follows in Section 2.3.2 and Section 2.3.3, all Information classified as Non-Trivial in Figure 7 (Potential Storage Leakage) and Figure 8 (Potential Query Leakage) is Leaked to the Server.



Lexicon	**Trivial**	**Non-Trivial**
Lexicon Terms In Plaintext Form		X
Lexicon Terms In Ciphertext Form	X	
Total Number of Lexicon Terms In Inverted Index	X	
Length of Individual Lexicon Terms In Plaintext Form		X
Length of Individual Lexicon Terms In Ciphertext Form	X	
Postings		
Number of Postings Per Lexicon Term, *i.e. Term-Document Frequency (TDF)*		X
Total Number of Postings In Inverted Index	X	
Document IDs In Plaintext Form		X
Document IDs In Ciphertext Form	X	
Document Collection		
Individual Documents In Plaintext Form		X
Individual Documents In Ciphertext Form	X	
Size of Individual Documents	X	
Total Number of Documents In Collection	X	

Figure 7: Potential Storage Leakage in SSE (Including Trivial Leakage and Non-Trivial Leakage).



Search Pattern	**Trivial**	**Non-Trivial**
Search String In Plaintext Form		X
Length Of Search String In Plaintext Form		X
Search String In Ciphertext Form	X	
Length Of Search String In Ciphertext Form	X	
Whether Or Not A Given Search String Has Been Searched For Previously	X	
Access Pattern		
Matching Document IDs In Ciphertext Form	X	
Matching Document IDs In Plaintext Form	X	
Memory Locations Of Matching Documents	X	
Matching Documents In Plaintext Form		X
Matching Documents In Ciphertext Form	X	
Search Pattern/Access Pattern Combined		
Time Taken For Query To Execute	X	
Number of Rounds of Interaction Between Client and Server	X	
Number of Documents Containing Search String, *i.e. TDF*	X	

Figure 8: Potential Query Leakage in SSE (Including Trivial Leakage and Non-Trivial Leakage).

From examining Figure 7 and Figure 8, it is evident that Leaking Information to the Server in ciphertext form is considered Trivial Information Leakage by the SSE

Literature – irrespective of whether such ciphertext is Leaked to the Server as part of Setup Leakage or Query Leakage.

As regards plaintext Information Leakage, such Leakage is *generally* considered Non-Trivial Information Leakage by the SSE Literature; however, a notable exception to this rule is Document IDs. From Figure 7 (Postings Section), it is noticeable that the Leakage of Document IDs in plaintext form is considered Non-Trivial Information Leakage in the case of Setup Leakage; while the exact same Information is classified as Trivial Information Leakage in the case of Query Leakage (Figure 8 – Access Pattern).

In addition to considering Information Leakage in the context of plaintext Information and ciphertext Information, the literature also considers Information Leakage from a statistical point of view; *that is, those statistics that be derived from Information, irrespective of whether the underlying Information is in plaintext form or ciphertext form*.

Generally, the SSE literature classifies statistical Information Leakage as Trivial Information Leakage; however one exception does exist. The statistic in question – known as Term-Document Frequency (TDF)*; that is, the number of Documents containing a given Term* - is classified as Non-Trivial Information Leakage in the case of Setup Leakage (see Postings in Figure 7); and Trivial Information Leakage in the case of Query Leakage (see Access Pattern in Figure 8) (much like Document IDs as discussed previously).

Admittedly, the decision to label certain Information Leakage as Non-Trivial for Setup Leakage and the decision to label the exact same Information Leakage as Trivial for Query Leakage appears bewildering. Nonetheless, this can be explained by the conservative approach to Information Leakage taken by researchers in the area. Generally, where certain Information Leakage is considered unavoidable (and the Information in question is classified as Trivial Information Leakage), researchers take the approach of allowing such information to be Leaked; however, rather than Leak such Information immediately; *that is, Storage Leakage*, researchers will typically guard such Information up to the point where its Leakage is absolutely necessary and therefore unavoidable (otherwise known as Controlled Disclosure); *that is, Query Leakage* (Curtmola *et al.,* 2006; Chase and Kamara, 2010; Cash *et al.,* 2013).

2.3.1.3 Security Definitions

A number of security definitions have been proposed for Searchable Encryption, the most prevalent of which is IND-CKA2 (Indistinguishable Against Adaptive Chosen Keyword Attacks) (Curtmola *et al.,* 2006; Bosch *et al.,* 2014).

Essentially, IND-CKA2 security requires that an adversary (in this case, the Server) learn nothing about the underlying Document Collection/SSE Inverted Index beyond the Search String - in ciphertext form - and the associated Search Results - in ciphertext form; *that is, Trivial Information Leakage*, even when the SSE scheme can be *adaptively* queried by the Server. Numerous SSE schemes have been developed that adhere to the IND-CKA2 notion of security (including the SSE discussed in Section 2.3.2 and Section 2.3.3)

Stronger definitions of Security do exist in the literature (Shen *et al.,* 2008); however such schemes are typically inefficient due to their refusal to allow Information Leakage in any form (Shen *et al.,* 2008; Bosch *et al.,* 2012).

2.3.2 SSE Inverted Index Construction

The steps involved in constructing an SSE Inverted Index are exactly the same as those involved in constructing an IR Inverted Index, albeit the Client has responsibility for generating the SSE Inverted Index, and various forms of encryption are applied to each dataset after they have been compiled; *that is, the Document Collection, the Lexicon and the Postings List (Goh, 2003)*.

In addition to the use of encryption, a different Data Structure – namely, an Array - is utilised to store Postings instead of a Linked List (as is used in the IR Inverted Index) (Curtmola *et al.,* 2006).

2.3.2.1 Lexicon

Rather than storing Lexicon Terms in plaintext form, SSE requires that a keyed-hash[12] of each Term be stored instead (Chase and Kamara, 2010; Kamara *et al.,* 2012).

The use of a keyed hash function for this purpose - instead of traditional reversible encryption - may seem curious at first; however researchers have successfully argued that the Lexicon's sole purpose within the Inverted Index is to provide the Client with the ability to carry out searches and nothing more. Given that the Lexicon is unlikely to be downloaded to the Client (and is therefore unlikely to be decrypted - unlike the actual Documents), the use of reversible encryption for encrypting Lexicon Terms has largely been abandoned (Chase and Kamara, 2010; Kamara *et al.,* 2012).

[12] The use of a Keyed Hash function for this purpose implicitly limits the number of Lexicon Terms to the maximum number of unique Hash Values produced by the associated Hash function. Should the number of Lexicon Terms exceed the maximum number of Hash Values, a Hash Collision will most definitely occur (and may even occur before this point is reached), leading to a scenario whereby two distinct plaintext Lexicon Terms have the same Hash Value; therefore leaving the Server unable to distinguish which Lexicon Term is being searched for. While Hash Collisions are unlikely for small data sets, they remain a distinct possibility for large data sets.

Aside from the aforementioned reasons, the use of a keyed hash function for this purpose has a number of advantages in terms of reduced Information Leakage and improved data security, including the following (Stallings, 2014, p.368-372):

- First and foremost, the use of a hash function (keyed or non-keyed) ensures that all encrypted Lexicon Terms within the SSE Inverted Index are of equal length (*a hash function produces a Hexadecimal String of fixed length*); therefore masking the length of all underlying plaintext Lexicon Terms.
- Secondly, the use of a hash function (again, keyed or non-keyed) ensures that an adversary has no means of decrypting the encrypted Lexicon Term back to its plaintext form.
- Thirdly, ensuring that a keyed hash function is used – instead of a traditional non-keyed hash function – protects SSE from Rainbow Table Attacks; *that is, pre-computed Hash Values of common Dictionary Words*.

As regards keyed-hash algorithms, the SSE literature states that any standardised secure algorithm can be utilised for Lexicon Encryption (*For Example: HMAC-MD5, HMAC-SHA256*) (Chase and Kamara, 2010; Kamara *et al.,* 2012; Cash *et al.,* 2013).

2.3.2.2 Postings List

The use of Linked Lists for Posting List storage is abandoned in SSE due to Setup Leakage resulting from their modus operandi; *that is, sequential memory access*, with Arrays being preferred instead (Curtmola *et al.,* 2006).

Specifically, given the first Link in a Linked List, it is a trivial process to examine all subsequent Links due to the fact that each Link in a Linked List contains a pointer to the next Link (see Figure 9). Given that each Term in an IR Inverted Index has its own dedicated Linked List to store Postings; it is therefore a trivial process to derive the Term-Document Frequency (TDF) for each Term in the Lexicon in advance of the associated Term being searched for (*Recall from Subsection 2.3.1.2 that TDF Leakage is considered Non-Trivial Information Leakage for Storage Leakage*) (Luenberger, 2006, p.243; Chase and Kamara, 2010).

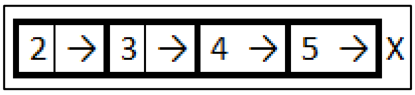
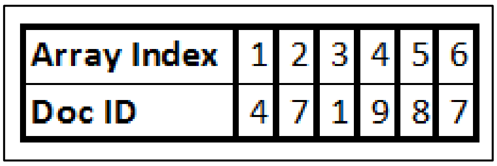


Figure 9: Linked List Data Structure (→ Denotes A Pointer to the Next Link in the Linked List).

Rather than using one Array for each Term in the Lexicon (doing so would also result in TDF Storage Leakage; *that is, the size of the Array would be equivalent to the TDF*), SSE utilises a single one dimensional Array to store all Postings for all Terms (see Figure 10). Utilising this approach, Setup Leakage amounts to the total number of Postings for the entire Lexicon*; that is, trivial Leakage*.



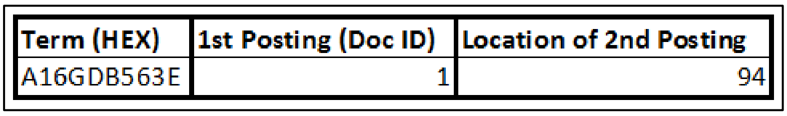
Array Index	1	2	3	4	5	6
Doc ID	4	7	1	9	8	7

Figure 10: Postings Stored In an Array.

Given that all Postings are now stored in a single one dimensional Array, some mechanism to keep track of what Postings belong to what Terms is therefore required. The solution to this problem is –ironically enough – relatively similar to a

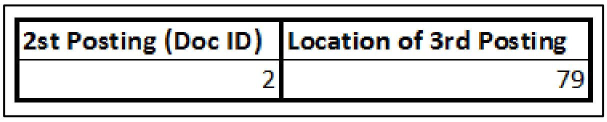
Linked List, albeit the solution involved does not utilise pointers (as is the case with Linked Lists).

In order to keep track of what Postings are associated with a given Term, SSE requires that the Document ID of the first Posting associated with a given Term is stored alongside the keyed-hash of the Term in the Lexicon Hash Table (in RAM) (*For Example: Doc ID 1*). Alongside this Document ID (in the Lexicon Hash Table) is an Array Index denoting the location of the second Posting associated with the Term (see Figure 11) (*For Example: 94*). At the Array Index in question is the Document ID of the 2nd Posting, as well as the Array Index denoting the location of the third Posting (*For Example: 79*) (see Figure 12) [13].



Term (HEX)	1st Posting (Doc ID)	Location of 2nd Posting
A16GDB563E	1	94

Figure 11: SSE Lexicon Node.



2st Posting (Doc ID)	Location of 3rd Posting
2	79

Figure 12: Postings Array Node.

Rather than storing all Postings sequentially within the Array, SSE requires that all Postings be shuffled to random locations within the Postings Array. As such, the

[13] Essentially, the first Posting is stored alongside its associated Term in RAM, with all other subsequent Postings being stored in the Postings Array. Alongside each Posting is the Array Index denoting the location of the next Posting.

second Posting for a Term may be located at Array Index 1000, while the third Posting may be located at Array Index 1.

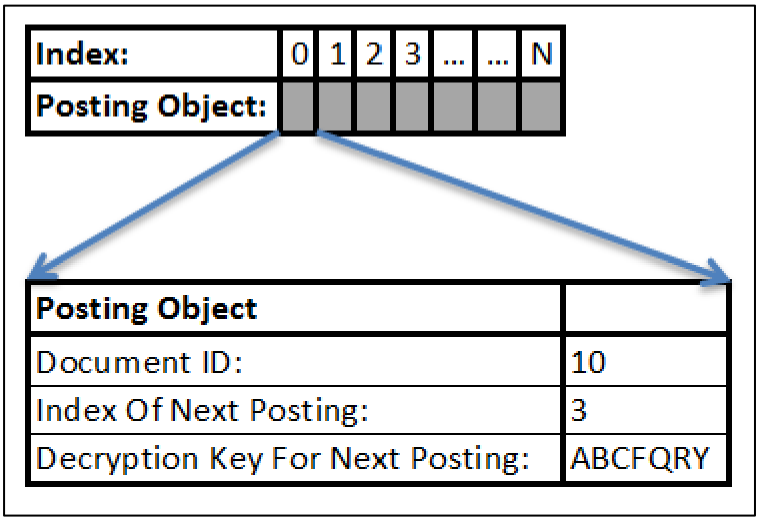
Despite utilising Arrays and arranging Postings in non-sequential order, the fact remains that the Information stored at each Index of the Postings Array is in plaintext form. As such, it is still a trivial process for the Server to calculate the TDF for each Term in the Lexicon in advance of the Term being searched for (as was the case previously with Linked Lists).

As a solution to this problem, SSE requires that each Document ID within the Postings Array be encrypted, as well as each 'Next Posting Location'; therefore preventing the Server from deducing this Information by merely being in possession of the SSE Inverted Index; *that is, Storage Leakage*.

Rather than encrypting all Postings using the same key, SSE requires that each Posting be encrypted using a different key. In an effort to reduce the number of keys the Client has to remember in order to utilise SSE, the literature recommends the following guidelines for encrypting the Posting Array:

1. The encryption/decryption key for the first Posting associated with each Term should be derived by passing its associated plaintext Term through a keyed hash function (the key utilised within the keyed hash function is a master key known only to the Client).
2. All subsequent Postings in the Array; *that is, 2nd Posting, 3rd Posting, 4th Posting, etc.*, are to be encrypted/decrypted using randomly generated encryption keys.

3. The key required to decrypt a given Posting (with the exception of the first Posting) is to be stored in the previous Posting associated with the Term in question (see Figure 13)[14].



Index:	0	1	2	3	…	…	N
Posting Object:							

Posting Object	
Document ID:	10
Index Of Next Posting:	3
Decryption Key For Next Posting:	ABCFQRY

Figure 13: Postings Array Node (Including Decryption Key Storage).

As regards encryption algorithms, the SSE literature states that any standardised secure symmetric algorithm can be utilised for Posting/Posting List Encryption (*For Example: AES, Triple DES*) (Curtmola *et al.,* 2006; Chase and Kamara, 2010; Kamara *et al.,* 2012; Cash *et al,* 2013).

[14] The decision to store Posting decryption keys on the Server side may seem obscure at first however the reader should recall that all Postings associated with a given Lexicon Term are 'chained together' throughout the Postings Array.

The first Posting in a Posting List is encrypted using a key generated by the Client, *i.e. Posting List Encryption - Step 1*. Whenever the Client searches for the associated Lexicon Term, the Client must also forward the key necessary to decrypt the first Posting, *i.e. Posting List Encryption - Step 1*. Decrypting the first Posting then reveals the key necessary to decrypt the following Posting, *i.e. Posting List Encryption – Step 2 and Step 3*. Without knowledge of the key required to decrypt the first Posting associated with a given Lexicon Term (Step 1), the Server is unable to decrypt subsequent Postings associated with the same Lexicon Term (Step 2 and Step 3).

2.3.2.3 Documents

As regards encryption algorithms, the SSE literature states that any standardised secure symmetric algorithm can be utilised for Document Encryption (*For Example: AES, Triple DES*) (Curtmola *et al.,* 2006; Chase and Kamara, 2010; Kamara *et al.,* 2012; Cash *et al.,* 2013).

2.3.2.4 Key Management

SSE requires the use of three encryption keys for SSE Inverted Index Construction and Querying. They are:

- One key for Lexicon Encryption/Searching (used to generate a keyed hash of each Lexicon Term/Search String)
- One master key used to derive encryption/decryption keys for the first Posting associated with each Lexicon Term.
- One key for Document Collection encryption/decryption (Curtmola *et al.*, 2006; Chase and Kamara, 2010; Kamara *et al.,* 2012; Cash *et al.,* 2013).

2.3.3 Querying an SSE Inverted Index

There are two types of SSE Schemes: Interactive; *that is, the Client and the Server exchange numerous messages before the Server responds with a set of Search Results*, and non-Interactive; *that is, the Client issues a Search String to the Server and the Server responds immediately with a set of Search Results (Bosch et al.,* *2014)*. The following description of querying an SSE Inverted Index covers the latter.

Given that the Lexicon of the SSE Inverted Index consists of a keyed-hash of each Term within the Document Collection, the Client is therefore required to generate a keyed-hash of their Search String in order to Query the Lexicon. The resulting Search String; *that is, an Encrypted Search String (ESS)*, is then forwarded to the Server (Chase and Kamara, 2010; Kamara *et al.,* 2012; Cash *et al.,* 2013). In addition to forwarding the ESS to the Server, the Client must also forward the decryption key necessary to decrypt the first Posting associated with the ESS (Curtmola *et al.,* 2006; Kamara *et al.,* 2012; Cash *et al.,* 2013).

In the event that the ESS is present in the Lexicon, the first Posting associated with the ESS is retrieved. The Server then proceeds to decrypt this information revealing the ID of the first Document containing the ESS, the Index Location of the second Posting, as well as the decryption key necessary to decrypt the information stored in the second Posting. This process then repeats until all Postings associated with the ESS have been retrieved and decrypted. Following this, the associated Document IDs are then used to retrieve the actual encrypted Documents – from secondary memory – that contain the ESS (Curtmola *et al.,* 2006; Kamara *et al.,* 2012; Cash *et al.,* 2013). Once all Documents are retrieved, they are then forwarded to the Client; *that is, Search Results (Song et al., 2000)*.

In the event that an ESS is not present in the Lexicon, this denotes that the ESS in question is not present in any Documents contained within the Collection (Luenberger, 2006, p.293; Manning *et al.,* 2008, p.10).

2.3.4 Implementations of SSE

Despite its efficiency, the fact remains that working implementations of SSE are few and far between. As part of this Literature Review, the author encountered a total of two papers discussing working implementations of SSE (Kamara *et al.,* 2012; Cash *et al.,* 2013); neither of which are available in the public domain.

Section 2.3.4.1 discusses the implementation of SSE developed by Kamara *et al.* (2012), while Section 2.3.4.2 discusses the implementation of SSE developed by Cash *et al.* (2013). This is followed by a critical analysis of both implementations in Section 2.3.4.3.

2.3.4.1 Kamara *et al.* (2012) Implementation

The implementation of SSE developed by Kamara *et al.* (2012) is a non-interactive, single Query Term SSE protocol. When compared to the description of SSE provided previously, the implementation by Kamara *et al.* (2012) is almost identical with the exception of the following:

- The implementation supports the addition and deletion of documents from the Document Collection (and therefore the SSE Inverted Index) – *The description of SSE previously assumed that the underlying Document*

Collection was static (updates to the SSE Inverted Index were not discussed as they were deemed beyond the scope of this dissertation).

- The implementation stores the entire Inverted Index; *that is, Lexicon and Posting Lists*, in RAM at all times.

In terms of programming languages, the implementation of SSE by Kamara *et al.* (2012) was developed using Microsoft C++.NET. Any and all cryptographic functionality associated with implementation employed the use of the Microsoft CNG library of cryptographic algorithms.

In relation to Data Structures, the exact Data Structure used for Lexicon Storage is not disclosed by Kamara *et al.* (2012). In the theoretical description of their scheme, Kamara *et al.* (2012) endorse the use of a 'Dictionary' Data Structure for Lexicon Storage; however the exact Data Structure used in the resulting implementation is not disclosed. Given that a number of Data Structures fall under the category of Dictionary Data Structures[15], the author can only speculate as to the exact Data Structure used. In terms of Posting List storage, Kamara *et al.* (2012) employ the use of a single one dimensional 'Array' Data Structure (as was the case with the description of SSE provided previously).

In terms of algorithms, the SSE implementation by Kamara *et al.* (2012) utilises 128 bit AES-CBC for Posting encryption, while HMAC-SHA256 is used for keyed hashing

[15] A Hash Tree Data Structure is classified as a Dictionary Data Structure.

of Lexicon Terms. The exact algorithm used for Document encryption is not disclosed.

In relation to Test Data, Kamara *et al.* (2012) tested their SSE implementation against three separate Test Data Sets: a subset of the Enron E-Mail Collection (16MB in size with approximately 1.5 million Postings), a collection of Microsoft Office Documents used by one of Microsoft's Business Groups (500MB in size with approximately 650,000 Postings), and a collection of Media Files (*For Example: MP3, WMA, JPG*) (500MB in size – number of postings not disclosed).

In terms of Research Results, the work of Kamara *et al.* (2012) focussed on two separate aspects of SSE: constructing the SSE Inverted Index, and querying the SSE Inverted Index.

For SSE Inverted Index construction, it should be noted that the Results presented by Kamara *et al.* (2012) only take into account the process of converting a pre-existing IR Inverted Index into an SSE Inverted Index and encrypting the associated Document Collection – the Results do not include the amount of time taken to generate the initial IR Inverted Index; nor do they take into account the time associated with transferring the SSE Inverted Index and the encrypted Document Collection from the Client to the Server. For the Enron E-Mail Test Data Set, constructing the associated SSE Inverted Index and encrypting the associated Document Collection took 52 seconds. For the Microsoft Office Document Collection Test Data Set, constructing the associated SSE Inverted Index and encrypting the associated Document Collection took approximately 33 seconds.

For SSE Inverted Index searching, it should be noted that the Results presented by Kamara *et al.* (2012) only take into account the process of retrieving and decrypting matching Postings from within an SSE Inverted Index that is permanently resident in RAM[16]– the Results do not include the amount of time taken to retrieve the SSE Inverted Index from secondary memory and loading it into RAM, nor do they include the amount of time taken to retrieve matching Documents from disk and returning them to the Client. In relation to SSE search time, it should be noted that search time is dependent on the number of matching documents associated with the search Term; *that is, the more frequent the Search Term appears in the Document Collection, the longer the associated Search operation will take*. As such, a common performance measure for the SSE Inverted Index is the amount of time taken to retrieve and decrypt the set of all Postings associated with the most commonly occurring Term within the Lexicon. In relation to the Enron E-Mail Test Data Set, retrieving the set of all Postings associated with the most commonly occurring Lexicon Term took 53 microseconds (µs), while identifying same took approximately 8 microseconds (µs) for the Microsoft Office Document Collection Test Data Set.

[16] On average, generating a keyed hash of a single Lexicon Term (either for SSE Inverted Index generation or searching an SSE Inverted Index) took 35 microseconds (µs) using the implementation developed by Kamara *et al.* (2012).

As regards hardware, the experiments conducted by Kamara *et al.* (2012) were performed on a Windows Server 2008R2 machine with an Intel Xeon L5520 Processor (2.26GHZ).

2.3.4.2 Cash *et al.* (2013) Implementation

The implementation of SSE developed by Cash *et al.* (2013) is a non-interactive, multiple Query Term SSE protocol. When compared to the description of SSE provided previously, the implementation by Cash *et al.* (2013) is almost identical with the exception of the following:

- The implementation supports Conjunctive and Boolean Queries – *The description of SSE previously assumed that all Queries consisted of a single Term (Conjunctive/Boolean Queries were not discussed as they were ruled beyond the scope of this dissertation)*.
- The implementation stores the entire Inverted Index; *that is, Lexicon, Posting List and Document Collection*, in secondary memory at all times (due to the fact the implementation is designed to scale to extremely large Data Sets).
- The implementation includes a RAM resident Data Structure known as an *X-Set* that works in combination with the Inverted Index Data Structure to aid with the execution of Conjunctive/Boolean queries.

In terms of programming languages, the implementation of SSE by Cash *et al.* (2013) was developed using the C programming language. Any and all cryptographic functionality associated with implementation employed the use of the OpenSSL library of cryptographic algorithms.

In relation to Data Structures, a single Data Structure known as a *T-Set* is used to store both the Lexicon and the Postings in the implementation presented by Cash *et al.* (2013). In essence, a T-Set is a modified Hash Table that can store a fixed number of values, instead of a single value (as is the case with a standard Hash Table); *that is, Key => Value 1, Value 2, …, Value N (T-Set) instead of Key => Value (Hash Table).* When stored on disk, the T-Set is subdivided into a number of smaller Hash Tables, with the size of each individual Hash Table based on the characteristics of the underlying Operating System and storage medium.

In terms of algorithms, the SSE implementation by Cash *et al.* (2013) utilises AES-FFX for Posting encryption, while AES-HMAC or AES-CMAC is used for keyed hashing of Lexicon Terms[17]. The exact algorithm used for Document encryption is not disclosed.

In relation to Test Data, Cash *et al.* (2013) tested their SSE implementation against three separate Test Data Sets: the entire Enron E-Mail Collection[18] (1.5 million Documents consisting of 1.2 million distinct Lexicon Terms), a 100,000 record Database generated from census data, and a number of subsets of the ClueWeb09

[17] The associated paper does not specify whether AES-HMAC or AES-CMAC was used during program execution. The paper simply states that the chosen algorithm depends on platform characteristics.

