

# A secure searcher for end-to-end encrypted email communication

---

Balamaruthu Mani

Dissertation 2015

Erasmus Mundus MSc in Dependable Software Systems



Department of Computer Science

National University of Ireland, Maynooth

Co. Kildare, Ireland

A dissertation submitted in partial fulfilment

of the requirements for the

Erasmus Mundus MSc Dependable Software Systems

Head of Department: Dr Adam Winstanley

Supervisor: Professor Barak A. Pearlmutter

8-June-2015

Word Count: 19469

## Abstract

Email has become a common mode of communication for confidential personal as well as business needs. There are different approaches to authenticate the sender of an email message at the receiver's client and ensure that the message can be read only by the intended recipient. A typical approach is to use an email encryption standard to encrypt the message on the sender's client and decrypt it on the receiver's client for secure communication. A major drawback of this approach is that only the encrypted email messages are stored in the mail servers and the default search does not work on encrypted data. This project details an approach that could be adopted for securely searching email messages protected using end-to-end encrypted email communication.

This project proposes an overall design for securely searching encrypted email messages and provides an implementation in Java based on a cryptographically secure Bloom filter technique to create a secure index. The implemented library is then integrated with an open source email client to depict its usability in a live environment. The technique and the implemented library are further evaluated for security and scalability while allowing remote storage of the created secure index. The research in this project would enhance email clients that support encrypted email transfer with a full secure search functionality.

## Categories

H.3 Information Storage and Retrieval

H.3.3 Information Search and Retrieval

E. Data

E.3 Data Encryption

## General Terms

Design, Documentation, Performance, Security

## Keywords

Email encryption, Secure Index, Bloom filter, Secure Search, Searching Encrypted Email, Email client

## Declaration

The main text of this project report is approximately 19,469 words long, excluding the table of contents and appendices. The work was performed during the current academic year under the supervision of Professor Barak A. Pearlmutter.

I hereby certify that this material, which I now submit for assessment on the program of study leading to the award of Master of Science in Dependable Software Systems, is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: \_\_\_\_\_  
Balamaruthu Mani

## Contents

Abstract.....	2
Categories .....	2
General Terms.....	2
Keywords .....	2
Declaration.....	3
1. Introduction.....	8
1.1 Overview.....	8
1.2 Motivation.....	8
1.3 Objectives .....	9
1.3.1 Primary objectives.....	9
1.3.2 Secondary objectives.....	9
1.3.3 Tertiary objectives.....	9
1.4 Project outcome .....	9
1.5 Dissertation Organisation.....	10
2 Context Survey.....	11
2.1 Background.....	11
2.1.1 Encryption basic techniques.....	11
2.1.2 End-to-End Encryption Techniques .....	11
2.2 Related Work .....	12
2.2.1 End-to-End Email Encryption Tools.....	13
2.2.2 Search Techniques on Encrypted data .....	17
2.2.3 Email Client Integration.....	20
3 Strategy .....	23
3.1 Development strategy .....	23
3.1.1 Technique selection.....	23
3.1.2 Implementation strategy.....	23
3.2 Testing strategy .....	24
3.2.1 JUnit and Scenario based testing.....	24
3.2.2 Performance .....	24
4 Design.....	25
4.1 Solution Design.....	25
4.1.1 Core Technique.....	25

4.1.2	Design Notation .....	26
4.1.3	Logical View.....	26
4.1.4	Deployment View .....	27
4.2	Library Design .....	32
4.2.1	Logical View.....	32
4.2.2	Runtime View .....	35
4.3	Integration Design.....	36
5	Implementation .....	38
5.1	Core Library Implementation.....	38
5.1.1	