[18] The work of Kamara *et al.* (2012) – discussed previously – employed the use of a subset of the Enron E-mail Collection – not the entire Collection as is the case with Cash *et al.* (2013).

collection of crawled web pages (the largest of which was 410GB in size (13,284,801 HTML Files) with approximately 2.7 billion Postings).

In terms of Research Results, the work of Cash *et al.* (2013) focussed on querying the SSE Inverted Index using both single Term Queries and Conjunctive/Boolean Queries[19]. As was the case with Kamara et al. (2012), the Results presented by Cash *et al.* (2013) only take into account the process of retrieving and decrypting matching Postings from within the SSE Inverted Index itself– the Results do not include the amount of time taken to retrieve matching Documents from disk and returning said Documents to the Client. Identifying and decrypting those Postings associated with the most frequently occurring Lexicon Term for the Enron E-Mail Test Data Set (690,492 Postings) took approximately 70 seconds (approximately 100 microseconds (µs) per Postings). Unlike, Kamara *et al.* (2012), Cash *et al.* (2013) does not disclose the amount of time taken to generate an ESS.

As regards hardware, the experiments conducted by Cash *et al.* (2013) were performed on an IBM Blade HS22 running a Linux operating system, with all secondary memory provided by a Storage Attached Network (SAN) device.

[19] Given that this dissertation is only concerned with single Term SSE Queries, only the Results of single Term SSE Queries – and not Conjunctive/Boolean SSE Queries - are discussed here.

2.3.4.3 Critical Analysis of Existing Implementations

From the Research Results presented by Kamara *et al.* (2012) and Cash *et al.* (2013), it is apparent that the search time associated with SSE is impressive – to the point that one could argue SSE is efficient enough to be deployed in a Cloud environment. In addition, the work of Cash *et al.* (2013) proves that SSE does indeed scale to large Data Sets whilst maintaining its search efficiency, and also has the ability to support Boolean/Conjunctive Queries in an efficient manner whilst maintaining Data/Query Privacy.

Despite such impressive Results, the author believes both papers focussed on the performance of a single component of SSE; *that is, searching an SSE Inverted Index*, and not SSE as a whole. Specifically, the author feels that both papers have glossed over the topic of SSE Inverted Index Construction. Given that constructing an SSE Inverted Index is a necessary pre-requisite to searching an SSE Inverted Index; the author feels the topic deserves significantly more attention than that which it has been given in the published literature thus far. Kamara *et al.* (2012) cover the topic briefly in their work; however as indicated previously, the Results presented are somewhat skewed by the fact they only include the Results of converting a pre-existing IR Inverted Index into an SSE Inverted Index – the Results do not include the time taken to generate the initial IR Inverted Index. Cash *et al.* (2013) make no mention of the time taken to generate the SSE Inverted Index used in their work.

In addition to largely ignoring the process of constructing an SSE Inverted Index, both papers have also ignored the process of transferring the SSE Inverted Index and the encrypted Document Collection from the Client to the Server. As Kamara *et al.* (2012) correctly points out, the time taken to transfer both the SSE Inverted Index and the encrypted Document Collection from the Client to the Server will vary depending on the underlying system (Kamara et al. (2012) failed to cover this part of SSE for this reason); however the author personally feels that the same can also be argued in relation to cryptographic operations (which are of course reported on in detail in both implementations).

When discussing their Results in relation to searching an SSE Inverted Index, both Kamara *et al.* (2012) and Cash *et al.* (2013) readily acknowledge that their Results only cover searching the SSE Inverted Index and decrypting the Postings associated with the Lexicon Term being searched – their Results do not include the time associated with retrieving and forwarding matching Documents to the Client – another essential component of SSE.

In addition to their failure to examine SSE as a whole, the author is also somewhat disappointed in the quality of information relating to the Test Data Sets and findings of both papers.

In relation to Test Data, Table 1 summarises the Test Data statistics published (and not published) in both papers[20].

Information Disclosed	**Kamara *et al.* (2012)**	**Cash *et al.* (2013)**
Number of Documents In Data Set	No	Yes
Number of Terms In Data Set	No	No
Number of Unique Terms In Data Set	No	Yes (Enron Data Set Only)
Number of Postings In Data Set	Yes (Postings In Media File Data Set Not Disclosed)	Yes (Postings In Census Data Set Not Disclosed)
Number of Postings Associated With Highest Frequency Lexicon Term	No	Yes (Not Disclosed For Media File Data Set)
Size of Test Data Set	Yes	Yes (Size Of Census Data Set Not Disclosed)

Table 1: Test Data Statistics

The total number of Terms in the Data Set is relevant in that it dictates the amount of work needed to be performed during Document Tokenisation; *that is, IR Inverted Index Construction*, the number of unique Terms in the Data Set is relevant in that it dictates the number of Terms contained within the Inverted Index (both the IR Inverted Index and the SSE Inverted Index), while the number of Postings in the Data Set is relevant in that it dictates the number of Postings contained within the Inverted Index (both the IR Inverted Index and the SSE Inverted Index). The number of Postings associated with the highest frequency Lexicon Term is relevant in that

[20] Please note that table cells highlighted in Red denote information that has not been disclosed, table cells highlighted in Orange denote information that has only been partially disclosed, while table cells highlighted in Green denote information that has been fully disclosed.

the Term in question is typically used to measure the worst case scenario of searching an SSE Inverted Index, while the size of the Test Data Set is relevant in terms of transmitting the Document Collection to the Server from the Client. As can be seen from Table 1, a number of these statistics are not disclosed (or are only partially disclosed) by the respective authors; therefore making it difficult to give context to the associated experiment results.

In relation to Inverted Index Construction statistics, Table 2 summarises the Test Data statistics published (and not published) in both papers[20].

Information Disclosed	**Kamara *et al.* (2012)**	**Cash *et al.* (2013)**
Time Taken To Generate IR Inverted Index	No	No
Size Of IR Inverted Index	No	No
Time Taken To Convert IR Inverted Index To SSE Inverted Index	Yes	No
Size of SSE Inverted Index	No	Yes
Time Taken To Encrypt Document Collection	Yes	No

Table 2: Inverted Index Construction Statistics

The time taken to generate the IR Inverted Index is significant in that the processing time is linear in the number of Terms contained within the Document Collection. The time taken to generate the SSE Inverted Index is significant in that the processing time is linear in the number of Postings contained within the IR Inverted Index, while the size of the SSE Inverted Index is relevant in terms of transmitting the SSE Inverted Index to the Server from the Client.

As can be seen in Table 2, neither Kamara *et al.* (2012) or Cash *et al.* (2013) disclose any information in relation to IR Inverted Index Construction. When reporting the Results of converting their IR Inverted Index to an SSE Inverted Index, Kamara *et al.* (2012) choose to do so by charting their Results against the size of the Test Data Set (in MB)[21]. Personally the author feels this information would be much more informative if it were charted against the number of Postings in the Test Data Set, given that the size of the underlying Data Set in no way reflects the number of unique Terms or Postings in the Data Set. *For Example: a 10MB DOCX file may contain the same Term repeated over and over again; that is, one unique Term => one Posting*. In addition, the author feels that the use of the Document Collection size here is a poor choice given the fact that different file formats can contain the same number of words, but differ greatly in size (such a TXT Files and DOCX Files)[21].

In relation to Inverted Index Querying statistics, Table 3 summarises the Test Data statistics published (and not published) in both papers[20].

[21] It should be noted that the chart in question also includes encrypting the associated Document Collection (which is of course dependant on the size of the underlying Document Collection); however the time associated with executing this portion of the task represents only a fraction of the time associated with generating the SSE Inverted Index.

Information Disclosed	**Kamara *et al.* (2012)**	**Cash *et al.* (2013)**
Time Taken To Generate ESS	Yes	No
Time Taken To Search SSE Inverted Index For Highest Frequency Lexicon Term (Including Decryption Of Postings)	Yes	Yes

Table 3: Inverted Index Querying Statistics.

As can be seen from Table 3, both Kamara *et al.* (2012) and Cash *et al.* (2013) have disclosed the time taken to search the SSE Inverted Index and to identify and decrypt the Postings associated with the highest frequency Lexicon Term. Unfortunately Kamara *et al.* (2012) did not publish the number of Postings associated with the highest frequency Lexicon Term; instead the amount of time associated with the search was published. Without the number of Postings associated with the highest frequency Lexicon Term, it is difficult to place into context the significance of the Results published. In the case of Cash *et al.* (2013), both the search time and the number of Postings associated with the highest frequency Lexicon Term were published, therefore providing readers with the ability to estimate the amount of time required to decrypt a single Posting[22]. In relation to the time taken to generate an ESS, only Kamara *et al.* (2012) have published their Results for this area. While Cash *et al.* (2013) have not revealed

[22] In the case of Cash et al. (2013), approximately 700,000 Postings were identified and decrypted in approximately 70 seconds for the highest frequency Lexicon Term (approximately 100 microseconds per Posting).

their statistics for this part of SSE Search, the author feels it is safe to assume that the cost of producing an ESS is miniscule given that the implementation developed by Kamara *et al.* (2012) does so in 35 microseconds (µs).

In relation to Test Environment statistics, Table 4 summarises the statistics published (and not published) in both papers[20].

Information Disclosed	**Kamara *et al.* (2012)**	**Cash *et al.* (2013)**
Operating System	Yes	Yes
Processor	Yes	No
RAM	No	No
Hard Disk Size	No	No

Table 4: Test Environment Statistics.

Given the fact that both authors acknowledge the effect that the underlying system can have on the experiment Results produced, the author is somewhat disappointed in the relation to the lack of information disclosed in both papers regarding the underlying Test Environments (see Table 4) [20].

In an effort to determine the performance cost of preserving Data/Query privacy using SSE, Cash *et al.* (2013) opted to perform a performance comparison between their implementation of SSE and a MySQL Server comprising a plaintext database. Personally, the author believes a comparison between an equivalent plaintext Information Retrieval (IR) system would be a much more appropriate comparison to make when determining the performance cost of SSE (given the fact that plaintext searching is the universally accepted method of IR); nonetheless, the author believes that the decision to perform a comparison against MySQL can be explained by the fact that their implementation of SSE is optimised for searching large Data

Sets stored in secondary memory and not primary memory (as is the case with all Database Servers – including MySQL).

2.4 Research Question(s)

From Section 2.3.4.3 (Critical Analysis of Existing Implementations), it is apparent that the author has identified a number of issues with the information available regarding existing implementations of SSE. In addition, the author has also identified that exiting Research in the area of SSE has almost exclusively focussed on the topic of searching in SSE, while largely ignoring the topic of SSE Inverted Index Construction.

As part of this dissertation, it is the author's intention to contribute towards the areas of weakness identified previously. With this in mind, the author has identified the following Research Question for their dissertation:

- RQ1: *How Efficient Is Searchable Symmetric Encryption (SSE) When Implemented And Deployed In A Cloud Environment?*
- RQ2: *What Is The Performance Cost Of Preserving Data/Query Privacy Using Searchable Symmetric Encryption (SSE) When Compared To Plaintext Information Retrieval (IR)?*

As indicated previously in Section 2.3.4.3, the existing SSE literature has failed to cover the whole spectrum of activities associated with SSE (see Table 5); hence RQ1. Additionally, the existing published literature has yet to examine the usage of SSE when deployed in a Cloud computing environment.

Activity	Covered In Existing Published Literature
SSE Inverted Index Construction	
IR Inverted Index Generation By Client	No
SSE Inverted Index Generation By Client	Yes
Document Collection Encryption By Client	Yes
Uploading Of SSE Inverted Index To Server	No
Uploading Of Encrypted Document Collection To Server	No
SSE Inverted Index Searching	
ESS Generation By Client	Yes
Identifying And Decrypting Matching Postings	Yes
Returning Matching Documents To Client	No

Table 5: SSE Activities Covered By Existing Literature.

In relation to RQ2, the existing published literature has only compared the performance of SSE with a Database Server, and not a traditional plaintext IR system that utilises an Inverted Index (Cash *et al.,* 2013).

Sub-Questions and Hypothesis emanating from RQ1 can be found in Section 2.4.1, while Sub-Questions and Hypothesis emanating from RQ2 can be found in Section 2.4.2.

2.4.1 RQ1: Sub-Questions and Hypothesis

Regarding RQ1, SSE can be divided into two distinct operations: SSE Inverted Index Construction (RQ1.1) and SSE Inverted Index Querying (RQ1.2).

Section 2.4.1.1 presents Sub-Questions and Hypothesis relating to SSE Inverted Index Construction, while Section 2.4.1.2 presents Sub-Questions and Hypothesis relating to SSE Inverted Index Querying.

2.4.1.1 SSE Inverted Index Construction

Table 6 presents Sub-Questions and Hypothesis emanating from RQ1 relating to SSE Inverted Index Construction.

RQ#	Sub-Question	Hypothesis
1.1.1	How Long Does It Take To Construct An IR Inverted Index?	*The time taken to construct an IR Inverted Index will be proportional to the number of Terms contained within the underlying Document Collection.*
1.1.2	How Long Does It Take To Convert An IR Inverted Index Into An SSE Inverted Index?	*The time taken to convert a pre-existing IR Inverted Index into an SSE Inverted Index will be proportional to the number of Lexicon Terms and Postings contained within the IR Inverted Index.*
1.1.3	How Long Does It Take To Encrypt a Document Collection?	*The time taken to encrypt a Document Collection will be approximately equivalent to the number of Terms in the underlying Document Collection.*

1.1.4	How Long Does It Take To Upload An SSE Inverted Index To The Server?	*The time taken to upload an SSE Inverted Index to the Server will be proportional to the size of the SSE Inverted Index.*
1.1.5	How Long Does It Take To Upload an Encrypted Document Collection to the Server?	*The time taken to upload an Encrypted Document Collection to the Server will be proportional to the size of the Encrypted Document Collection.*

Table 6: Sub-Questions And Hypothesis For RQ1 Relating To SSE Inverted Index Construction.

2.4.1.2 SSE Inverted Index Querying

Table 7 presents Sub-Questions and Hypothesis emanating from RQ1 relating to SSE Inverted Index Querying.

RQ#	**Sub-Question**	**Hypothesis**
1.2.1	How Long Does It Take To Generate An Encrypted Search String (ESS)?	*The time taken to generate an ESS will be proportional to the size of the plaintext Search String.*
1.2.2	How Long Does It Take To Search An SSE Inverted Index?	*The time take to Search an SSE Inverted Index will be proportional to the number of Postings associated with the Lexicon Term searched for.*
1.2.3	How Long Does It Take To Return All Matching Documents To The Client?	*The time taken to return all matching Documents to the Client will be proportional to the size of the matching Document Collection*

Table 7: Sub-Questions And Hypothesis For RQ1 Relating To SSE Inverted Index Querying.

2.4.2 RQ2: Sub-Questions and Hypothesis

Regarding RQ2, SSE and plaintext IR can be divided into two distinct operations: Document Collection Uploading (RQ2.1) and Inverted Index Querying (RQ2.2).

Section 2.4.2.1 presents Sub-Questions and Hypothesis relating to Document Collection Uploading, while Section 2.4.2.2 presents Sub-Questions and Hypothesis relating to Inverted Index Querying.

2.4.2.1 Plaintext IR Uploading Vs. SSE Uploading

Table 8 presents Sub-Questions and Hypothesis emanating from RQ2 relating to Document Collection Uploading.

RQ#	Sub-Question	Hypothesis
2.1.1	How Long Does It Take To Upload A Document Collection Using SSE, When Compared With Traditional Plaintext Information Retrieval (IR)?	*Uploading a Document Collection using SSE will take longer when compared to plaintext IR due to the requirement of the Client to generate an SSE Inverted Index (and upload it to the Server), as well as the requirement that the Client will encrypt the associated Document Collection prior to uploading it to the Server.*

Table 8: Sub-Questions and Hypothesis for RQ2 Relating To Document Collection Uploading

2.4.2.2 Plaintext IR Querying Vs. SSE Querying

Table 9 presents Sub-Questions and Hypothesis emanating from RQ2 relating to Inverted Index Querying.

RQ#	Sub-Question	Hypothesis
2.2.1	How Long Does It Take To Query An SSE Inverted Index, When Compared To Traditional Plaintext Information Retrieval (IR) Inverted Index Querying?	*Querying an Inverted Index using SSE will take longer when compared to plaintext IR due to the requirement of the Server to decrypt SSE Postings.*

Table 9: Sub-Questions and Hypothesis for RQ2 Relating To Inverted Index Querying

3. Software Requirements Specification

3.1 Introduction

3.1.1 Purpose

This Chapter provides a detailed description of the Software Requirements for the initial prototypes of the “PlainTXT Storage and Search Engine” and “CipherTXT Storage and Search Engine“ applications developed as part of this dissertation.

3.1.2 Project Scope

Both software artefacts produced as part of this dissertation have been developed with a view to providing answers to the Research Questions identified previously in Chapter 2. Both artefacts are examples of personal file hosting applications. Like all file hosting applications, the objective of both the “PlainTXT Storage and Search Engine” and “CipherTXT Storage and Search Engine“ is to allow service users to store their files in the Cloud, and to access/retrieve those files as and when needed (via a web browser).

In the case of the “PlainTXT Storage and Search Engine” application, users will be able to store their personal files in plaintext form, as well as having the ability to search and retrieve those files by forwarding queries to the application in plaintext form.

In the case of the “CipherTXT Storage and Search Engine” application, users will be provided with the exact same functionality as the “PlainTXT Storage and Search Engine” application, with the exception that both user’s files and queries are encrypted prior to being forwarded to the application for storage/usage.

Given the prototype status of both applications, a number of standard features and functionality typically associated with personal file hosting services have been classified as out of scope for the initial version of both software artefacts. Features and functionality considered out of scope for both applications can be seen in Table 10:

No Support for Multiple Users	*The initial prototype(s) will be designed and implemented with a single user in mind. Support for this feature may be added in future editions.*
No Support for User Authentication	*The initial prototype(s) will be designed and implemented without user authentication. Simply accessing the application(s) at their specified URLs will provide the user with access to the full functionality of the application(s) (no username/password will be necessary). Support for this feature may be added in future editions.*

Limited File Format Support	*The initial prototype(s) will only allow for TXT files to be uploaded to the application(s). Support for additional file formats may be added in future editions.*
No Support for Viewing List of Documents Uploaded Previously	*The user will be unable to view the list of files they have uploaded to the application(s) previously. Support for this feature may be added in future editions.*
No Support for Viewing File Contents Online	*The user will be unable to view the contents of files online; that is, within the browser, irrespective of its file format. Instead, users will have to download files to a client machine in order to view them. Support for this feature may be added in future editions.*
No Support for Document Collection Updates	*Once a Document Collection has been uploaded to the Server, the user will be unable to update that Document Collection; that is, add an additional Document to the Collection, edit an existing Document within the Collection or delete an existing Document within the Collection. Essentially, the Document Collection will be non-modifiable unless the Document Collection is overwritten with a newly uploaded Document Collection. Support*

	for this feature may be added in future editions.
Limited Query Support	*The initial prototype(s) will only allow for single Term Queries (as was the case with the Literature Review). Support for multi-Term Queries and other advanced forms of Querying may be added in future editions.*
No Support For Document ID Generation	*The initial prototype(s) will not support the generation of unique Document IDs for TXT Documents uploaded to the Server. Instead, the initial prototype(s) assumes the user has chosen a unique numeric name for each TXT Document, prior to being uploaded to the Server. Support for this feature may be added in future editions.*

Table 10: Features Considered Out Of Scope For Software Artefacts.

3.1.3 Overview

Section 3.2.1 presents the Functional Requirements for the “PlainTXT Storage and Search Engine” application, while Section 3.2.2 presents the Functional Requirements for the “CipherTXT Storage and Search Engine” application.

Section 3.3 presents a set of Non-Functional Requirements that apply to both applications.

3.2 Functional Requirements

3.2.1 PlainTXT Storage and Search Engine

The following are the Functional Requirements for the “PlainTXT Storage and Search Engine” application:

- PFR-001: Upload TXT Document(s) To Server.
- PFR-002: Retrieve TXT Documents Containing Specified Search String.

3.2.2 CipherTXT Storage and Search Engine

The following are the Functional Requirements for the “CipherTXT Storage and Search Engine” application:

- CFR-001: Encrypt and Upload TXT Document(s) To Server.
- CFR-002: Retrieve TXT Documents Containing Encrypted Search String.
- CFR-003: Decrypt Encrypted TXT Documents Retrieved From Server.

3.3 Non-Functional Requirements

The following are the Non-Functional Requirements for both the “PlainTXT Storage and Search Engine” application and the “CipherTXT Storage and Search Engine” application:

- NFR-001: The software shall be Efficient.
- NFR-002: The software shall be Robust.
- NFR-003: The software shall be Maintainable.
- NFR-004: The software shall be Reliable.
- NFR-005: The software shall be Usable.
- NFR-006: The software shall be Secure.

4. Software Design

4.1 Introduction

This Chapter provides the Design details for the “PlainTXT Storage and Search Engine” and “CipherTXT Storage and Search Engine” applications developed as part of this dissertation. Section 4.2 incorporates the High-Level Design details of both applications; *that is, Use Case Descriptions, Detailed Activity Diagrams and User Interface Design*, while Section 4.3 incorporates the Low-Level Design of both applications; *that is, Sequence Diagrams*.

4.2 High Level Design

Section 4.2.1 denotes the Use Case Descriptions associated with both the “PlainTXT Storage and Search Engine” application as well as the “CipherTXT Storage and Search Engine” application. Section 4.2.2 denotes the Detailed Activity Diagrams associated with same, while Section 4.2.3 denotes the User Interface Design diagrams associated with same.

4.2.1 Use Case Descriptions

Section 4.2.1.1 denotes the Use Case Descriptions associated with the ‘PlainTX Storage and Search Engine’ application, while Section 4.2.1.2 denotes the Use Case Descriptions associated with the ‘CipherTXT Storage and Search Engine’ application.

4.2.1.1 PlainTXT Storage and Search Engine

Section 4.2.1.1.1 outlines the Use Case Description for PFR-001; Section 4.2.1.1.2 outlines the Use Case Description for PFR-002, while Section 4.2.1.1.3 provides a graphical summary (in the form of an Activity Diagram) of both Use Case Descriptions.

Figure 14 outlines the Use Case Diagram associated with the “PlainTXT Storage and Search Engine” application.

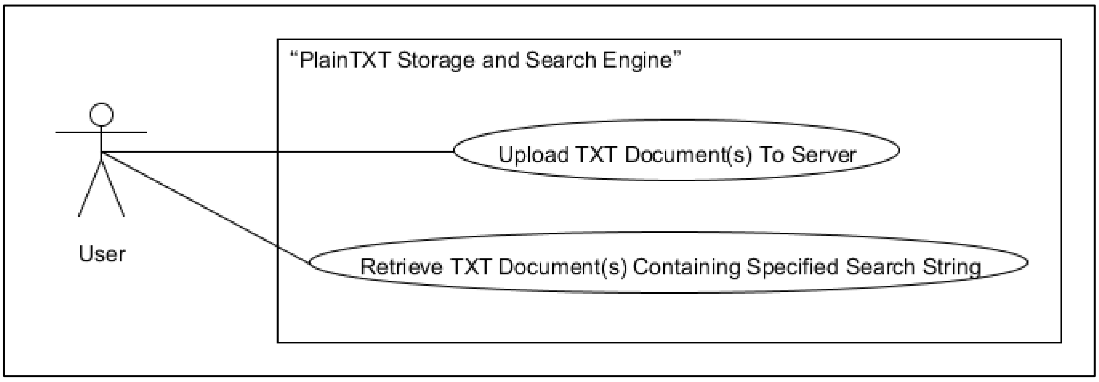

“PlainTXT Storage and Search Engine”
User
Upload TXT Document(s) To Server
Retrieve TXT Document(s) Containing Specified Search String

Figure 14: “PlainTXT Storage and Search Engine” Use Case Diagram.

4.2.1.1.1 PFR-001: Upload TXT Document(s) To Server Use Case

Use Case	Upload TXT Document(s) To Server.
Objective	To Upload One Or More TXT Files To The Server.
Pre-Condition	1. Server Is Running. 2. User Has Connected To Application.
Main Flow	1. User Selects ‘Upload File(s)’ Button. 2. Application Displays A Dialog Box Consisting Of Two Buttons – One Titled ‘Choose Files’ (*To Select Which Files To Upload*) – And Another Titled ‘Upload’ (*Which Forwards The Chosen Files To The Server For Storage*). 3. User Selects ‘Choose Files’ Button. 4. Application Displays A File Chooser Dialog Box (*Options: Open, Cancel*). 5. User Chooses TXT File(s) To Be Uploaded To Server. 6. User Chooses ‘Open’. 7. File Chooser Dialog Box Closes. 8. User Selects ‘Upload’ Button. 9. Application Displays ‘Upload Successful’.
Alternative Flow	1A. User Fails To Select ‘Upload File(s)’ Button. 2A. Dialog Box Fails To Display. 3A. User Fails To Select ‘Choose File(s)’ Button. 4A. File Chooser Dialog Box Fails To Display.

	5A. User Fails To Select Any Files To Upload. 6A. User Chooses ‘Cancel’ Button. 7A. File Chooser Dialog Box Fails To Close. 8A. User Fails To Select ‘Upload’ Button. 9A. Application Displays ‘Upload Failed’.
Post Condition	Users Files Have Been Uploaded To Server.

Table 11: PFR-001: Upload TXT Document(s) To Server Use Case Description.

4.2.1.1.2 PFR-002: Retrieve TXT Documents Containing Specified Search String Use Case

Use Case	Retrieve TXT Document(s) Containing Specified Search String.
Objective	To Retrieve Those Files From The Server That Contain A Specified Search String.
Pre-Condition	1. Server Is Running. 2. User Has Connected To Application. 3. TXT Files Have Been Uploaded To Server Previously.
Main Flow	1. User Enters Search String Into 'Search' Text Field. 2. User Selects 'Search' Button. 3. Application Transmits ZIP File To User Containing All TXT Files Containing Specified Search String.
Alternative Flow	1A. User Fails To Enter Search String Into 'Search' Text Field. 2A. User Fails To Select 'Search' Button. 3A. No TXT Files Contain Specified Search String.
Post Condition	User Receives ZIP File From Server Containing All TXT Files Matching Specified Search String.

Table 12: PFR-002: Retrieve TXT Documents Containing Specified Search String Use Case Description.

4.2.1.1.3 Use Case Summary (Activity Diagram)

The following Activity Diagram summarises the steps involved in each Use Case for the “PlainTXT Storage and Search Engine” application.

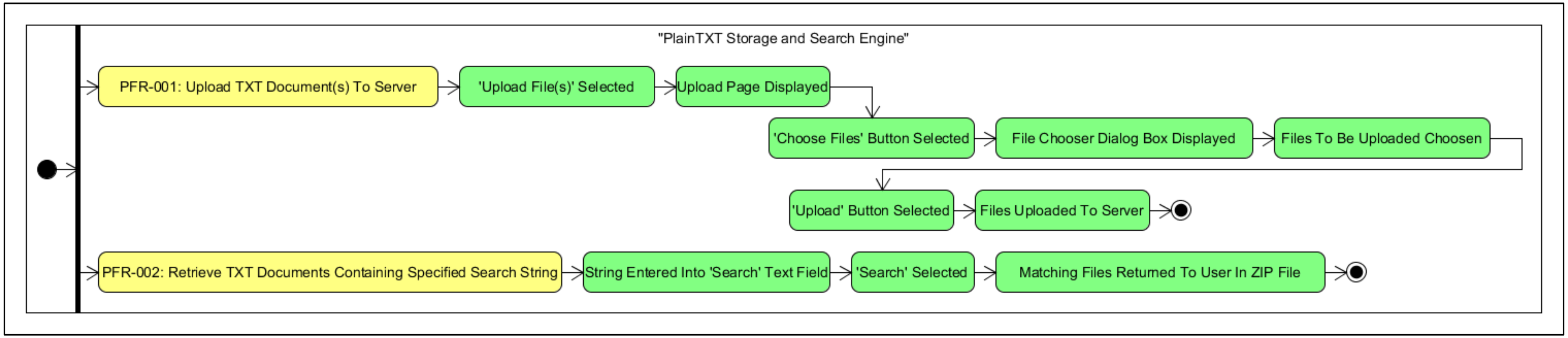

"PlainTXT Storage and Search Engine"
PFR-001: Upload TXT Document(s) To Server
'Upload File(s)' Selected
Upload Page Displayed
'Choose Files' Button Selected
File Chooser Dialog Box Displayed
Files To Be Uploaded Choosen
'Upload' Button Selected
Files Uploaded To Server
PFR-002: Retrieve TXT Documents Containing Specified Search String
String Entered Into 'Search' Text Field
'Search' Selected
Matching Files Returned To User In ZIP File

Figure 15: “PlainTXT Storage and Search Engine” Activity Diagram (Use Case Description Summary).

4.2.1.2 CipherTXT Storage and Search Engine

Section 4.2.1.2.1 outlines the Use Case Description for CFR-001; Section 4.2.1.2.2 outlines the Use Case Description for CFR-002; Section 4.2.1.2.3 outlines the Use Case Description for CFR-003, while Section 4.2.1.2.4 provides a graphical summary (in the form of an Activity Diagram) of all three Use Case Descriptions.

Figure 16 outlines the Use Case Diagram associated with the “CipherTXT Storage and Search Engine” application.

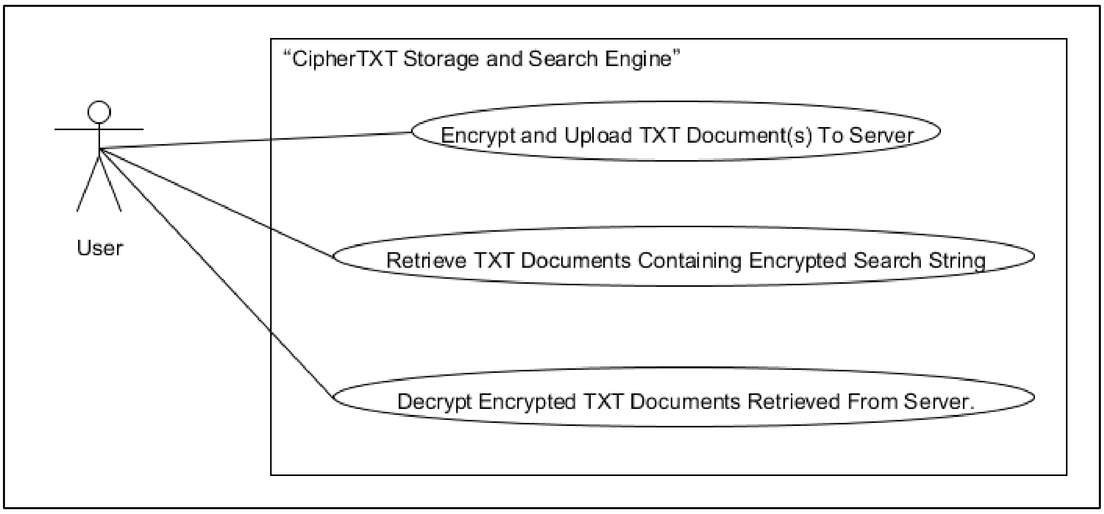

“CipherTXT Storage and Search Engine”
User
Encrypt and Upload TXT Document(s) To Server
Retrieve TXT Documents Containing Encrypted Search String
Decrypt Encrypted TXT Documents Retrieved From Server.

Figure 16: “CipherTXT Storage and Search Engine” Use Case Diagram.

4.2.1.2.1 CFR-001: Encrypt and Upload TXT Document(s) To Server Use Case

Use Case	Encrypt and Upload TXT Document(s) To Server.
Objective	To Encrypt One Or More TXT Files And Upload Them To The Server (In Encrypted Form).
Pre-Condition	1. Server Is Running. 2. User Has Connected To Application.
Main Flow	1. User Selects 'Upload File(s)' Button. 2. Application Displays A Dialog Box Consisting Of Three Text Fields And Two Buttons. Text Fields: 'Lexicon Password' (*Password To Encrypt Inverted Index Lexicon With*). 'Postings Password' (*Master Key Used To Generate Posting Passwords*). Document Password (*Password To Encrypt Documents With*). Buttons: 'Choose Files' (*To Select Which Files To Upload*) 'Upload' (*Forwards Chosen Files To The Server For Storage*). 3. User Enters Lexicon Password Into 'Lexicon Password' Text Field. 4. User Enters Master Postings Password Into 'Postings Password' Text Field. 5. User Enters Document Password Into 'Document Password' Text Field. 6. User Selects 'Choose Files' Button. 7. Application Displays A File Chooser Dialog Box (*Options:*

	Open, Cancel). 8. User Chooses TXT File(s) To Be Uploaded To Server. 9. User Chooses 'Open'. 10. File Chooser Dialog Box Closes. 11. User Selects 'Upload' Button. 12. Application Displays 'Upload Successful'.
Alternative Flow	1A. User Fails To Select 'Upload File(s)' Button. 2A. Dialog Box Fails To Display. 3A. User Fails To Enter Lexicon Password Into 'Lexicon Password' Text Field. 4A. User Fails To Enter Master Postings Password Into 'Postings Password' Text Field. 5A. User Fails To Enter Document Password Into 'Document Password' Text Field. 6A. User Fails To Select 'Choose File(s)' Button. 7A. File Chooser Dialog Box Fails To Display. 8A. User Fails To Select Any Files To Upload. 9A. User Selects 'Cancel' Button. 10A. File Chooser Dialog Box Fails To Close. 11A. User Fails To Select 'Upload' Button. 12A. Application Displays 'Upload Failed'.
Post Condition	Users TXT Files Have Been Encrypted And Uploaded To Server.

Table 13: CFR-001: Encrypt and Upload TXT Document(s) To Server Use Case Description.

4.2.1.2.2 CFR-002: Retrieve TXT Documents Containing Encrypted Search String Use Case

Use Case	Retrieve TXT Document(s) Containing Encrypted Search String.
Objective	To Retrieve Those Files From The Server That Contain A Specified Encrypted Search String.
Pre-Condition	1. Server Is Running. 2. User Has Connected To Application. 3. TXT Files Have Been Uploaded To Server Previously.
Main Flow	1. User Enters Search String Into 'Search' Text Field. 2. User Enters Lexicon Password Into 'Lexicon Password' Text Field. 3. User Enters Postings Password Into 'Postings Password' Text Field. 4. User Selects 'Search' Button. 5. Application Transmits ZIP File To User Containing All TXT Files Containing Encrypted Search String.
Alternative Flow	1A. User Fails To Enter Search String Into 'Search' Text Field. 2A. User Fails To Enter Lexicon Password Into 'Lexicon Password' Text Field. 2B. User Enters Incorrect Lexicon Password Into 'Lexicon Password' Text Field. 3A. User Fails To Enter Postings Password Into 'Postings

	Password' Text Field. 3B. User Enters Incorrect Postings Password Into 'Postings Password' Text Field. 4A. User Fails To Select 'Search' Button. 5A. Valid Lexicon Password Used – Valid Postings Password Used - No TXT Files Contain Encrypted Search String. 5B. Invalid Lexicon Password Used - Valid Postings Password Used - No TXT Files Contain Encrypted Search String. 5C. Valid Lexicon Password Used – Invalid Postings Password Used -No TXT Files Contain Encrypted Search String. 5D. Invalid Lexicon Password Used - Invalid Postings Password Used - No TXT Files Contain Encrypted Search String.
Post Condition	User Receives ZIP File From Server Containing All TXT Files Matching Encrypted Search String.

Table 14: CFR-002: Retrieve TXT Documents Containing Encrypted Search String Use Case Description.

4.2.1.2.3 CFR-003: Decrypt Encrypted TXT Documents Retrieved From Server Use Case.

Use Case	Decrypt Encrypted TXT Documents Retrieved From Server.
Objective	To Decrypt Encrypted TXT Documents Received From Server (Contained Within ZIP File).
Pre-Condition	1. Server Is Running. 2. User Has Connected To Application. 3. The User Possess A ZIP File Comprising Encrypted TXT Documents.
Main Flow	1. User Selects 'Decrypt' Button. 2. Application Displays A Dialog Box Consisting Of One Text Field And Two Buttons. <u>Text Fields:</u> Document Password (*Password To Decrypt TXT Documents With*). <u>Buttons:</u> 'Choose ZIP File' (*To Select ZIP File Containing TXT Documents*) 'Unzip/Decrypt Files' (*To Decrypt TXT Files Contained Within Designated ZIP File Using Specified Password*). 3. User Enters Document Password Into 'Document Password'

	Text Field. 4. User Selects ‘Choose ZIP File’ Button. 5. Application Displays A File Chooser Dialog Box (*Options: Open, Cancel*). 6. User Chooses ZIP File Containing Encrypted TXT Documents. 7. User Chooses ‘Open’. 8. File Chooser Dialog Box Closes. 9. User Selects ‘Unzip/Decrypt Files’ Button. 10. Application Generates A Folder (In The Same Location As The ZIP File Selected) Containing All TXT Files In Plaintext Form.
Alternative Flow	1A. User Fails To Select ‘Decrypt Files’ Button. 2A. Dialog Box Fails To Display. 3A. User Fails To Enter Document Password Into ‘Document Password’ Text Field. 3B. User Enters Incorrect Document Password Into ‘Document Password’ Text Field. 4A. User Fails To Select ‘Choose ZIP File’ Button. 5A. File Chooser Dialog Box Fails To Display. 6A. User Fails To Choose ZIP File. 6B. User Chooses Incorrect ZIP File. *For Example: a ZIP File Containing No Encrypted TXT Documents*. 7A. User Selects ‘Cancel’ Button.

	8A. File Chooser Dialog Box Fails To Close. 9A. User Fails To Select ‘Unzip/Decrypt Files’ Button. 10A. Application Unable To Decrypt TXT Files – Incorrect Password Entered.
Post Condition	User Possesses Folder Containing TXT Files In Plaintext Form (Having Previously Been In Ciphertext Form).

Table 15: CFR-003: Decrypt Encrypted TXT Documents Retrieved From Server Use Case Description.

4.2.1.2.4 Use Case Summary (Activity Diagram)

The Activity Diagram overleaf summarises the steps involved in each Use Case for the “CipherTXT Storage and Search Engine” application.

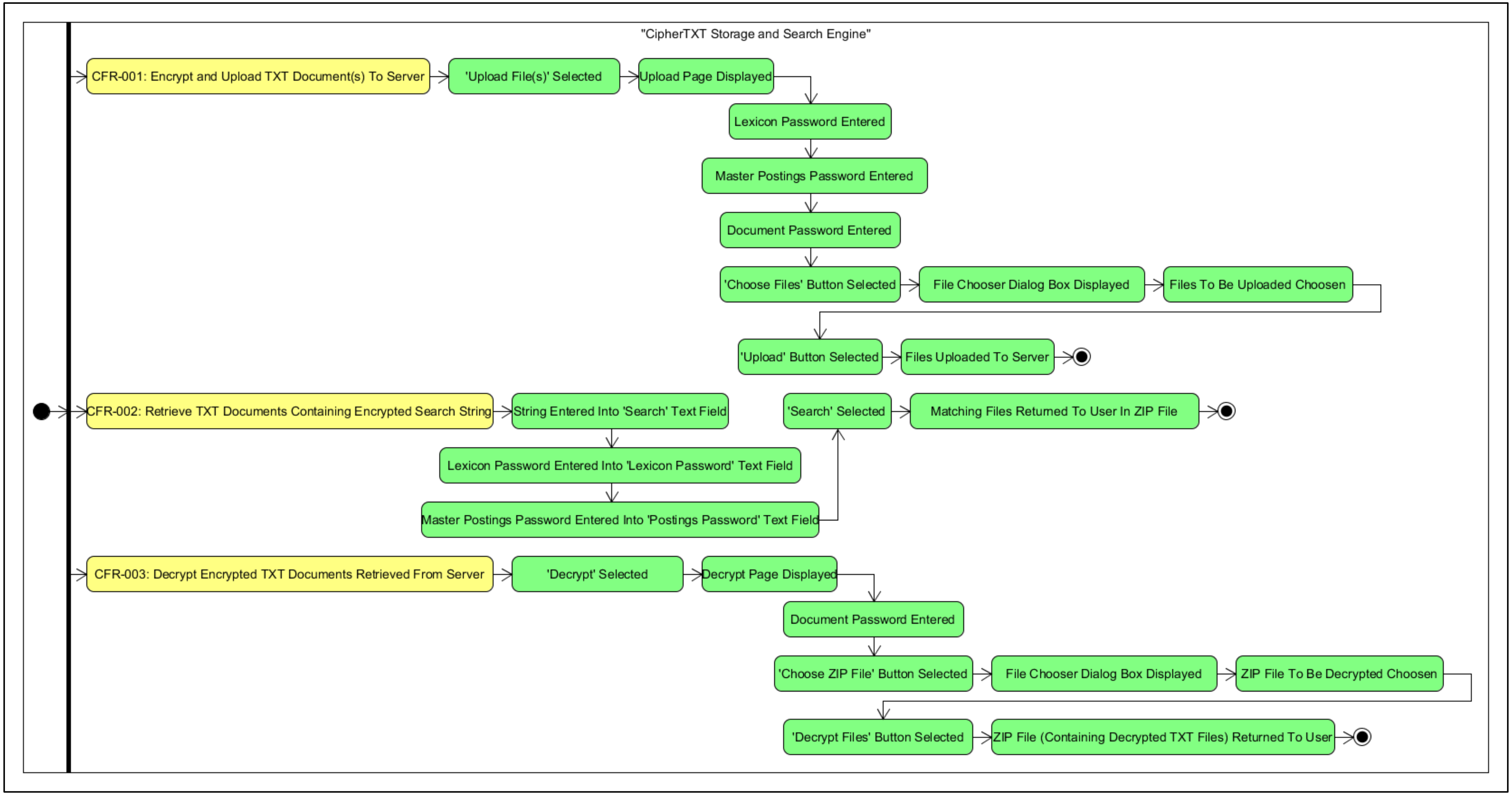

"CipherTXT Storage and Search Engine"
CFR-001: Encrypt and Upload TXT Document(s) To Server
'Upload File(s)' Selected
Upload Page Displayed
Lexicon Password Entered
Master Postings Password Entered
Document Password Entered
'Choose Files' Button Selected
File Chooser Dialog Box Displayed
Files To Be Uploaded Choosen
'Upload' Button Selected
Files Uploaded To Server
CFR-002: Retrieve TXT Documents Containing Encrypted Search String
String Entered Into 'Search' Text Field
Lexicon Password Entered Into 'Lexicon Password' Text Field
Master Postings Password Entered Into 'Postings Password' Text Field
'Search' Selected
Matching Files Returned To User In ZIP File
CFR-003: Decrypt Encrypted TXT Documents Retrieved From Server
'Decrypt' Selected
Decrypt Page Displayed
Document Password Entered
'Choose ZIP File' Button Selected
File Chooser Dialog Box Displayed
ZIP File To Be Decrypted Choosen
'Decrypt Files' Button Selected
ZIP File (Containing Decrypted TXT Files) Returned To User

Figure 17: "CipherTXT Storage and Search Engine" Activity Diagram (Use Case Description Summary).

4.2.2 Detailed Activity Diagram

The Activity Diagrams encountered previously in Section 4.2.1 simply listed the steps involved in each Use Case from the end users perspective. Neither diagram outlined the processing steps to be carried out as part of each Use Case, nor did they designate what processing is performed by the Client and what processing is performed by the Server.

Section 4.2.2.1 denotes the Detailed Activity Diagram for the “PlainTXT Storage and Search Engine” application, while Section 4.2.2.2 denotes the Detailed Activity Diagram for the “CipherTXT Storage and Search Engine” application. In both Diagrams, user actions are highlighted in Green; processing carried out by the Client is highlighted in Blue; while processing carried out by the Server is highlighted in Red.

4.2.2.1 PlainTXT Storage and Search Engine

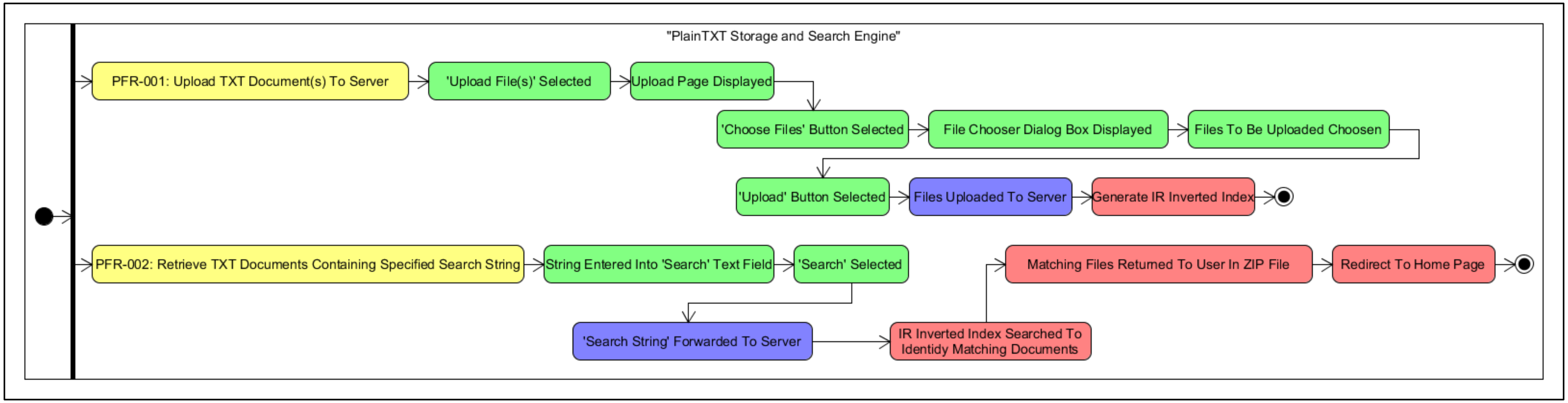

"PlainTXT Storage and Search Engine"
PFR-001: Upload TXT Document(s) To Server
'Upload File(s)' Selected
Upload Page Displayed
'Choose Files' Button Selected
File Chooser Dialog Box Displayed
Files To Be Uploaded Choosen
'Upload' Button Selected
Files Uploaded To Server
Generate IR Inverted Index
PFR-002: Retrieve TXT Documents Containing Specified Search String
String Entered Into 'Search' Text Field
'Search' Selected
Matching Files Returned To User In ZIP File
Redirect To Home Page
'Search String' Forwarded To Server
IR Inverted Index Searched To Identidy Matching Documents

Figure 18: PlainTXT Storage and Search Engine - Detailed Activity Diagram.

4.2.2.2 CipherTXT Storage and Search Engine

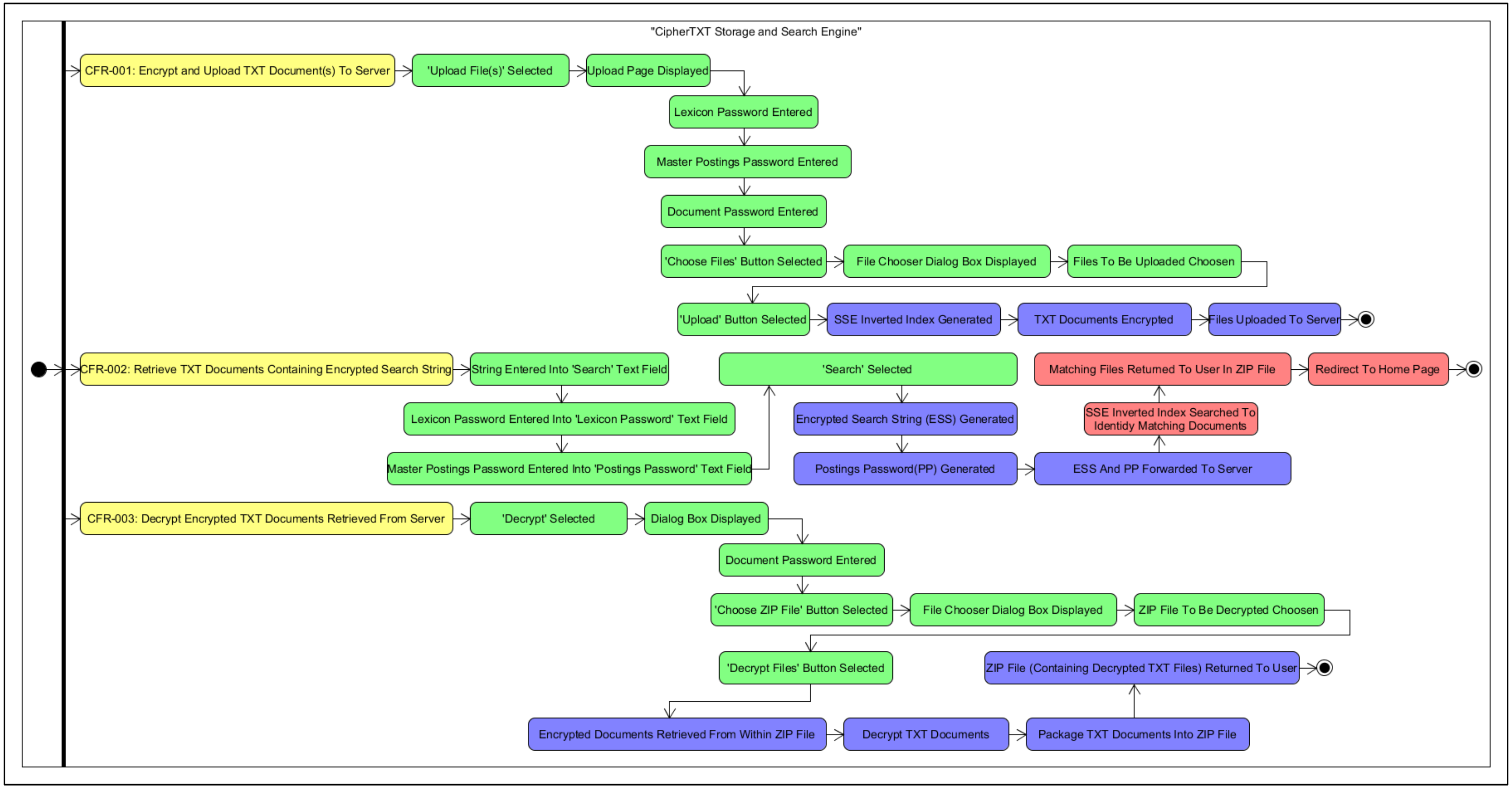

"CipherTXT Storage and Search Engine"
CFR-001: Encrypt and Upload TXT Document(s) To Server
'Upload File(s)' Selected
Upload Page Displayed
Lexicon Password Entered
Master Postings Password Entered
Document Password Entered
'Choose Files' Button Selected
File Chooser Dialog Box Displayed
Files To Be Uploaded Choosen
'Upload' Button Selected
SSE Inverted Index Generated
TXT Documents Encrypted
Files Uploaded To Server
CFR-002: Retrieve TXT Documents Containing Encrypted Search String
String Entered Into 'Search' Text Field
Lexicon Password Entered Into 'Lexicon Password' Text Field
Master Postings Password Entered Into 'Postings Password' Text Field
'Search' Selected
Encrypted Search String (ESS) Generated
Postings Password(PP) Generated
ESS And PP Forwarded To Server
SSE Inverted Index Searched To Identidy Matching Documents
Matching Files Returned To User In ZIP File
Redirect To Home Page
CFR-003: Decrypt Encrypted TXT Documents Retrieved From Server
'Decrypt' Selected
Dialog Box Displayed
Document Password Entered
'Choose ZIP File' Button Selected
File Chooser Dialog Box Displayed
ZIP File To Be Decrypted Choosen
'Decrypt Files' Button Selected
Encrypted Documents Retrieved From Within ZIP File
Decrypt TXT Documents
Package TXT Documents Into ZIP File
ZIP File (Containing Decrypted TXT Files) Returned To User

Figure 19: CipherTXT Storage and Search Engine - Detailed Activity Diagram.

4.2.3 User Interface Design

Section 4.2.3.1 denotes the User Interface Design for the “PlainTXT Storage and Search Engine” application, while Section 4.2.3.2 denotes the User Interface Design for the “CipherTXT Storage and Search Engine” application.

4.2.3.1 PlainTXT Storage and Search Engine

Section 4.2.3.1.1 denotes the User Interface Design of the Home Page for the “PlainTXT Storage and Search Engine” application, while Section 4.2.3.1.2 denotes the User Interface Design of the Upload page for same.

4.2.3.1.1 Home Page (Includes Search Bar)

Figure 20 denotes the User Interface Design of the Home Page for the "PlainTXT Storage and Search Engine" application.

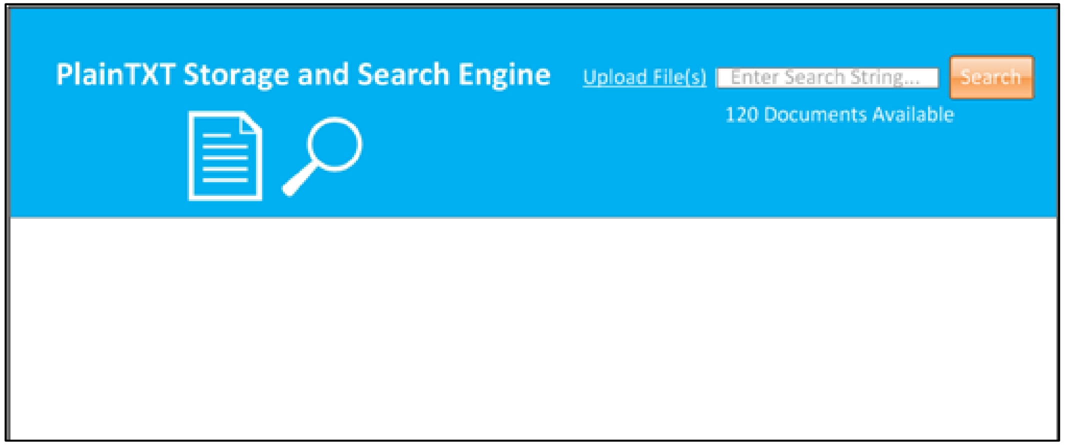
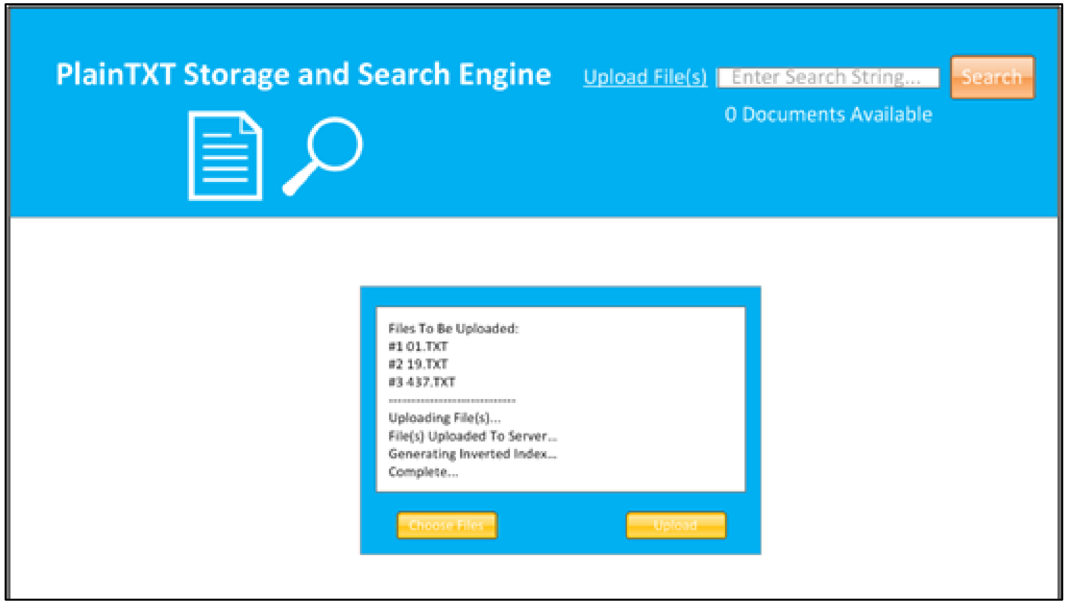


Figure 20: PlainTXT Storage and Search Engine - Home Page Design.

4.2.3.1.2 Upload Page

Figure 21 denotes the User Interface Design of the Upload Page for the “PlainTXT Storage and Search Engine” application.



```
Files To Be Uploaded:
#1 01.TXT
#2 19.TXT
#3 437.TXT
------------------------------
Uploading File(s)...
File(s) Uploaded To Server...
Generating Inverted Index...
Complete...
```

Figure 21: PlainTXT Storage and Search Engine - Upload Page Design.

4.2.3.2 CipherTXT Storage and Search Engine

Section 4.2.3.2.1 denotes the User Interface Design of the Home Page for the "CipherTXT Storage and Search Engine" application, Section 4.2.3.2.2 denotes the User Interface Design of the Upload page for same, while Section 4.2.3.2.3 denotes the User Interface Design of the Decrypt page for same also.

4.2.3.2.1 Home Page (Includes Search Bar)

Figure 22 denotes the User Interface Design of the Home Page for the “CipherTXT Storage and Search Engine” application.

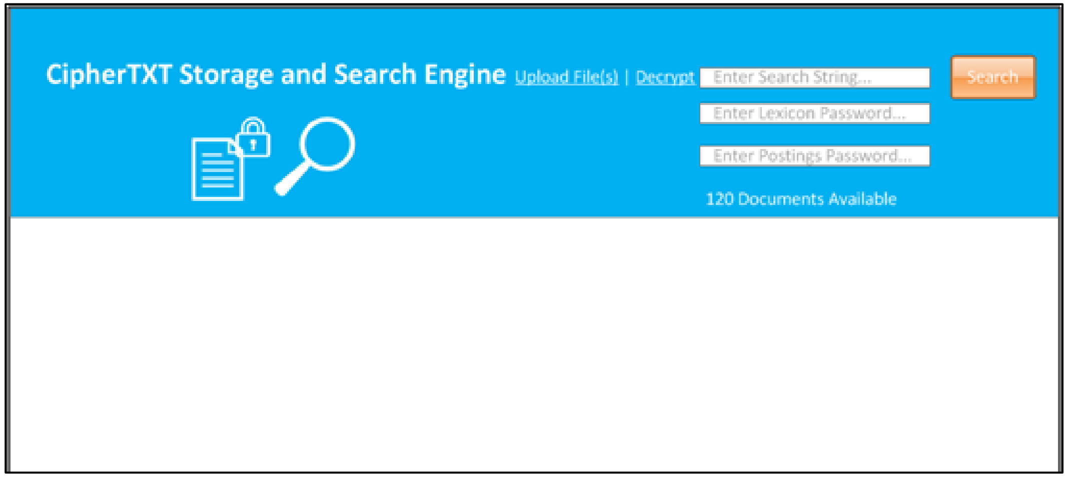
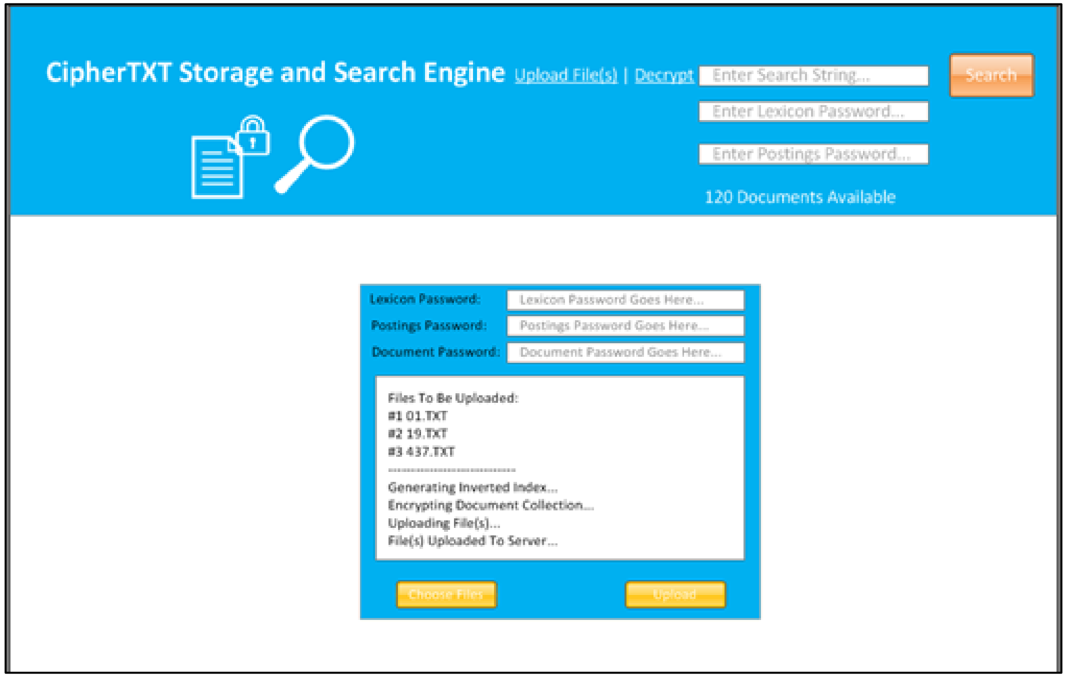

CipherTXT Storage and Search Engine
Upload File(s) | Decrypt
Enter Search String...
Search
Enter Lexicon Password...
Enter Postings Password...
120 Documents Available

Figure 22: CipherTXT Storage and Search Engine - Home Page Design.

4.2.3.2.2 Upload Page

Figure 23 denotes the User Interface Design of the Upload Page for the “CipherTXT Storage and Search Engine” application.


CipherTXT Storage and Search Engine
Upload File(s) | Decrypt
Enter Search String...
Search
Enter Lexicon Password...
Enter Postings Password...
120 Documents Available
Lexicon Password:
Lexicon Password Goes Here...
Postings Password:
Postings Password Goes Here...
Document Password:
Document Password Goes Here...
Choose Files
Upload

```
Files To Be Uploaded:
#1 01.TXT
#2 19.TXT
#3 437.TXT
------------------------------
Generating Inverted Index...
Encrypting Document Collection...
Uploading File(s)...
File(s) Uploaded To Server...
```

Figure 23: CipherTXT Storage and Search Engine - Upload Page Design.

4.2.3.2.3 Decrypt Page

Figure 24 denotes the User Interface Design of the Decrypt Page for the “CipherTXT Storage and Search Engine” application.

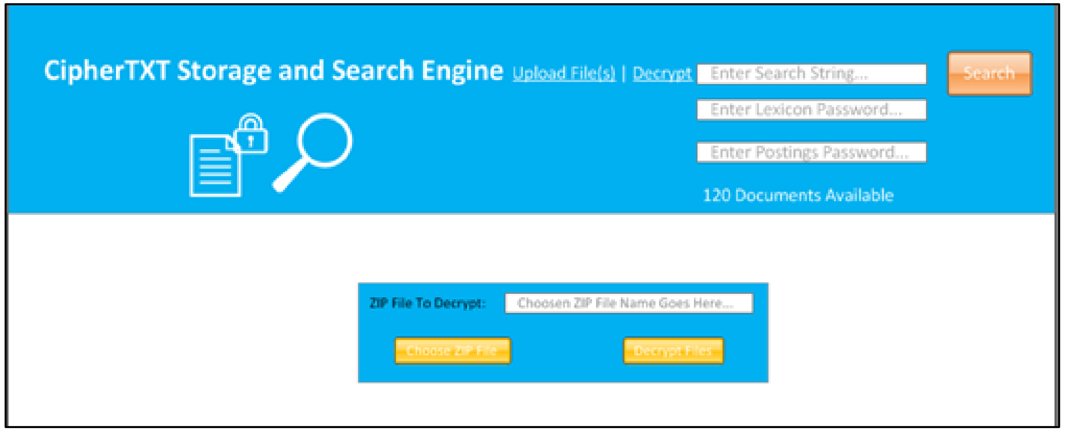


Figure 24: CipherTXT Storage and Search Engine - Decrypt Page Design.

4.3 Low Level Design

Section 4.3.1 denotes the Sequence Diagrams associated with both the “PlainTXT Storage and Search Engine” application as well as the “CipherTXT Storage and Search Engine” application.

4.3.1 Sequence Diagrams

Section 4.3.1.1 denotes the Sequence Diagrams associated with the various Use Cases of the “PlainTXT Storage and Search Engine” application, while Section 4.3.1.2 denotes same for the “CipherTXT Storage and Search Engine”

Components highlighted in red in the following Sequence Diagrams denote Components residing on the Client-Side of the associated functionality, while Components highlighted in blue denote Components residing on the Server-Side of the associated functionality.

4.3.1.1 PlainTXT Storage and Search Engine

Section 4.3.1.1.1 denotes the Sequence Diagram associated with PFR-001, while Section 4.3.1.1.2 denotes the Sequence Diagram associated with PFR-002.

4.3.1.1.1 PFR-001: Upload TXT Document(s) To Server

Due to the size of the Sequence Diagram associated with PFR-001, the author has had to split the Sequence Diagram in to two separate Diagrams. The first half of the Sequence Diagram denotes the functionality from the perspective of the Client (see Figure 25), while the second half of the Sequence Diagram denotes the functionality from the perspective of the Server (see Figure 26).

In relation to the `POST(PT_FILE_UPLOAD_URL)` and `POST(PT_GENERATE_INVERTED_INDEX_URL)` interactions between the `PT_Client_To_Server` and `Web_Server` components in Figure 25, the reader should be aware that the Server Side functionality associated with both interactions has been abbreviated in Figure 25, but is expanded upon in detail in Figure 26.

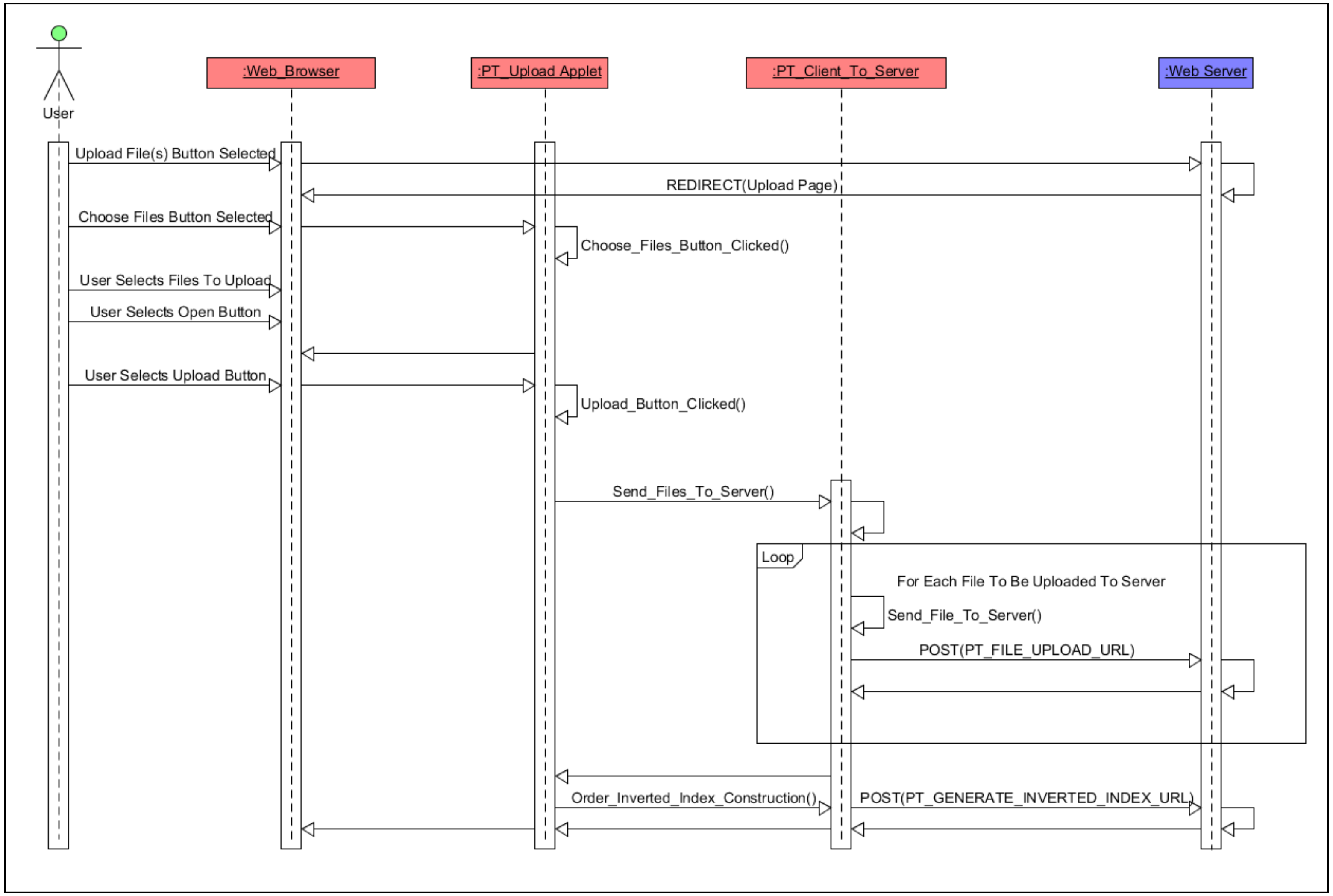

User
:Web_Browser
:PT_Upload Applet
:PT_Client_To_Server
:Web Server
Upload File(s) Button Selected
REDIRECT(Upload Page)
Choose Files Button Selected
Choose_Files_Button_Clicked()
User Selects Files To Upload
User Selects Open Button
User Selects Upload Button
Upload_Button_Clicked()
Send_Files_To_Server()
Loop
For Each File To Be Uploaded To Server
Send_File_To_Server()
POST(PT_FILE_UPLOAD_URL)
Order_Inverted_Index_Construction()
POST(PT_GENERATE_INVERTED_INDEX_URL)

Figure 25: PFR-001 Sequence Diagram (Client Side).

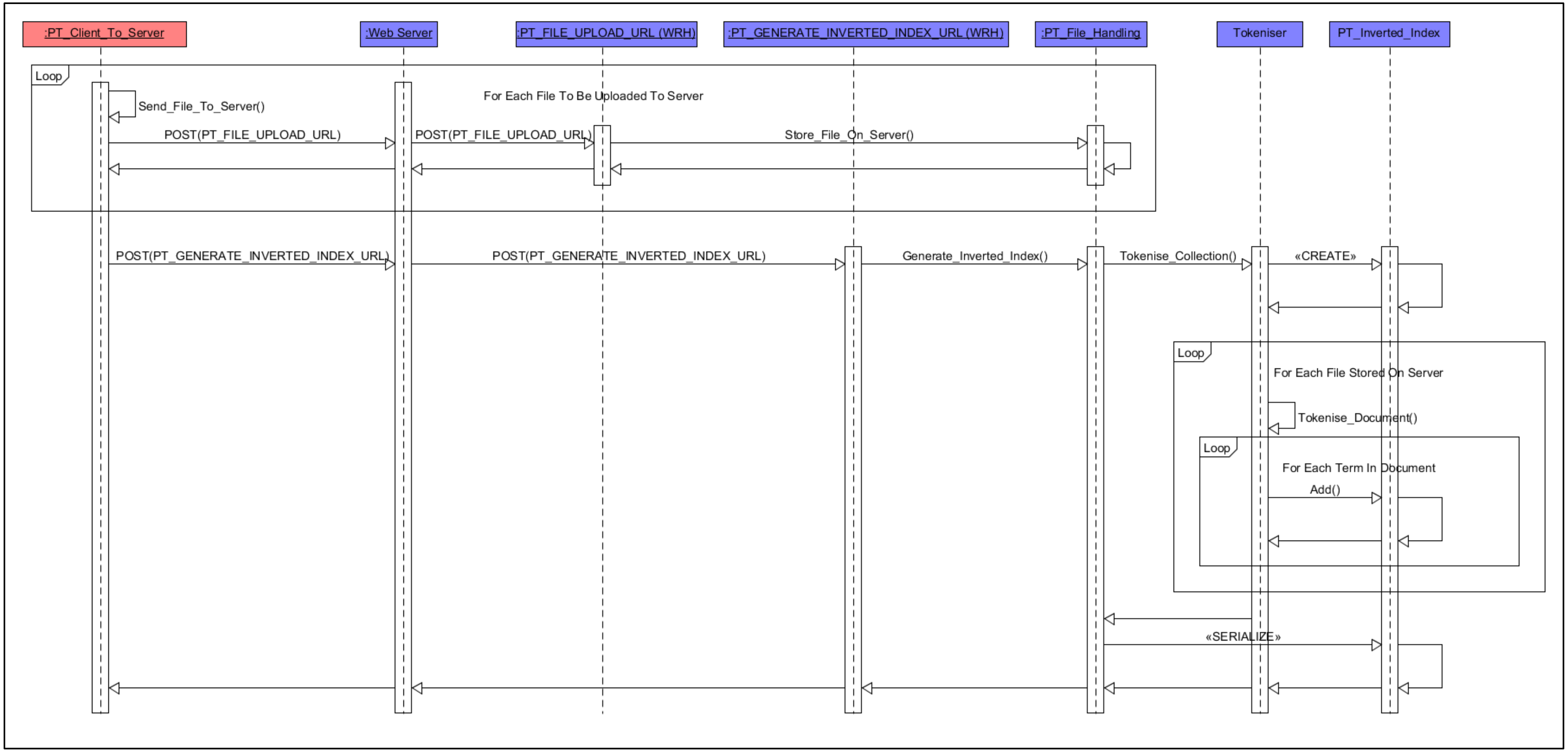

:PT_Client_To_Server
:Web Server
:PT_FILE_UPLOAD_URL (WRH)
:PT_GENERATE_INVERTED_INDEX_URL (WRH)
:PT_File_Handling
Tokeniser
PT_Inverted_Index
Loop
For Each File To Be Uploaded To Server
Send_File_To_Server()
POST(PT_FILE_UPLOAD_URL)
POST(PT_FILE_UPLOAD_URL)
Store_File_On_Server()
POST(PT_GENERATE_INVERTED_INDEX_URL)
POST(PT_GENERATE_INVERTED_INDEX_URL)
Generate_Inverted_Index()
Tokenise_Collection()
«CREATE»
Loop
For Each File Stored On Server
Tokenise_Document()
Loop
For Each Term In Document
Add()
«SERIALIZE»

Figure 26: PFR-001 Sequence Diagram (Server Side).

4.3.1.1.2 PFR-002: Retrieve TXT Documents Containing Specified Search String

Figure 27 denotes the Sequence Diagram associated with PFR-002.

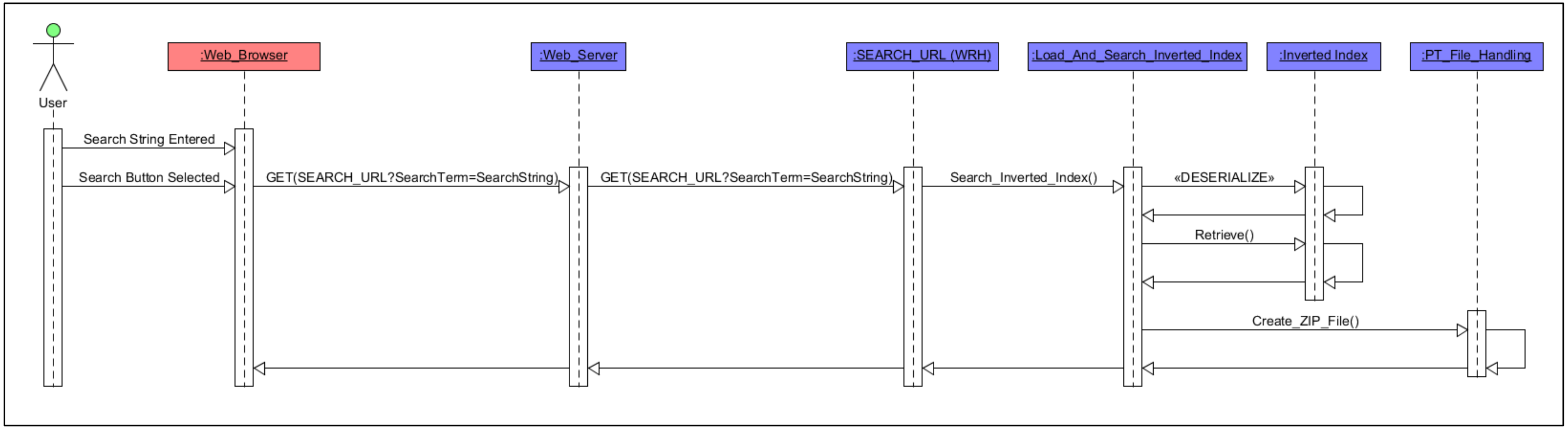


Figure 27: PFR-002 Sequence Diagram.

4.3.1.2 CipherTXT Storage and Search Engine

Section 4.3.1.2.1 denotes the Sequence Diagram associated with CFR-001, Section 4.3.1.2.2 denotes the Sequence Diagram associated with CFR-002, while Section 4.3.1.2.3 denotes the Sequence Diagram associated with CFR-003.

4.3.1.2.1 CFR-001: Encrypt and Upload TXT Document(s) To Server

Due to the size of the Sequence Diagram associated with CFR-001, the author has had to split the Sequence Diagram in to three separate Diagrams. The first third of the Sequence Diagram denotes the functionality from the perspective of the Client (see Figure 28); however the reader should be aware that a number of details associated with the `Generate_SSE_Inverted_Index()` method have been abbreviated for readability purposes (these details can however be seen in Figure 29 – the second third of the Diagram). In addition, a number of details associated with both the `POST(CT_FILE_UPLOAD_URL)` and `POST(CT_ INVERTED_INDEX_UPLOAD_URL)` interactions between the `CT_Client_To_Server` and `Web_Server` components in Figure 28 have been abbreviated, however these are expanded upon in detail in Figure 30 which denotes the functionality from the perspective of the Server (the final third of the Diagram).

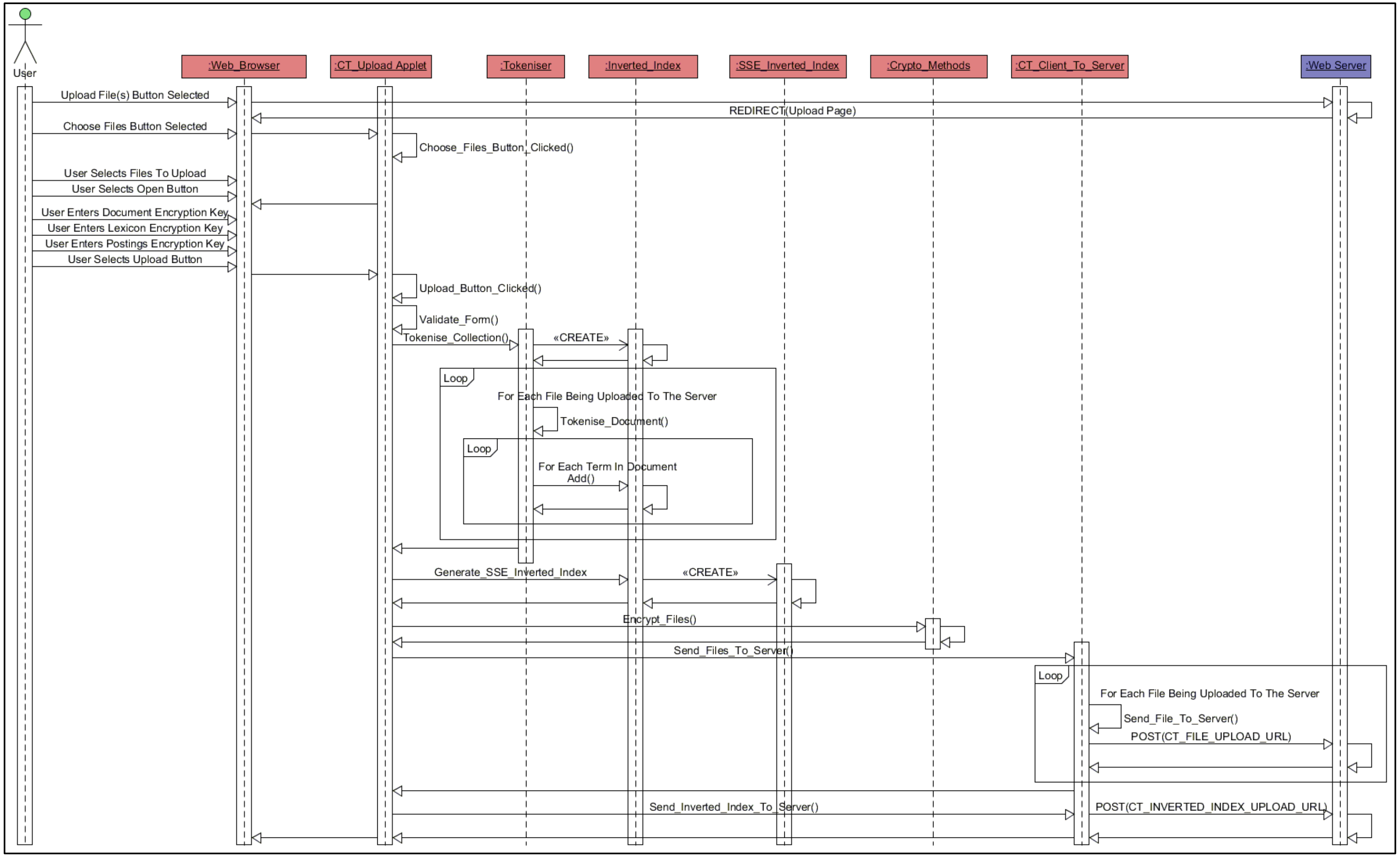

User
:Web_Browser
:CT_Upload Applet
:Tokeniser
:Inverted_Index
:SSE_Inverted_Index
:Crypto_Methods
:CT_Client_To_Server
:Web Server
Upload File(s) Button Selected
REDIRECT(Upload Page)
Choose Files Button Selected
Choose_Files_Button_Clicked()
User Selects Files To Upload
User Selects Open Button
User Enters Document Encryption Key
User Enters Lexicon Encryption Key
User Enters Postings Encryption Key
User Selects Upload Button
Upload_Button_Clicked()
Validate_Form()
Tokenise_Collection()
«CREATE»
Loop
For Each File Being Uploaded To The Server
Tokenise_Document()
Loop
For Each Term In Document
Add()
Generate_SSE_Inverted_Index
«CREATE»
Encrypt_Files()
Send_Files_To_Server()
Loop
For Each File Being Uploaded To The Server
Send_File_To_Server()
POST(CT_FILE_UPLOAD_URL)
Send_Inverted_Index_To_Server()
POST(CT_INVERTED_INDEX_UPLOAD_URL)

Figure 28: CFR-001 Sequence Diagram (Client Side – With `Genereate_SSE_Inverted_Index()` Details Omitted).

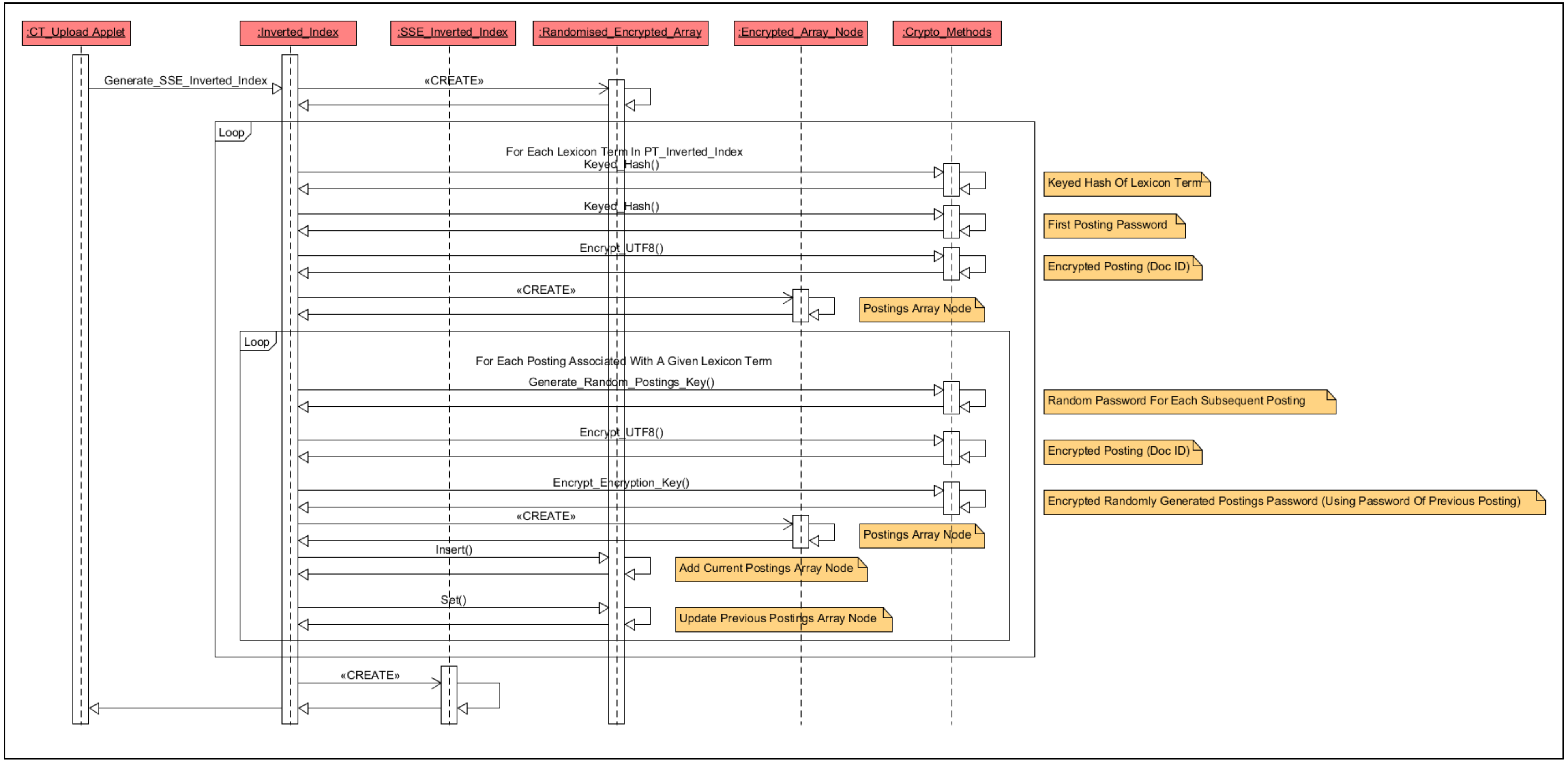

:CT_Upload Applet
:Inverted_Index
:SSE_Inverted_Index
:Randomised_Encrypted_Array
:Encrypted_Array_Node
:Crypto_Methods
Generate_SSE_Inverted_Index
«CREATE»
Loop
For Each Lexicon Term In PT_Inverted_Index
Keyed_Hash()
Keyed Hash Of Lexicon Term
Keyed_Hash()
First Posting Password
Encrypt_UTF8()
Encrypted Posting (Doc ID)
«CREATE»
Postings Array Node
Loop
For Each Posting Associated With A Given Lexicon Term
Generate_Random_Postings_Key()
Random Password For Each Subsequent Posting
Encrypt_UTF8()
Encrypted Posting (Doc ID)
Encrypt_Encryption_Key()
Encrypted Randomly Generated Postings Password (Using Password Of Previous Posting)
«CREATE»
Postings Array Node
Insert()
Add Current Postings Array Node
Set()
Update Previous Postings Array Node
«CREATE»

Figure 29: `Generate_SSE_Inverted_Index()` Sequence Diagram.

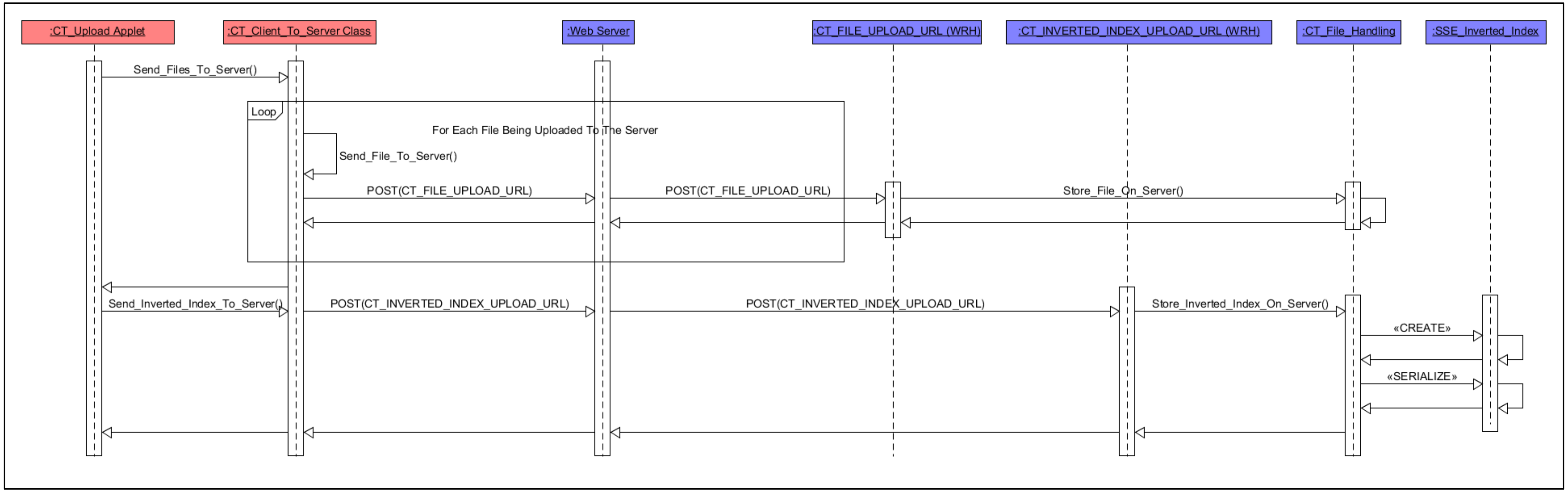

:CT_Upload Applet
:CT_Client_To_Server Class
:Web Server
:CT_FILE_UPLOAD_URL (WRH)
:CT_INVERTED_INDEX_UPLOAD_URL (WRH)
:CT_File_Handling
:SSE_Inverted_Index
Send_Files_To_Server()
Loop
For Each File Being Uploaded To The Server
Send_File_To_Server()
POST(CT_FILE_UPLOAD_URL)
POST(CT_FILE_UPLOAD_URL)
Store_File_On_Server()
Send_Inverted_Index_To_Server()
POST(CT_INVERTED_INDEX_UPLOAD_URL)
POST(CT_INVERTED_INDEX_UPLOAD_URL)
Store_Inverted_Index_On_Server()
«CREATE»
«SERIALIZE»

Figure 30: CFR-001 Sequence Diagram (Server Side).

4.3.1.2.2 CFR-002: Retrieve TXT Documents Containing Encrypted Specified Search String Use Case

Figure 31 denotes the Sequence Diagram associated with CFR-002.

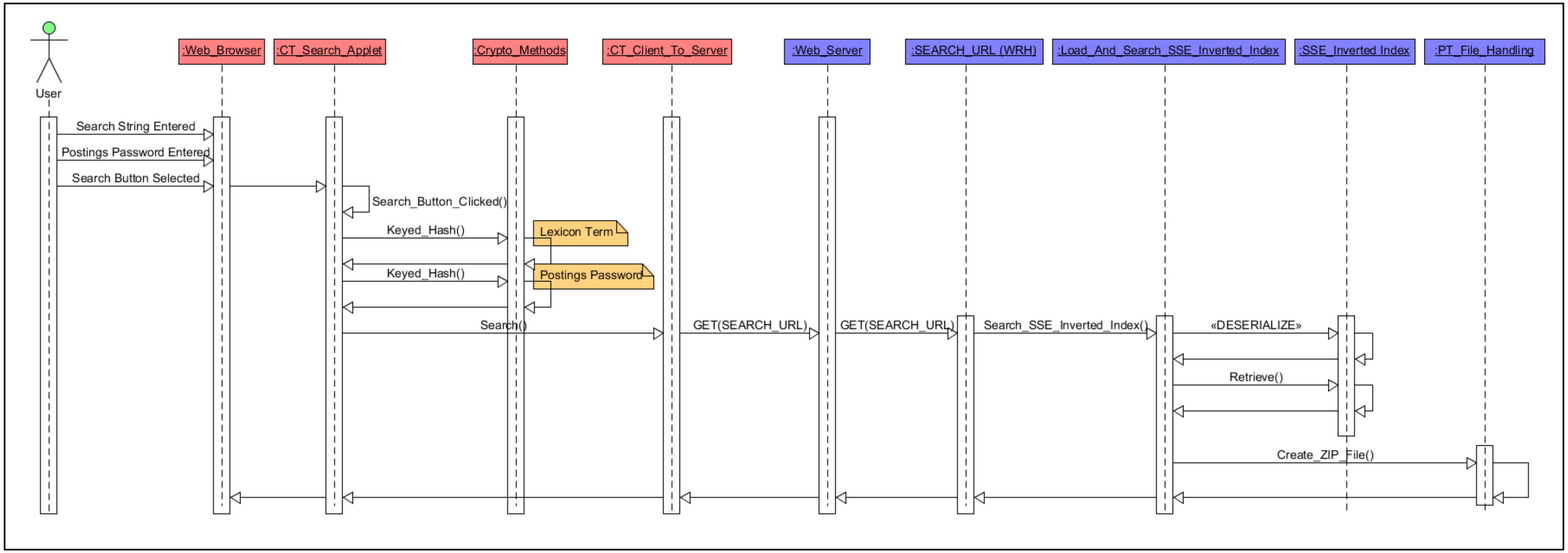


Figure 31: CFR-002 Sequence Diagram.

4.3.1.2.3 CFR-003: Decrypt Encrypted TXT Documents Retrieved From Server

Figure 32 denotes the Sequence Diagram associated with CFR-003.

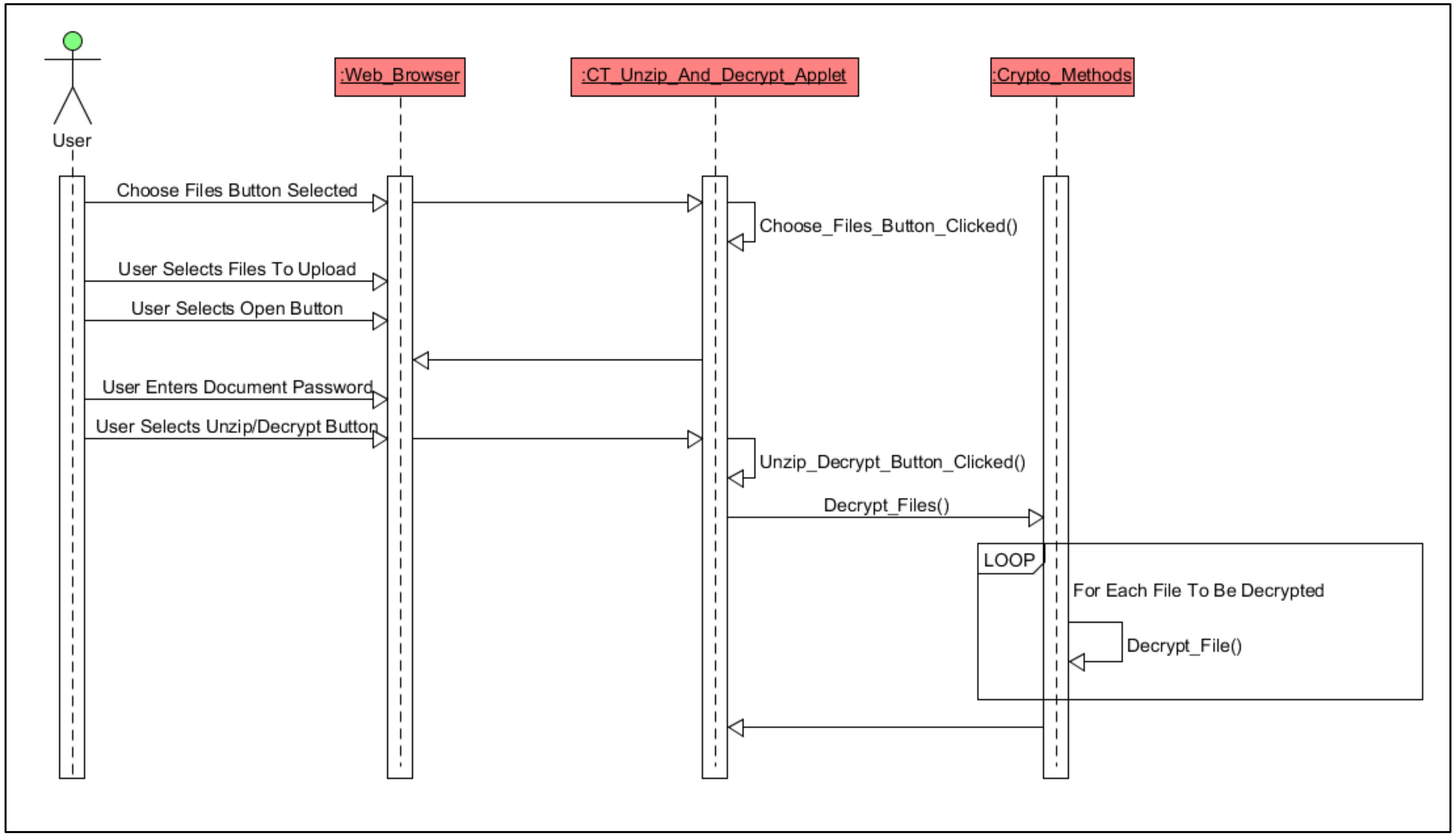

User
:Web_Browser
:CT_Unzip_And_Decrypt_Applet
:Crypto_Methods
Choose Files Button Selected
Choose_Files_Button_Clicked()
User Selects Files To Upload
User Selects Open Button
User Enters Document Password
User Selects Unzip/Decrypt Button
Unzip_Decrypt_Button_Clicked()
Decrypt_Files()
LOOP
For Each File To Be Decrypted
Decrypt_File()

Figure 32: CFR-003 Sequence Diagram.

5. Implementation

Both the “PlainTXT Storage and Search Engine” and “CipherTXT Storage and Search Engine“ applications developed as part of this dissertation were implemented using the Java Programming Language. All Client-Side functionality associated with both applications was implemented in the form of Java Applets, while all Server-Side functionality was implemented in the form of Java Servlets.

The SSE scheme underlying the “CipherTXT Storage and Search Engine” application is the scheme described previously in the Literature Review (Chapter 2); *that is, Kamara et al. (2012)*. As such, all Data Structures and Security measures outlined previously in Section 2.3 have been applied and utilised in the implementation of SSE developed as part of this dissertation.

The core functionality of the “CipherTXT Storage and Search Engine” application is contained within the following Java Classes and Java Methods:

- `Tokeniser.JAVA` (see Section 5.1)
- `Inverted_Index.JAVA` (see Section 5.1)
- `Crypto_Methods.JAVA` (see Section 5.2)
- `Generate_SSE_Inverted_Index() Method` (Contained Within `Inverted_Index.JAVA` - see Section 5.3)

- SSE_Inverted_Index.JAVA (see Section 5.3)
- Encrypted_Array_Node.JAVA (see Section 5.3)
- Randomised_Encrypted_Array.JAVA (see Section 5.3)
- Retrieve() Method (Contained Within SSE_Inverted_Index.JAVA - see Section 5.4)

5.1 Tokeniser.JAVA and Inverted_Index.JAVA

The Tokeniser Class is responsible for Document Tokenisation in both the "PlainTXT Search and Storage Engine" and "CipherTXT Storage and Search Engine" applications developed as part of this dissertation.

An array of File Objects (representing the chosen Document Collection) is first passed into the Tokenise_Collection() method of the Tokeniser Class, which then proceeds to tokenise each Document on a one-by-one basis.

The individual Terms contained within each TXT file are retrieved using the Scanner Class and its next() Method. Each Term encountered during

Document Tokenisation is passed to an Inverted_Index Object (which comprises a HashMap<String, HashSet<Integer>> Object)[23].

Before a given Term is added into the underlying HashMap Object, the HashMap is first examined to determine whether or not the Term was added to the HashMap previously[24]. In the event that a Term was not added to the HashMap previously, the Term is simply inserted directly in to the HashMap along with a HashSet Object comprising the DocID of the Document currently being tokenised. In the event that the Term was present in the HashMap previously, the Terms associated HashSet is instead retrieved from the HashMap and then updated to include the ID of the Document currently being tokenised (before then being re-inserted into the HashMap).

5.2 Crypto_Methods.JAVA

All cryptographic functionality associated with the "CipherTXT Storage and Search Engine" application is contained within the Crypto_Methods.JAVA Class.

The Crypto_Methods.JAVA Class comprises the following methods:

- Keyed_Hash()

[23] String => Lexicon Term; HashSet<Integer> => Posting List associated with Lexicon Term.

[24] Simply adding the Term directly to the HashMap will overwrite the existing entry (if any) (including the set of Document IDs associated with the Term previously).

- Generate_Random_Postings_Key()
- Encrypt_UTF8()
- Decrypt_UTF8()
- Encrypt_Encryption_Key()
- Decrypt_Decryption_Key()
- Derive_Key()
- Encrypt_Files()
- Decrypt_Files()

The Keyed_Hash() method of the Crypto_Methods.JAVA Class comprises an instance of the built in Java Mac Class. The Mac Object is configured to generate keyed hash values using the HMAC-MD5 algorithm. The Keyed_Hash() method is used for two purposes in the implementation of SSE developed as part of this dissertation: 1) To generate a keyed hash for each Lexicon Term within the SSE Inverted Index, and 2) To generate the encryption/decryption key used to encrypt/decrypt the first Posting associated with each Lexicon Term. The method returns a Base64 encoded String representation of the keyed hash value generated.

The Generate_Random_Postings_Key() method of the Crypto_Methods.JAVA Class comprises an instance of the built in Java SecureRandom Class. The SecureRandom Object is configured to generate a randomised 128 bit value which can be used to encrypt the second (and all subsequent) Posting associated with a given Lexicon Term. The method returns a Base64 encoded `String` representation of the key generated.

The `Encrypt_UTF8()` and `Decrypt_UTF8()` methods of the `Crypto_Methods.JAVA` Class comprise two instances of the built in Java `Cipher` Class. The `Cipher` Object in the `Encrypt_UTF8()` Method is configured to encrypt UTF8 encoded `String`s into Base64 encoded `String`s using AES/CBC/PKCS5Padding encryption, while the `Cipher` Object in the `Decrypt_UTF8()` Method is configured to decipher encrypted Base64 encoded `String`s into plaintext UTF8 encoded `String`s using same. The `Encrypt_UTF8()` and `Decrypt_UTF8()` Methods are used to encrypt and decrypt Postings and Posting Pointers in the implementation of SSE developed as part of this dissertation.

The `Encrypt_Encryption_Key()` and `Decrypt_Decryption_Key()` methods of the `Crypto_Methods.JAVA` Class comprise two instances of the built in Java `CipherInputStream` Class. The `CipherInputStream` Object in the `Encrypt_Encryption_Key()` Method is configured to convert Base64 encoded `String`s into encrypted Base64 encoded `String`s using AES/CBC/PKCS5Padding encryption, while the `CipherInputStream` Object in the `Decrypt_Decryption_Key()` Method is configured to convert encrypted Base64 encoded `String`s into plaintext Base64 encoded `String`s using same. The `Encrypt_Encryption_Key()` and `Decrypt_Decryption_Key()`

Methods are used to encrypt and decrypt Posting keys generated by the `Generate_Random_Postings_Key()` Method outlined previously.

The `Derive_Key()` method of the `Crypto_Methods.JAVA` Class comprises an instance of the built in Java `SecretKeyFactory` Class. The `SecretKeyFactory` Object is configured to generate a secret key value using the PBKDFHMAC-SHA1 algorithm. The `Derive_Key()` method of the `Crypto_Methods.JAVA` Class is used to derive encryption/decryption keys for TXT file encryption in the implementation of SSE developed as part of this dissertation.

The `Encrypt_Files()` and `Decrypt_Files()` methods of the `Crypto_Methods.JAVA` Class comprise two instances of the built in Java `Cipher` Class. The `Cipher` Object in the `Encrypt_Files()` Method is configured to convert plaintext TXT files into encrypted TXT files using AES/CBC/PKCS5Padding encryption, while the `Decrypt_Files()` Method is configured to convert encrypted TXT files into plaintext TXT files using same.

5.3 Generate_SSE_Inverted_Index() Method, SSE_Inverted_Index.JAVA, Encrypted_Array_Node.JAVA, and Randomised_Encrypted_Array.JAVA.

The Generate_SSE_Inverted_Index() Method of the Inverted_Index Class is responsible for converting an IR Inverted Index into an SSE Inverted Index in the implementation of SSE developed as part of this dissertation.

As part of the process of generating an SSE Inverted Index, the Generate_SSE_Inverted_Index() Method first iterates over all entries within the HashMap<String, HashSet<Integer>> Object underlying the Inverted_Index.

For each Lexicon Term contained within the plaintext IR HashMap, a Keyed_Hash() is generated for the Term, along with a password which will be used later to encrypt the first Posting associated with the Lexicon Term (also generated using the Keyed_Hash() method; albeit with a different password).

Following this, the set of all Posting associated the Lexicon Term are then retrieved and iterated over; *that is, the HashSet<Integer> Object contained within the IR HashMap Object*.

For each Posting encountered, the associated Document ID is then encrypted and added to an Encrypted_Array_Node Object. In the case of the first Posting, the Posting is encrypted using the second Keyed_Hash() value generated from its associated Lexicon Term, while all subsequent Postings are encrypted using keys generated using the Generate_Random_Postings_Key() Method of the Crypto_Methods Class (Note that each Encrypted_Array_Node Object is encrypted using a different randomised key) .

The Encrypted_Array_Node Object associated with the first Posting is stored alongside its associated Lexicon Term within the HashMap<String, Encrypted_Array_Node> Object contained with the SSE_Inverted_Index, while all subsequent Postings are stored within the Randomised_Encrypted_Array Object contained with the SSE_Inverted_Index.

In addition to storing encrypted Postings, each Encrypted_Array_Node Object also stores the location of the next Encrypted_Array_Node Object associated with the Lexicon Term in question, as well as the key necessary to decrypt the contents of the next Encrypted_Array_Node Object. As such, whenever a new Encrypted_Array_Node Object is created, its associated encryption key is then stored in the Encrypted_Array_Node Object that preceded it in the list of Postings (with the exception of the first Encrypted_Array_Node Object associated with a Lexicon Term - SSE requires the user to be able to manually generate this key as and when needed). In addition, whenever an

Encrypted_Array_Node Object is inserted into the Randomised_Encrypted_Array Object, the randomised Array Index assigned to the Encrypted_Array_Node Object is then stored in the Encrypted_Array_Node Object that preceded it in the list of Postings.

5.4 Retrieve() Method

The Retrieve() Method of the SSE_Inverted_Index Class is responsible for searching[25] the SSE Inverted Index associated with the implementation of SSE developed as part of this dissertation.

Prior to executing the Retrieve() Method, the user must first generate a Keyed_Hash() of their Search Term, as well as the Postings Password associated with their Search Term (again, generated using the Keyed_Hash() Method).

The Keyed_Hash() associated with the users Search Term is then used to lookup the HashMap<String, Encrypted_Array_Node> Object contained with the SSE_Inverted_Index Class.

Should the Search Term be present in the HashMap, the associated Encrypted_Array_Node Object is then returned from the HashMap and decrypted using the Postings Password generated by the user.

[25] Note that searching the SSE Inverted Index also included identifying and decrypting all Postings associated with the Lexicon Term searched for.

Decrypting the Encrypted_Array_Node reveals three pieces of information:

- The Document ID associated with the Posting,
- The index location of the next Encrypted_Array_Node Object associated with the Search Term (contained within the Randomised_Encrypted_Array associated with the SSE_Inverted_Index).
- The key required to decrypt the next Encrypted_Array_Node.

The process then repeats until a Randomised_Encrypted_Array Node is found that contains no index location for a next Posting. At this point, the set of all Postings associated with the Search Term have been identified and decrypted.

6. Testing

Section 6.1 denotes the details associated with the hardware/software used during Testing, Section 6.2 denotes the details associated with the Data Sets used during Testing, while Section 6.3 denotes the Experimental Results obtained during Testing.

6.1 Test Environment

Table 16 denotes the details associated with the pertinent software utilised during application Testing, while Table 17 denotes the details associated with the hardware utilised during application Testing.

Operating System:	Windows Ultimate 64-Bit SP1

Java Development Kit (JDK):
Java Runtime Environment (JRE):
	Java Version: 8
	Update: 51
	Build: 16

Web Server (Localhost):	Apache Tomcat 7.0.56 *Included As Part Of XAMMP 5.6.8 Package*

Table 16: Test Environment - Pertinent Software Details.

Device Type:	Laptop
Processor:	Intel Core i7 4900MQ @2.8GHz *Quad Core*
Motherboard:	Notebook W35xSS_370SS Motherboard
RAM:	24GB RAM (3 X 8GB KINGSTON DDR3 @ 800MHz)
Hard Disk:	925GB SSHD
RAID:	RAID 1 (Software Based RAID)

Table 17: Test Environment - Hardware Details.

All tests were conducted using the default Java Virtual Machine (JVM) - no additional runtime parameters were configured.

6.2 Test Data

All experiments conducted as part of this dissertation were performed on the ’20 Newsgroups’ Data Set (Rennie, 2008).

In its original form, the ’20 Newsgroups’ Data Set consists of 18,828 files, subdivided into 20 folders. Initially, each file in the Data Set has a numeric file name between 4 and 6 digits in length[26] with no file extension.

[26] Recall previously from the Software Requirements Specification that it is assumed that each file in the Test Data Set has a unique numeric name assigned to it.

Prior to being used in the experiments, the author first attempted to move all files in the Data Set into a single folder; however at this point the author noted that the names of all files in the Data Set are not unique (the contents of each file are unique however (Rennie, 2008)). In an effort to avoid duplicate file names, the author randomly assigned an 8 digit numeric name to each file in the Data Set[26]. In addition, the author also appended the TXT file extension to each file in the Data Set.

As part of Testing, the author tested each aspect of SSE with Data Sets that increased in size by an order of magnitude. As such, it was necessary to derive smaller subsets from the full '20 Newsgroups' Test Data Set. In total, 5 subsets were derived (DS1 – DS5). The details associated with each subset – and the full Data Set (DS6) – can be seen in Table 18.

Data Set Name	DS1	DS2	DS3	DS4	DS5	DS6
# of Docs	1	10	100	1,000	10,000	18,828
# of Terms	320	2,612	33,611	281,363	2,738,580	5,130,520
# of Unique Terms	206	1,297	10,996	52,134	258,463	377,880
# of Postings In Data Set	206	1,650	19,838	168,768	1,672,576	3,138,449
# of Postings Associated With Highest Frequency Lexicon Term	1 *All Terms*	10 *And*	100 *Subject:*	1,000 *From:*	10,000 *Subject:*	18,828 *Subject:*
Size	1.9KB	16.1 KB	215KB	1.7MB	17.3MB	32.3MB

Table 18: Test Data Set Statistics[27].

6.3 Experimental Results

Section 6.3.1 denotes the Experimental Results associated with SSE Inverted Index Construction, Section 6.3.2 denotes the Experimental Results associated with SSE Inverted Index Searching, while Section 6.3.3 denotes the Experimental Results associated with the comparison of SSE and plaintext Information Retrieval (IR).

Please note that all Results presented in this dissertation represent average values obtained over ten executions of each experiment.

6.3.1 SSE Inverted Index Construction

Section 6.3.1.1 denotes the Experimental Results associated with generating a plaintext Information Retrieval (IR) Inverted Index and Section 6.3.1.2 denotes the

[27] The statistics shown in Table 18 were generated by executing the `Stat_Counter.java` program on each Data Set.

Experimental Results associated with converting an IR Inverted Index into an SSE Inverted Index. Section 6.3.1.3 denotes the Experimental Results associated with encrypting an entire Document Collection, while Section 6.3.1.4 and Section 6.3.1.5 denote the Experimental Results associated with uploading both the encrypted Document Collection and the SSE Inverted Index to the Server. Section 6.3.1.6 presents a set of aggregate Experimental Results (from Section 6.3.1.1 – Section 6.3.1.5) covering the set of all activities associated with constructing an SSE Inverted Index.

6.3.1.1 IR Inverted Index Construction

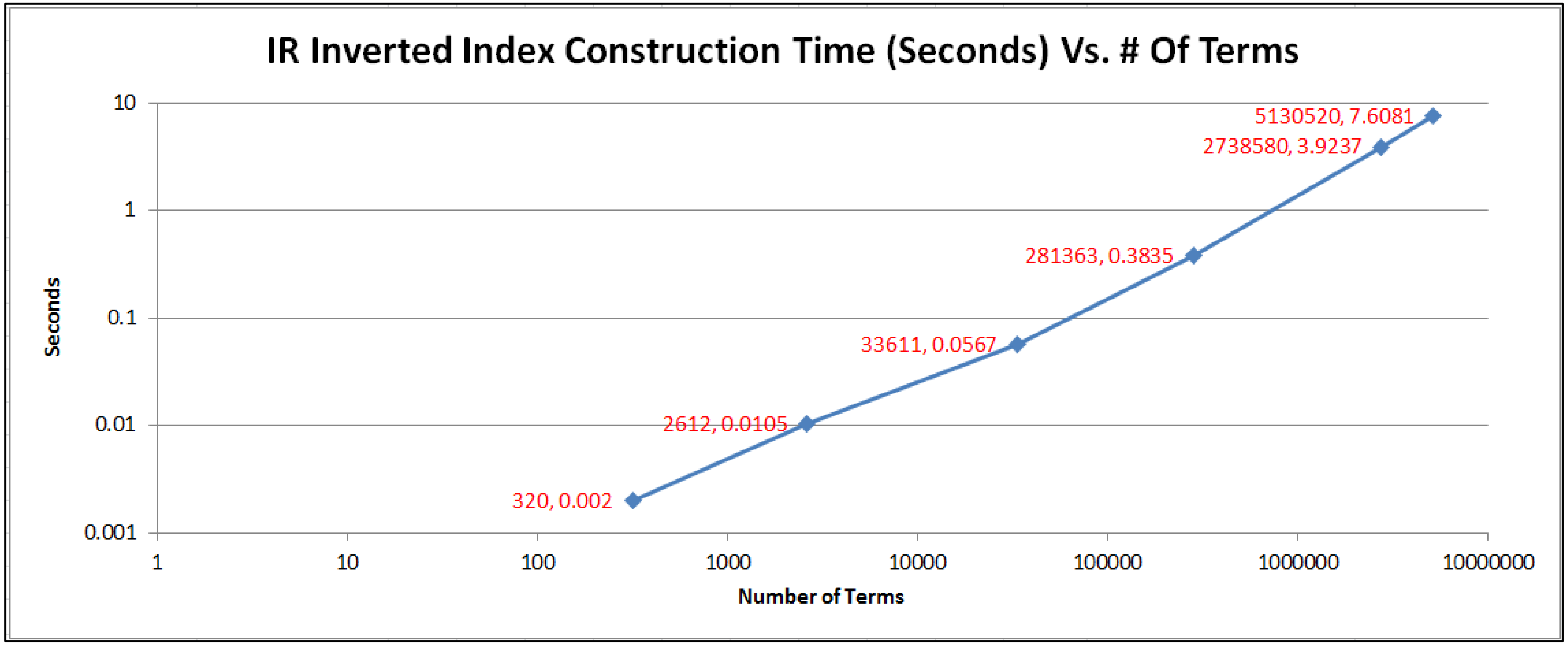


Figure 33: Information Retrieval (IR) Inverted Index Construction Time vs. Number of Terms in Collection.

Figure 33 denotes the Experimental Results associated with generating a plaintext Information Retrieval (IR) Inverted Index for each Test Data Set outlined previously. Figure 33 compares the time taken to generate the IR Inverted Index against the number of Terms in the Document

Collection; *that is, the Test Data Set*, from which the IR Inverted Index is being generated. As can be seen in Figure 33, the time associated with constructing an IR Inverted Index appears to increase linearly as the number of Terms in the underlying Document Collection increases. In relation to Test Data, an IR Inverted Index was generated for Test Data Set 6 (approximately 5 million Terms) in approximately 7.6 seconds.

The Results shown in Figure 33 were obtained by executing `IR_Inverted_Index_Construction_Time.java` on each Data Set outlined previously.

6.3.1.2 SSE Inverted Index Construction

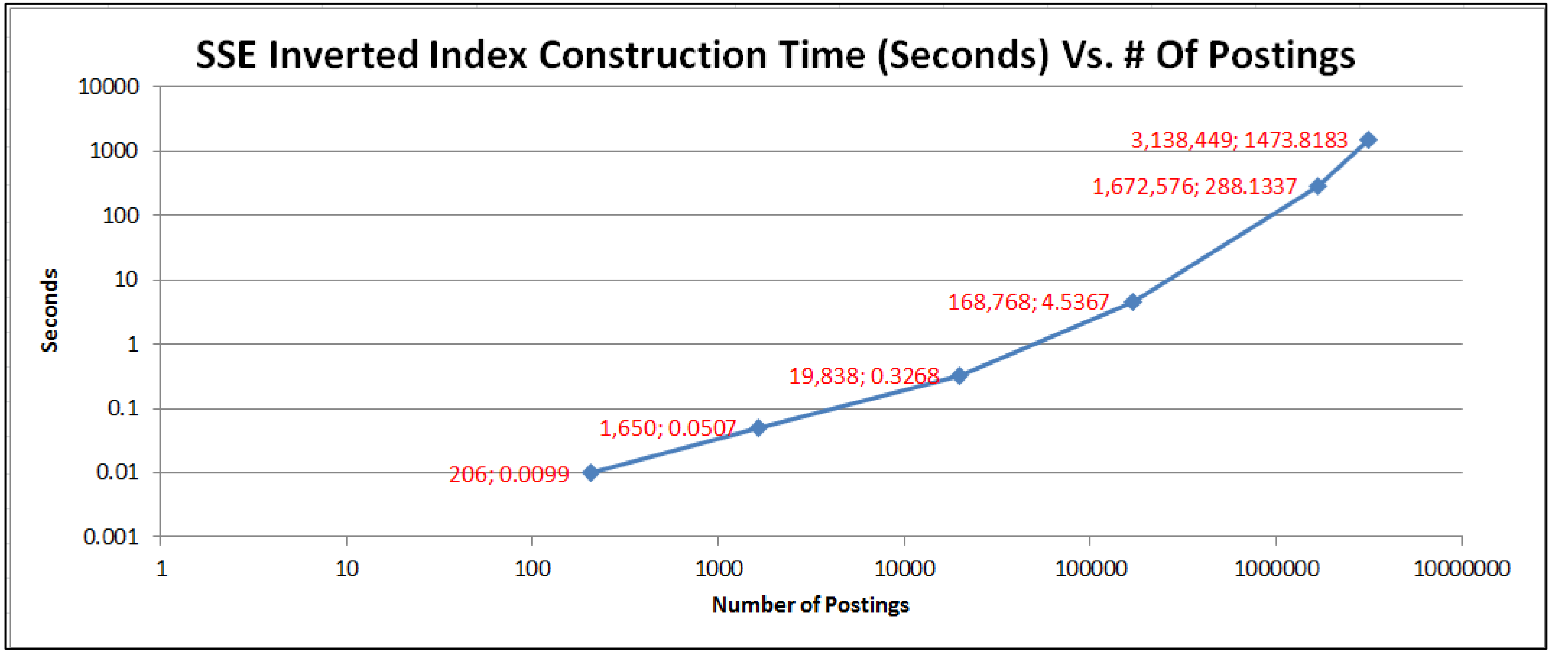


Figure 34: SSE Inverted Index Construction Time vs. No of Postings in IR Inverted Index.

Figure 34 denotes the Experimental Results associated with converting a plaintext Information Retrieval (IR) Inverted Index into an SSE Inverted Index for each Data Set outlined previously. Figure 34 compares the time taken to generate the SSE Inverted Index against the number of

Postings in the IR Inverted Index from which the SSE Inverted Index is generated. For the first four Test Data Sets (DS1 – DS4), the time associated with constructing an SSE Inverted Index appears to increase linearly as the number of Postings in the underlying IR Inverted Index increases; however the time taken to generate an SSE Inverted Index for DS5 and DS6 increases dramatically (when compared to the number of Postings in the underlying IR Inverted Index). In relation to Test Data Sets, an SSE Inverted Index was generated for Test Data Set 4 (281,363 Postings – approximately 3.2 Postings per Lexicon Term) in 1.5 seconds. For Test Data Set 5 (1,672,576 Postings – approximately 6.5 Postings per Lexicon Term), an SSE Inverted Index was generated in 4 minutes 48 seconds. For Test Data Set 6 (3,138,449 Postings – approximately 8.3 Postings per Lexicon Term), an SSE Inverted Index was generated in 24 minutes 34 seconds.

The Results shown in Figure 34 were obtained by executing `SSE_Inverted_Index_Construction_Time.java` on each Data Set outlined previously.

6.3.1.3 Document Collection Encryption

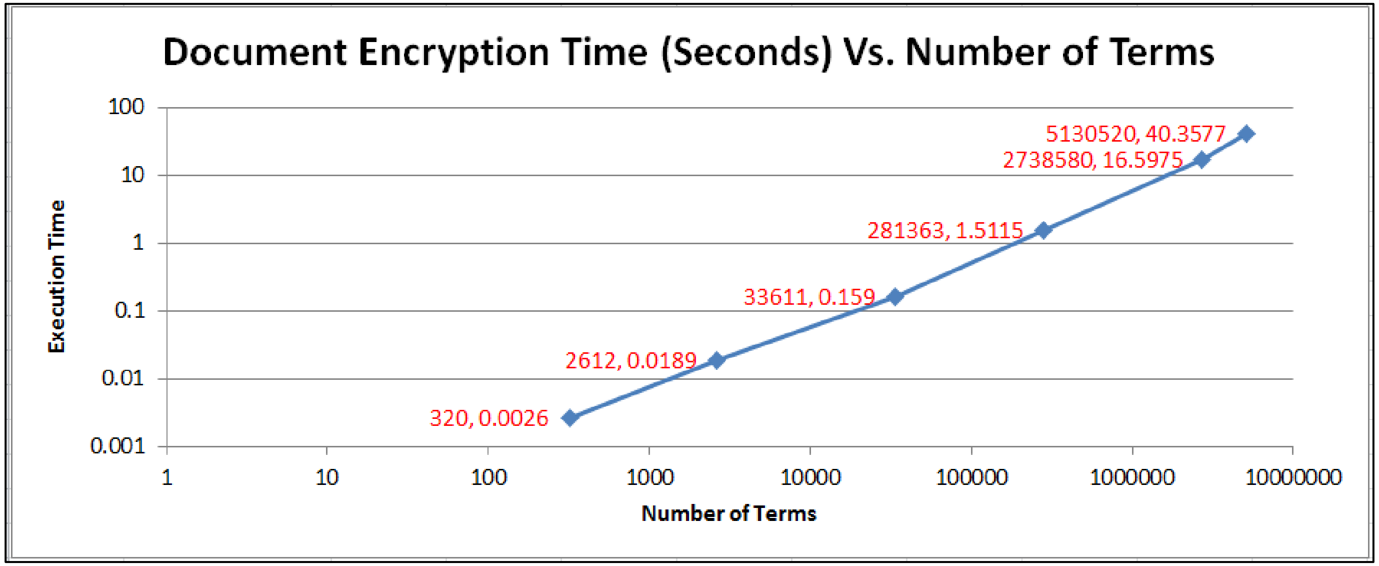


Figure 35: Document Collection Encryption Time vs. Number of Terms in Collection.

Figure 35 denotes the Experimental Results associated with encrypting the Document Collections comprising each of the Test Data Sets. Figure 35 compares the time taken to encrypt each Document Collection against the total number of Terms contained within each Document Collection. As can be seen in Figure 35, the time associated with encrypting the Document Collection appears to increase linearly as the number of Terms in the underlying Document Collection increases. In relation to Test Data Sets, the Document Collection associated with Test Data Set 6 was encrypted in 40 seconds.

The Results shown in Figure 35 were obtained by executing `File_Encryption_Time.java` on each Data Set outlined previously.

6.3.1.4 SSE Inverted Index Upload

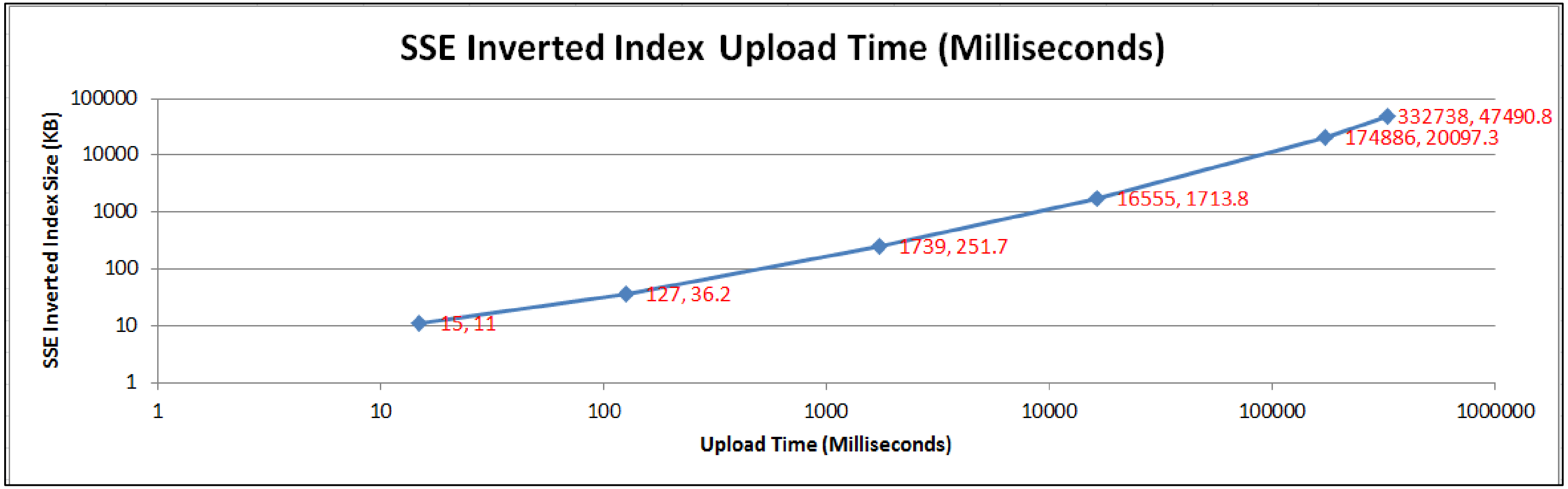


Figure 36: SSE Inverted Index Upload Time vs. Size of SSE Inverted Index.

Figure 36 denotes the Experimental Results associated with uploading an SSE Inverted Index (generated from each Test Data Set) to the Server. Figure 36 compares the time taken to upload the SSE Inverted Index to the Server against the size of the SSE Inverted Index. As can be seen in Figure 36, the time associated with uploading the SSE Inverted Index to the Server appears to increase linearly as the size of the SSE Inverted

Index increases. In relation to Test Data, the SSE Inverted Index associated with Test Data Set 6 (325MB) was uploaded to the Server in 47.5 seconds.

The reader should be aware that the Experimental Results presented in Figure 36 includes the time taken to upload the SSE Inverted Index to the Server, as well as the time taken to serialise the SSE Inverted Index to disk (once the SSE Inverted Index has been received by the Server).

The Results shown in Figure 36 were obtained by executing `SSE_Upload_Timer.java`.

6.3.1.5 Encrypted Document Collection Upload

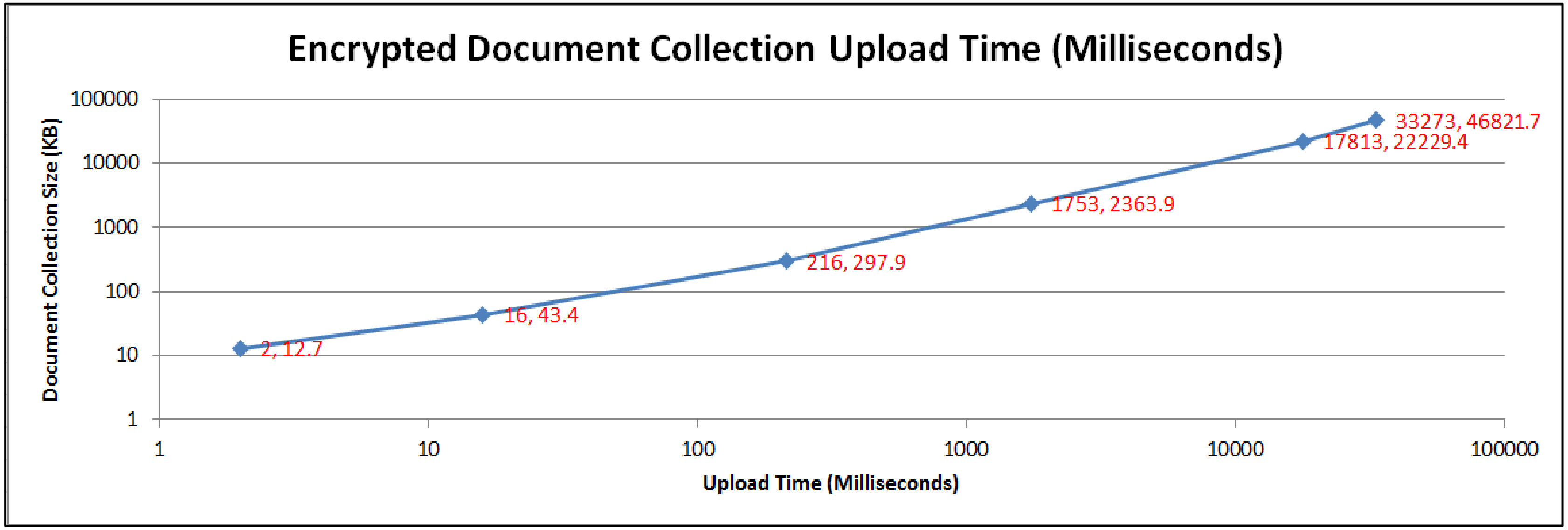


Figure 37: Encrypted Document Collection Upload Time vs. Encrypted Document Collection Size.

Figure 37 denotes the Experimental Results associated with uploading an encrypted Document Collection (generated from each Test Data Set) to the Server. Figure 37 compares the time taken to upload the encrypted Document Collection to the Server against the size of the encrypted Document Collection. As can be seen in Figure 37, the time associated with uploading the encrypted Document Collection to the Server appears to increase linearly as the size of the encrypted Document Collection increases. In relation to Test Data, the encrypted Document Collection associated with Test Data Set 6 (32.5MB) was uploaded to the Server in 46.8 seconds.

The reader should be aware that the Experimental Results presented in Figure 37 include the time taken to upload the encrypted Document Collection to the Server, as well as the time taken to store the encrypted Document Collection on disk (once the encrypted Document Collection has been received by the Server).

The Results shown in Figure 37 were obtained by executing `SSE_Upload_Timer.java`; *that is, the same software as used in Section 6.3.1.4 previously*.

6.3.1.6 Aggregate Results

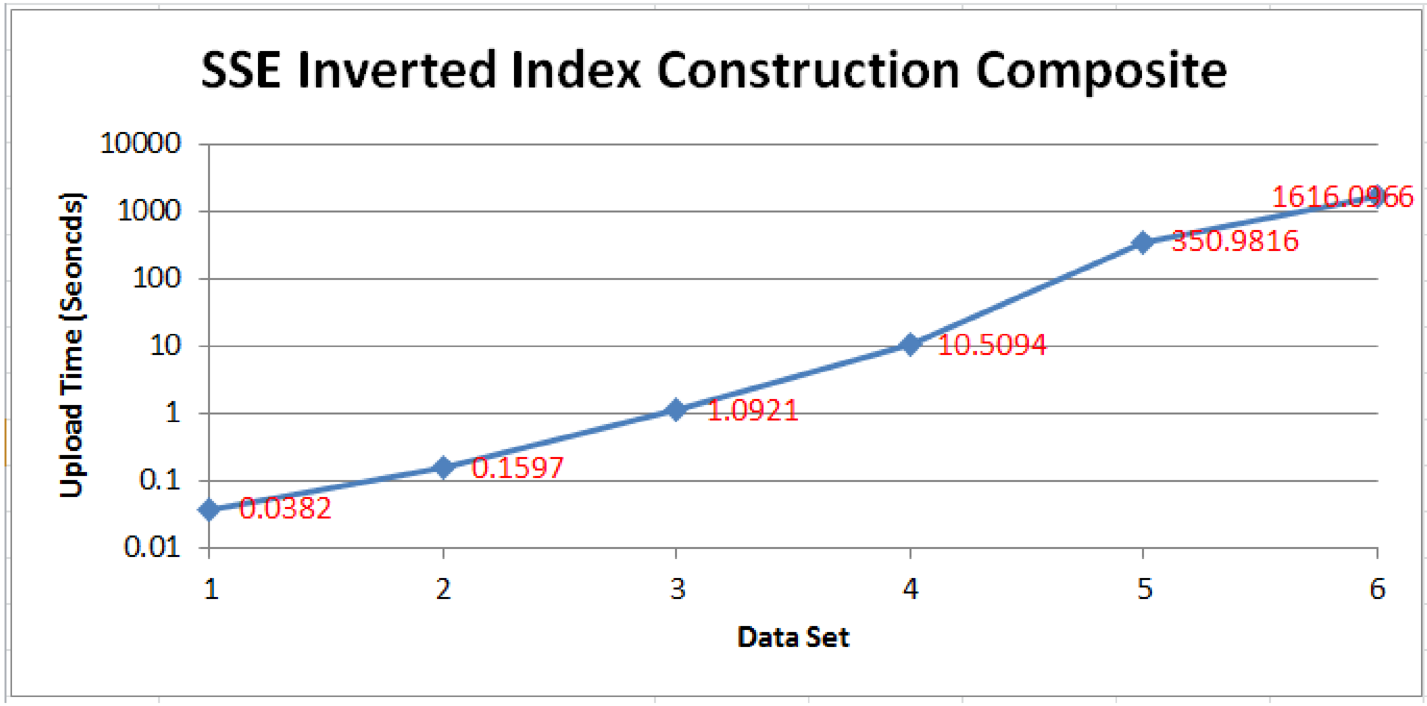


Figure 38: SSE Inverted Index Construction Composite.

Figure 38 denotes the total time taken to create an IR Inverted Index, convert it to an SSE Inverted Index, encrypt the associated Document Collection and upload both the SSE Inverted Index and the encrypted Document Collection to the Server for each Test Data Set outlined previously.

In relation to Test Data, the whole process of constructing an SSE Inverted Index and uploading all associated data to the Server took 10.5 seconds for Test Data Set 4. To carry out the same work on Test Data Set 5 took 5 minutes 50 seconds, while carrying out the same work on Test Data Set 6 took 26 minutes 56 seconds.

6.3.2 SSE Inverted Index Querying

Section 6.3.2.1 denotes the Experimental Results associated with generating an Encrypted Search String (ESS) for SSE, while Section 6.3.2.2 denotes the Experimental Results associated with searching an SSE Inverted Index. Finally, Section 6.3.2.3 denotes the Experimental Results associated with searching the SSE Inverted Index and downloading all matching Documents to the Client (in ZIP File Format).

6.3.2.1 ESS Generation

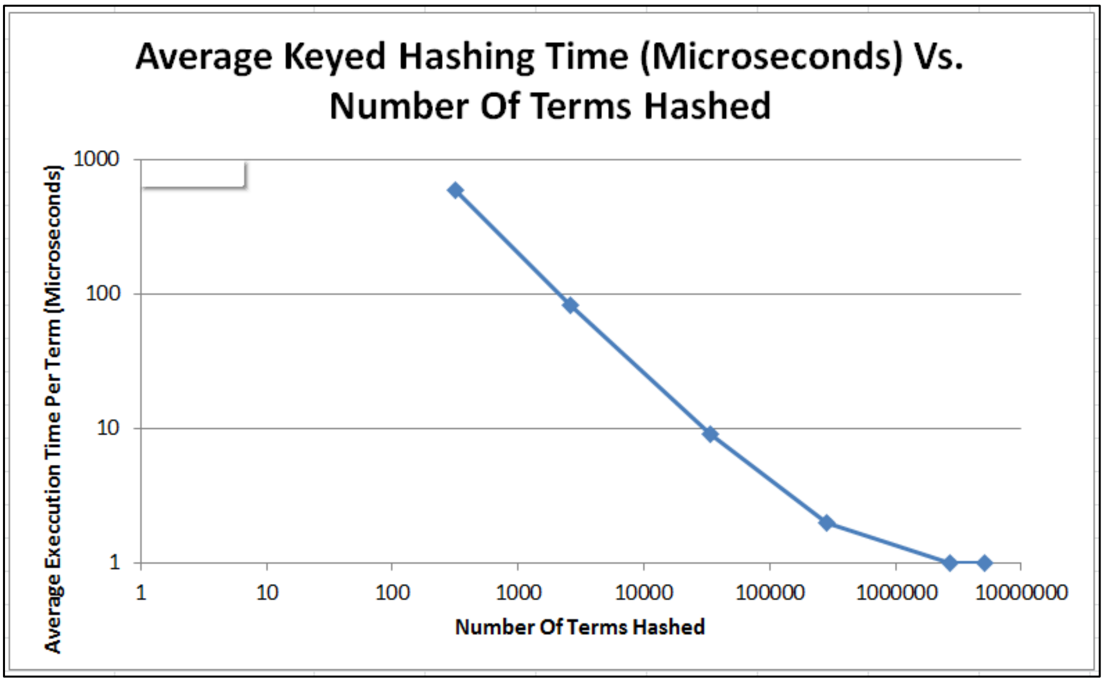


Figure 39: Encrypted Search String (EES) Generation Time vs. Number of Terms in Document Collection.

Figure 39 denotes the Experimental Results associated with generating Encrypted Search Strings (ESS) for SSE.

For each Lexicon Term within the Test Data Sets outlined previously, an ESS; *that is, a keyed hash*, was generated. As can be seen in Figure 39, the time taken to generate an ESS is by no means constant. The Experimental Results appear to show that the more ESS that are generated, the faster the execution time of the underlying `Keyed_Hash()` Method.

The Results shown in Figure 37 were obtained by executing `ESS_Generation_Time.java`.

6.3.2.2 Identifying and Decrypting Matching Postings

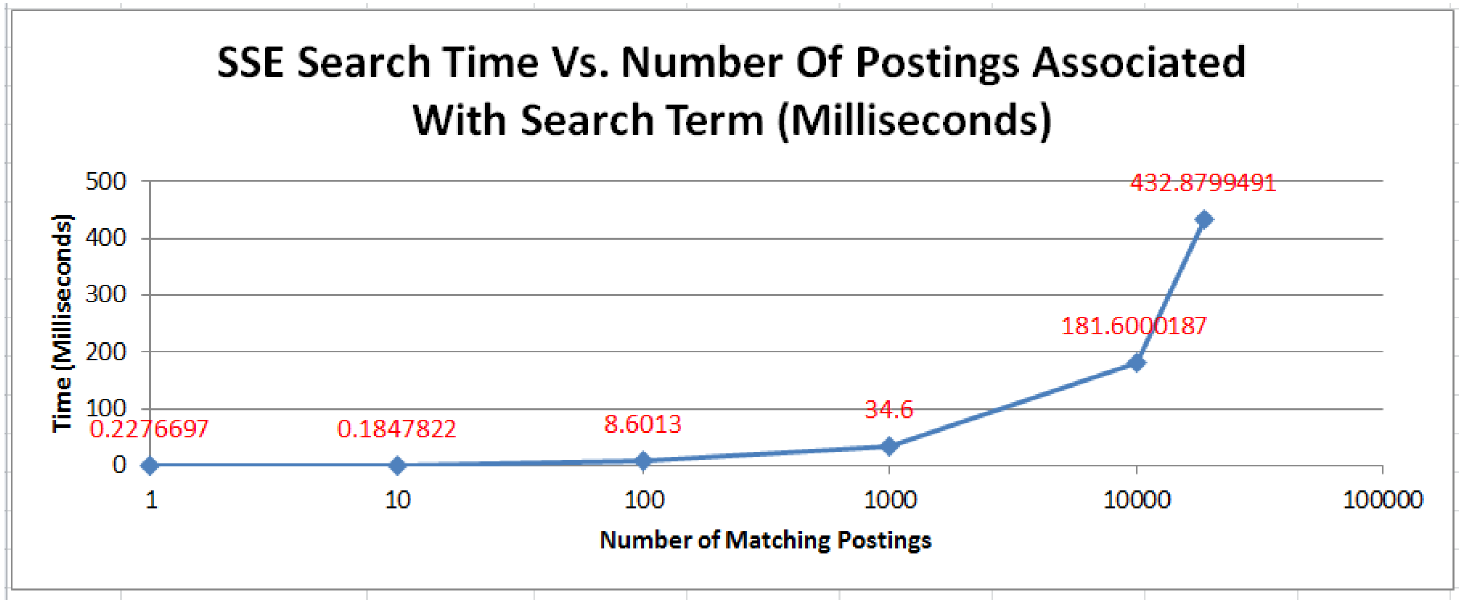


Figure 40: SSE Search Time vs. Number of Matching Postings in SSE Inverted Index.

Figure 40 denotes the Experimental Results associated with searching an SSE Inverted Index and identifying (and decrypting) the Postings associated with the most frequently occurring Lexicon Term within the underlying Document Collection. Figure 40 compares the time taken to search the SSE Inverted Index against the number of Postings associated with the most frequently occurring Lexicon Term within the underlying Document Collection.

In relation to Test Data, the SSE Inverted Index associated with Test Data Set 6 was searched and all Postings associated with the most frequently occurring Lexicon Term (18,828 Postings) were identified in 432 milliseconds.

6.3.2.3 Aggregate Results

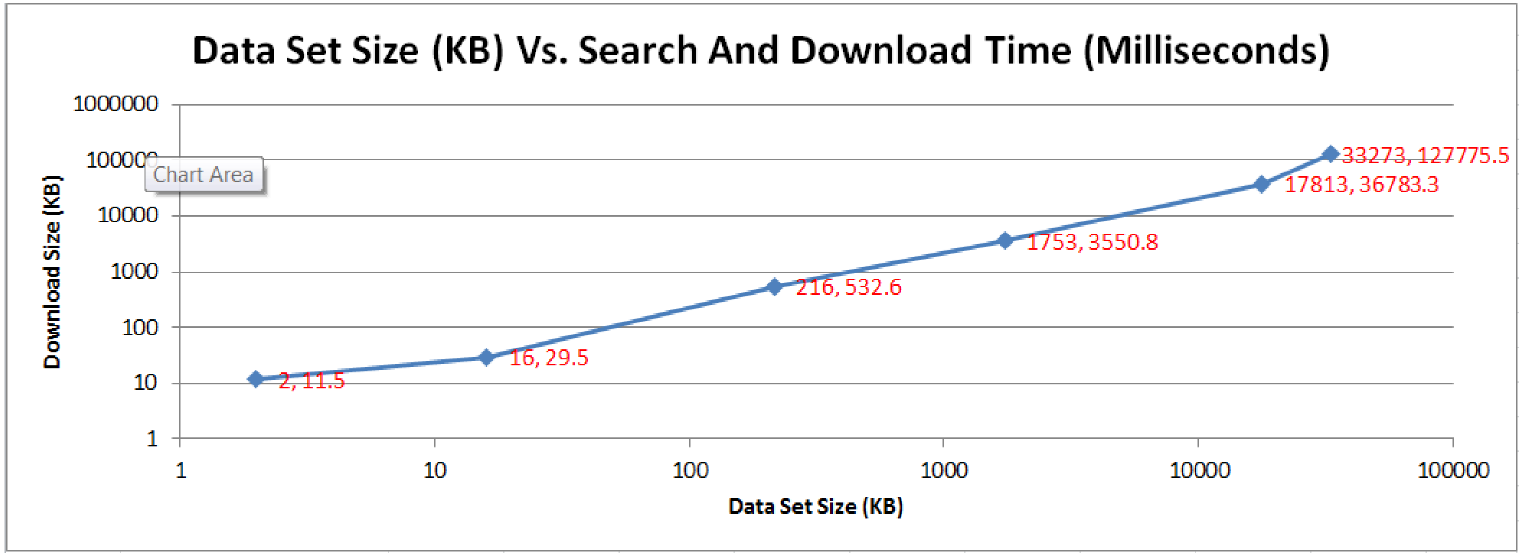


Figure 41: Data Set Size vs. Search and Download Time.

Figure 41 denotes the Experimental Results associated with searching an SSE Inverted Index for the most frequently occurring Lexicon Term within the underlying Document Collection and returning all matching Documents to the Client. Figure 41 compares the time taken to search the SSE Inverted Index and return all matching Documents against the size of the Document Collection returned.

The reader should be aware that the Experimental Results presented in Figure 41 also include the time taken to encapsulate the set of all matching Documents within a ZIP File, which is then returned to the Client.

In relation to Test Data, the set of matching Document associated with the most frequently occurring Lexicon Term contained within Test Data Set 6 was searched and all Documents returned to the Client (32.5 MB) in 2 minutes 7 seconds.

6.3.3 Performance of SSE vs. Plaintext Information Retrieval (IR)

Section 6.3.3.1 denotes the Experimental Results associated with the comparison of plaintext Information Retrieval (IR) uploading and SSE uploading, while Section 6.3.3.2 denotes the Experimental Results associated with the comparison of plaintext IR Inverted Index querying and SSE Inverted Index querying.

6.3.3.1 Plaintext Information Retrieval (IR) Uploading vs. SSE Uploading

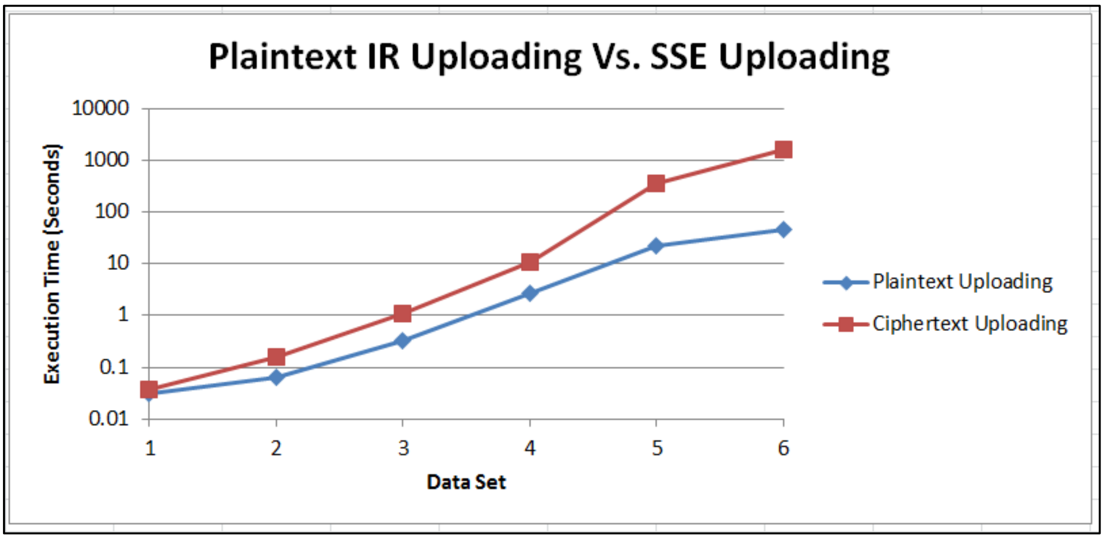


Figure 42: Plaintext IR Uploading vs. SSE Uploading.

Figure 42 denotes the Experimental Results associated with the comparison of traditional plaintext Information Retrieval (IR) uploading and SSE uploading. Those values associated with IR uploading in Figure 42 represent the time taken to upload the Document Collection associated with each Test Data Set from the Client machine to the Server. Those values associated with SSE uploading in Figure 42 represent the time taken to generate the SSE Inverted Index, encrypt the associated Document Collection, and uploading both the Inverted Index and encrypted Document Collection to the Server. From Figure 42, it is immediately obvious that the amount of time necessary for SSE uploading increases in a non-linear manner when compared to the amount of time necessary for plaintext IR uploading.

6.3.3.2 Plaintext Information Retrieval (IR) Querying vs. SSE Querying

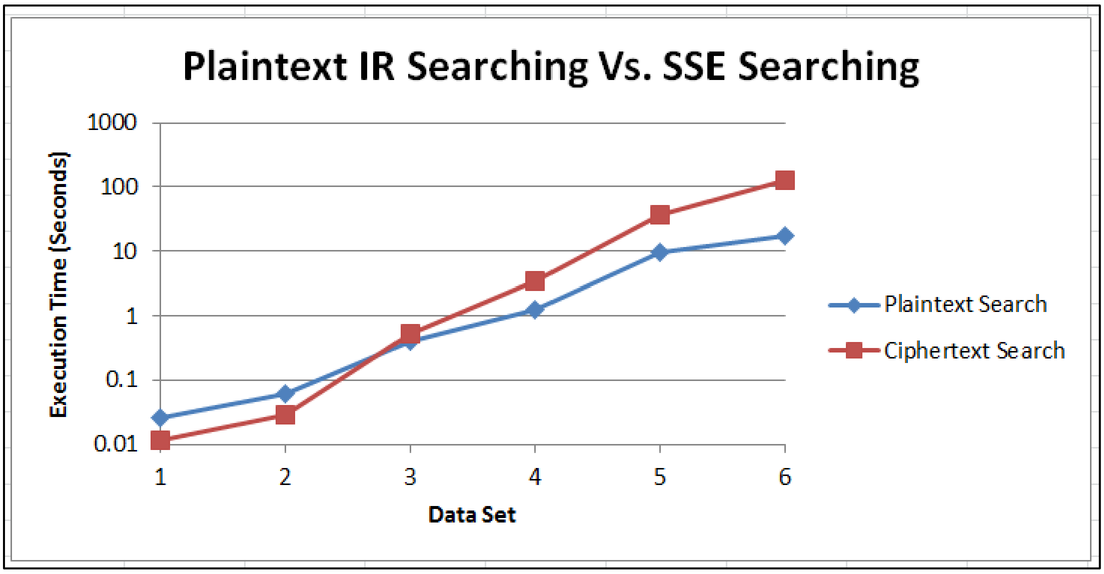


Figure 43: Plaintext IR Querying vs. SSE Querying.

Figure 43 denotes the Experimental Results associated with the comparison of traditional plaintext Information Retrieval (IR) querying and SSE querying.

The Experimental Results presented in Figure 43 consist of the time taken to identify the set of all Postings associated with the most frequently occurring Lexicon Term in the underlying Document Collection, and encapsulating the set of all matching Document within a ZIP File which is then returned to the Client.

As was the case with Figure 42 previously, it is immediately obvious from Figure 43 that the amount of time necessary for SSE querying increases in a non-linear manner when compared to the amount of time necessary for plaintext IR querying.

7. Evaluation

In relation to searching an SSE Inverted Index, the Research Results provide additional proof of the efficiency of SSE when implemented in software. The implementation of SSE developed as part of this dissertation was able to identify and decrypt a single Posting associated with a given Lexicon Term in approximately 22 microseconds (µs). This performance is comparable with the implementations of SSE developed by Kamara *et al.* (2012)[28] and Cash *et al.* (2013) [29] previously.

Regarding the efficiency of constructing an SSE Inverted Index, the Research Results are somewhat inconclusive. Given the five steps involved in constructing an SSE Inverted Index[30], each step in the implementation of SSE produced as part of this dissertation performed as expected with the exception of the second step: *Converting an IR Inverted Index to an SSE Inverted Index*. For Test Data Set 1 (DS1) through Test Data Set 4 (DS4), an SSE Inverted Index was generated from an existing IR Inverted Index in a time linear to the number of Postings stored in the IR Inverted Index; however, for DS5 and DS6, this apparent linear performance decreased dramatically. Despite investigating the problem at length, the author has

[28] 7.3 Microseconds (µs) per Posting.

[29] 100 Microseconds (µs) per Posting.

[30] 1) Generating an IR Inverted Index, 2) Converting IR Inverted Index to SSE Inverted Index, 3) Encrypting Document Collection, 4) Uploading SSE Inverted Index to Server and 5) Uploading Encrypted Document Collection to Server.

been unable to diagnose the exact cause of this performance degradation. The author feels this decrease in performance can be attributed to a combination of one or more of the following: 1) The Java Virtual Machines (JVM) Garbage Collection functionality, 2) Insufficient Java Heap memory, 3) The use of `String` Objects in the `Encrypted_Array_Node` Class, 4) The size of the SSE Inverted Index, and 5) The requirement of the `HashMap iterator()` Method to store an additional copy of the IR Inverted Index `HashMap` on the Java Heap while the SSE Inverted Index is being constructed.

Regarding the first and second point, the author dynamically analysed the 'CipherTXT Storage and Search Engine' application using both the Java Mission Control and Java Flight Recorder applications. In both cases, the author noted that the JVM Garbage Collector was extremely active (eradicating up to 2GB of Objects from the Java Heap on a regular basis) (see Figure 44).

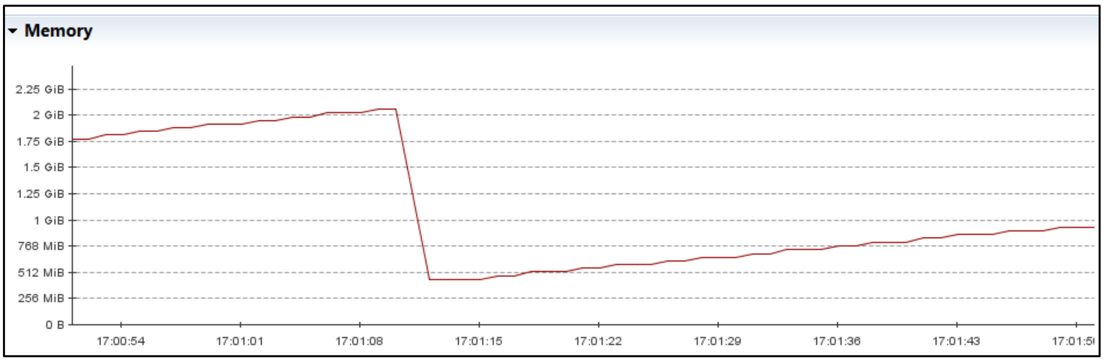


Figure 44: Java Heap Memory Usage and Garbage Collection Statistics for SSE Inverted Index Construction.

Regarding points one, two and three, it is evident that a significant number of Objects are being stored on the Java Heap as the implementation of SSE converts the IR Inverted Index to an SSE Inverted Index. The author feels that one possible explanation for this is the use of String Objects in the Encrypted_Array_Node Class. String Objects are used in the Encrypted_Array_Node Class to store encrypted Document IDs, encrypted Indexes of subsequent Postings, as well as keys required to encrypt/decrypt subsequent Postings. Given that String is a form of Object - and not a primitive data type – all String Objects are therefore stored in the Heap area of the Java Virtual Machines (JVM) memory (Oracle, 2015a).

Regarding points one, two and four, the author noted that the SSE Inverted Index associated with DS5 was 171MB in size, while the SSE Inverted Index associated with DS6 was 325MB in size. When compared to their plaintext equivalent, the DS5 IR Inverted Index is 25MB in size (146MB smaller than its SSE counterpart), while the DS6 IR Inverted Index is 42.8MB (282.2MB smaller than its SSE counterpart). Evidently the SSE Inverted Index associated with both DS5 and DS6 occupy a significant amount of memory. The presence of such large Objects in the Java Heap obviously reduces the amount of space available for subsequent Objects; therefore increasing the frequency of Garbage Collection (Oracle, 2015a).

Regarding points one, two and five, the iterator() method of the HashMap Class may also be a factor in the performance degradation associated with DS5 and DS6. As part of the process of converting the IR Inverted Index to an SSE Inverted Index, the IR Inverted Index must first be loaded into the Java Heap, with each entry in the IR Inverted Index then being examined and subsequently added to the SSE Inverted Index. In order to examine each entry in the IR Inverted Index, the iterator() method must be executed on the HashMap Object underlying the IR Inverted Index (the HashMap Class does not support iteration in any other way). In order to operate, the iterator() method create must first create an exact replica of the IR Inverted Index HashMap on the Heap (that supports iteration); therefore doubling the amount of Heap space associated with the IR Inverted Index (Oracle, 2015c). As indicated previously, the IR Inverted Index associated with DS5 is 25MB in size (increasing to 50MB during SSE Inverted Index Construction as a result of using the iterator() method), while the IR Inverted Index associated with DS6 is 42.8MB (increasing to 85.6MB during SSE Inverted Index Construction as a result of using the iterator() method). As mentioned previously, the presence of such large Objects on the Java Heap reduces the amount of space available for additional Objects and also has the effect of increasing the frequency of Garbage Collection (Oracle, 2015a).

The author has identified two other potential causes of the performance degradation associated with DS5 and DS6: 1) Hash Collisions Occurring As A Result Of Inserting Keys Into The SSE Inverted Index HashMap Object, and 2) The Natural Performance Degradation Associated With An Ever Expanding HashMap Object; however the authors Research appears to have ruled both potential causes out.

Regarding Hash Collisions in a HashMap, the author noted that the location of an Object within a HashMap is determined by the value resulting from executing the hashCode() method associated with the Object being inserted into the HashMap. In the event that two Objects produce the same hashCode() value, the `HashMap` Class must then execute the `compare()` method associated with both Objects to determine whether or not both Objects are in fact equivalent to each other (Oracle, 2015c). Personally, the author does not feel that Hash Collisions are an issue in theis implementation of SSE as the `hashCode()` method associated with the `String` Class produces a 32 bit hash value (approximately 4.3 billion different Hash Values) (Oracle, 2015e); therefore making Hash Collisions highly unlikely for data sets the size of DS5 and DS6.

Regarding the natural performance degradation associated with an ever expanding `HashMap`, the author has noted that a Java `HashMap` Object must be created with a specified initial capacity; *that is, number of expected entries*, and a specified expected load; *that is, the percentage of the initial capacity that must be used before the capacity of the `HashMap` is increased*. In the event that that the load specified for the `HashMap` is exceeded, a new `HashMap` Object must then be constructed (this is done automatically by the `HashMap` Class). The process of constructing a new `HashMap` Object requires that each entry in the existing `HashMap` Object be retrieved, re-hashed, and inserted into the new – larger – `HashMap` Object (Oracle, 2015c). Personally, the author does not feel this is an issue that affects this implementation of SSE as the author has taken the initial capacity and load factor of the SSE `HashMap` into consideration and constructed the `HashMap` in a manner that does not require the `HashMap` to be expanded.

Regarding Research Results relating to upload speeds and download speeds, the reader should be aware that a localhost web server was used during Testing; as such, the time associated with uploading and downloading data may appear significantly faster than those which are achievable using a live system.

8. Conclusions and Further Research

Section 8.1 presents the Conclusions derived from the authors work on this dissertation, while Section 8.2 discusses potential further Research.

8.1 Conclusions

Given the similarity between Searchable Symmetric Encryption (SSE) and plaintext Information Retrieval (IR), it is inevitable that comparisons will be made between the two. While having a number of goals and functions in common, the fact remains that the primary goal of SSE is to provide Data and Query Privacy. Given this – as well as the fact that SSE operates in a manner that differs greatly from plaintext IR - the author is of the option that SSE should be viewed as a separate paradigm in the context of Information Retrieval, and not an extension of plaintext IR.

In order to provide Data and Query Privacy, SSE requires a significant amount of additional processing time to carry out a task when compared to the processing time associated with carrying out the same task using plaintext IR. In terms of the performance overhead of using SSE, the Research Results show that little or no correlation exists between the time associated with carrying out a task using plaintext IR, and carrying out the same task using SSE (see Section 6.3.3). In general, the Research Results have shown that the time taken to carry out a task using SSE is greater than the time taken to carry out the same task using plaintext

IR; nonetheless, this is to be expected given that the process of uploading a Document Collection to the Server using SSE requires the Client to first generate an SSE Inverted Index, encrypt the underlying Document Collection and then upload both to the Server, as well as the need for the Sever to decrypt Postings as part of SSE querying.

The Research Results show that carrying out a task using SSE is directly proportional to the amount of information involved. In the case of constructing an IR Inverted Index, the Research Results show that the time taken to generate an IR Inverted Index is directly proportional to the number of Terms contained in the underlying Document Collection (see Section 6.3.1.1). Converting the same IR Inverted Index to an SSE Inverted Index is directly proportional to the number of Postings contained within the IR Inverted Index[31] (see Section 6.3.1.2), while the time taken to encrypt the underlying Document Collection is directly proportional to the number of Terms contained within the Document Collection (see Section 6.3.1.3). In relation to searching in SSE, the time taken to identify and decrypt the set of Postings associated with a given Lexicon Term is directly proportional to the number of Postings (see Section 6.3.2.2).

[31] With the exception of Test Data Set 5 and Test Data Set 6 – see Section 7 and Section 8.2 for further explanation.

Regarding the question of whether or not SSE is efficient enough to be deployed in a Cloud environment, the author if of the opinion that the answer to this question is context dependant.

If deployed in an environment whereby Search Results only have to be returned to the user in small quantities (such as an Internet Search Engine (*For Example: ten results at a time*)), then SSE would be more than efficient, irrespective of the size of the underlying Data Set (due to the fact that only a small number of Postings would need to be decrypted at a given time).

If deployed in an environment whereby all Search Results must be returned at once (as was the case with the implementation of SSE developed as part of this dissertation, as well as the implementations developed by Kamara *et al.* (2012) and Cash *et al.* (2013), the author is of the opinion that SSE would only be suitable for small and medium sized Data Sets. When applied to large Data Sets, SSE querying can become inefficient as its search time is directly proportional to the number of matching Postings (which is likely to be significant for large Data Sets).

Regarding the possible commercialisation of SSE, the success of such a product would undoubtedly hinge on the knowledge of those people using the product. Users of such a product would need to be aware that SSE provides Data/Query Privacy in exchange for the efficiency associated with plaintext IR, and that an SSE Inverted Index – while slow to construct for large Data Sets – is designed to achieve efficient search speeds whilst maintaining Data Privacy.

8.2 Further Research

Regarding the Literature Review carried out as part of this dissertation, the author would like to acknowledge that the description of the Inverted Index provided in Section 2.2 constitutes the Inverted Index in its most basic form; *that is, denoting whether or not a Term is contained within a Document Collection, as well as support for single Term Queries only*; nonetheless, the description provided is sufficient enough to cover the usage of the Inverted Index in SSE. Manning *et al.* (2008) provides a comprehensive overview of the Inverted Index as it is used in plaintext IR, including numerous extensions to the description provided in this dissertation. Furthermore, the author would also like to acknowledge that the topic of Update Leakage; *that is, Information Leakage resulting from changes being made to the SSE Inverted Index and the underlying Document Collection*, was not covered in the Literature Review as it was deemed beyond the scope of this dissertation. Kamara *et al.* (2012) and Van Liesdonk *et al.* (2010) give a comprehensive overview of the topic.

Regarding improvements to the Implementation of SSE developed as part of this dissertation, the author would like to diagnose and rectify the performance issues encountered when generating an SSE Inverted Index for large Data Sets. In addition, the author would also like to apply secure coding practices to the implementation of SSE developed as part of this dissertation.

In relation to the aforementioned performance issue, the author feels the performance degradation associated with generating an SSE Inverted Index for large Data Sets is due to the use of `String` Objects for storing ciphertext.

All built-in cryptographic methods of the Java library output ciphertext in `byte[]` array form initially (Oracle, 2015b; Oracle, 2015d). In the `Crypto_Methods` Class utilised as part of the authors implementation of SSE, each `byte[]` array produced as part of a cryptographic operation is converted to a Base64 encoded `String` Object before being returned to the calling component. The decision to use `String` Objects to store ciphertext – instead of `byte[]` arrays - was taken by the author with a view to simplifying the process of debugging cryptographic operations.

In hindsight, the author now realises that each item of ciphertext produced by the `Crypto_Methods` Class exists in two forms on the Java Heap: In `byte[]` array form, and `String` form. Evidently this represents a significant amount of Heap space wastage. For this reason alone, the author feels it may be a worthwhile exercise re-developing the implementation of SSE using `byte[]` arrays instead of `String` Objects with a view to improving performance when dealing with larger Data Sets; *that is, DS5, DS6*.

In addition to the use of `String` Objects, another potential cause of the aforementioned performance degradation may be the applications reliance on automated memory management. At no point during the implementation of the 'CipherTXT Storage and Search Engine' application was manual memory management utilised; *that is, manual de-allocation of memory*. All memory management associated with the application is managed by the Java Virtual Machine and automated calls to the Java Garbage Collector. Given this, the author feels it may be a worthwhile exercise re-engineering the application with manual memory management in mind, with a view to improving the performance of the application when generating an SSE Inverted Index for large Data Sets.

In relation to secure coding, the author readily admits that no secure coding principles were applied during the development of both the 'PlainTXT Storage and Search Engine' and 'CipherTXT Storage and Search Engine' applications. Given that 'CipherTXT Storage and Search Engine' is a data security application, the author would like to apply secure coding principles to the application at some point in the future.

Regarding extending the functionality of the Implementation of SSE developed as part of this dissertation, the author would like to investigate the feasibility of incorporating support for multiple term queries (as demonstrated by Cash *et al.* (2013)), as well as including support for adding and deleting Documents from the underlying Document Collection (and updating the SSE Inverted Index appropriately – on the Server side - as demonstrated by Kamara *et al.* (2012) and Van Liesdonk *et al.* (2010).

Furthermore, given the size of the Inverted Indexes produced by the implementation of SSE produced as part of this dissertation, the author believes that the topic of Inverted Index compression may also be worth researching (Luenberger, 2006, p.290).

9. References

Arkin, B. (2013) *IMPORTANT CUSTOMER SECURITY ANNOUNCEMENT* [online], available: http://blogs.adobe.com/conversations/2013/10/important-customer-security-announcement.html [accessed 13/09/2015].

Boneh, D., Crescenzom, G. D., Ostrovsky, R. and Rersiano, G. (2004) 'Public Key Encryption With Keyword Search', in Cachin, C. and Camenisch, J. L., eds., *International Conference on the Theory and Applications of Cryptographic Techniques*, Interlaken, Switzerland, Berlin/Heidelberg: Springer, 506-522.

Bosch, C., Hartel, P., Jonker, W. and Peter, A. (2014) *A Survey of Provably Secure Searchable Encryption* [online], available: http://eprints.eemcs.utwente.nl/24788/01/a18-bosch.pdf [accessed 18/04/2015].

Bosch, C., Tang, Q., Hartel, P. and Jonker, W. (2012) *Selective Document Retrieval from Encrypted Database* [online], available: http://core.ac.uk/download/pdf/11483856.pdf [accessed 13/09/2015].

Cash, D., Jarecki, S., Jutla, C., Krawczyk, H., Rosu, M. C. and Steiner, M. (2013) *Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries* [online], available: https://eprint.iacr.org/2013/169.pdf [accessed 13/09/2015].

Chang, Y.-C. and Mitzenmacher, M. (2005) *Privacy Preserving Keyword Searches on Remote Encrypted Data* [online], available: http://www.eecs.harvard.edu/~michaelm/postscripts/acns2005.pdf [accessed 13/04/2015].

Chase, M. and Kamara, S. (2010) *Structured Encryption and Controlled Disclosure* [online], available: http://eprint.iacr.org/2011/010.pdf [accessed 17/03/2015].

Chunsheng, G. (2011) *New Fully Homomorphic Encryption over the Integers* [online], available: https://eprint.iacr.org/2011/118.pdf [accessed 13/09/2015].

Claycomb, W. R. and Nicoll, A. (2012) *Insider Threats to Cloud Computing: Directions for New Research Challenges* [online], available: https://resources.sei.cmu.edu/asset_files/WhitePaper/2012_019_001_52385.pdf [accessed 14/09/2015].

Columbus, L. (2015) *Roundup Of Cloud Computing Forecasts And Market Estimates, 2015* [online], available: http://www.forbes.com/sites/louiscolumbus/2015/01/24/roundup-of-cloud-computing-forecasts-and-market-estimates-2015/ [accessed 13/09/2015].

Curtmola, R., Garay, J., Kamara, S. and Ostrovsky, R. (2006) *Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions* [online], available: http://eprint.iacr.org/2006/210.pdf [accessed 12/02/2015].

Dictionary.com (2015a) *Dictionary* [online], available: http://dictionary.reference.com/browse/Dictionary?s=t [accessed 02/03/2015].

Dictionary.com (2015b) *Word* [online], available: http://dictionary.reference.com/browse/word [accessed 03/02/2015].

Eurostat (2014) *Cloud computing - statistics on the use by enterprises* [online], available: http://ec.europa.eu/eurostat/statistics-explained/index.php/Cloud_computing_-_statistics_on_the_use_by_enterprises [accessed 13/09/2015].

Gentry, C. (2009) *A Fully Homomorphic Encryption Scheme*, unpublished thesis (PhD), Stanford University.

Gentry, C., Halvei, S. and Smart, N. P. (2015) *Homomorphic Evaluation of the AES Circuit (Updated Implementation)* [online], available: https://eprint.iacr.org/2012/099.pdf [accessed 04/05/2015].

Goh, E. (2003) *Secure Indexes* [online], available: http://crypto.stanford.edu/~eujin/papers/secureindex/secureindex.pdf [accessed 12/05/2015].

Goldreich, O. and Ostrovsky, R. (1992) *Software Protection and Simulation on Oblivious RAMs* [online], available: [accessed 08/05/2015].

Hahn, F. and Kerschbaum, F. (2014) *Searchable Encryption with Secure and Efficient Updates* [online], available: http://fkerschbaum.org/ccs14b.pdf [accessed 13/09/2015].

Hashizume, K., Rosado, D. G., Fernández-Medina, E. and Fernandez, E. B. (2013) *An analysis of security issues for cloud computing* [online], available:

http://www.jisajournal.com/content/pdf/1869-0238-4-5.pdf [accessed 14/09/2015].

ICO (2014) *Monetary Penalty Notice: Ministry of Justice* [online], available: https://ico.org.uk/media/action-weve-taken/mpns/2656/ministry-of-justice-monetary-penalty-notice-26082014.pdf [accessed 14/09/2015].

ICO (2015) *Monetary Penalty Notice: Staysure.co.uk Limited* [online], available: https://ico.org.uk/media/action-weve-taken/mpns/1043368/staysure-monetary-penalty-notice.pdf [accessed 14/09/2015].

Intermedia.net (2014) *The Ex-Employee Menace* [online], available: https://www.intermedia.net/Reports/RogueAccess [accessed 14/09/2015].

Kamara, S. (2013) *How To Search On Encrypted Data* [online], available: http://research.microsoft.com/en-us/um/people/senyk/slides/encryptedsearch-full.pdf [accessed 25/04/2015].

Kamara, S., Papamanthou, C. and Roeder, T. (2012) *Dynamic Searchable Symmetric Encryption* [online], available: https://eprint.iacr.org/2012/530.pdf [accessed 04/05/2015].

Kerris, N. and Muller, T. (2014) *Apple Media Advisory - Update to Celebrity Photo Investigation* [online], available: http://www.apple.com/pr/library/2014/09/02Apple-Media-Advisory.html [accessed 13/09/2014].

Levick.com (2015) *DATA SECURITY & PRIVACY* [online], available: http://levick.com/experience/specialty/data-security-privacy [accessed 14/09/2015].

Luenberger, D. G. (2006) 'Information Science' in, Princeton, New Jersey: Princeton University Press, 284-300.

Manning, C. D., Raghavan, P. and , S., H. (2008) *Introduction to Information Retrieval*, Cambridge, England: Cambridge University Press.

Mather, T., Kumaraswamy, S. and Latif, S. (2009) *Cloud Security and Privacy*, California: O'Reilly.

Nguyen, M., Chau, N., Jung, S. and Jung, S. (2014) *A Demonstration of Malicious Insider Attacks inside*

Cloud IaaS Vendor [online], available: http://www.ijiet.org/papers/455-F028.pdf [accessed 14/09/2015].

Oracle (2015a) *Chapter 2. The Structure of the Java Virtual Machine* [online], available: https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html [accessed 09/08/2015].

Oracle (2015b) *Cipher* [online], available: http://docs.oracle.com/javase/8/docs/api/javax/crypto/Cipher.html [accessed 09/08/2015].

Oracle (2015c) *HashMap* [online], available: https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html [accessed 09/08/2015].

Oracle (2015d) *Mac* [online], available: http://docs.oracle.com/javase/8/docs/api/javax/crypto/Mac.html [accessed 09/08/2015].

Oracle (2015e) *String* [online], available: http://docs.oracle.com/javase/8/docs/api/java/lang/String.html [accessed 09/08/2015].

OWASP (2013) *Top 10 2013-A6-Sensitive Data Exposure* [online], available: https://www.owasp.org/index.php/Top_10_2013-A6-Sensitive_Data_Exposure [accessed 14/09/2015].

Rennie, J. (2008) *The 20 Newsgroups Data Set* [online], available: http://qwone.com/~jason/20Newsgroups/ [accessed 03/08/2015].

Shen, E., Shi, E. and Waters, B. (2008) *Predicate Privacy in Encryption Systems* [online], available: https://eprint.iacr.org/2008/536.pdf [accessed 13/09/2015].

Song, D. X., Wagner, D. and Perrig, A. (2000) 'Practical Techniques For Searches On Encrypted Data', in Titsworth, F. M., ed., *IEEE Symposium on Security and Privacy, 2000*, Berkeley, California, 14-17 May 2000, Washington, D.C.: IEEE Computer Society, 44-55.

Sony (2014) *Message for current and former Sony Pictures employees and dependants, and for production employees* [online], available: http://www.sonypictures.com/corp/notification/SPE_Cyber_Notification.pdf?#zoom=100 [accessed 13/09/2014].

Stallings, W. (2014) *Cryptography and Network Security: Principles And Practices*, New Jersey: Pearson Education.

Stefanov, E., Papamanthou, C. and Shi, E. (2013) *Practical Dynamic Searchable Encryption*

with Small Leakage [online], available: https://eprint.iacr.org/2013/832.pdf [accessed 13/09/2015].

Van Liesdonk, P., Sedghi, S., Doumen, J., Hartel, P. and Jonker, J. (2010) 'Computationally Efficient Searchable Symmetric Encryption', in Jonker, W. and Petković, M., eds., *Proceedings of the 7th VLDB conference on Secure data management*, Singapore, Springer, 87-100.

Zhang, Y., Reiter, M. K., Juels, A. and Ristenpart, T. (2012) *Cross-VM Side Channels and Their Use to Extract*

Private Keys [online], available: http://www.cs.unc.edu/~reiter/papers/2012/CCS.pdf [accessed 13/09/2015].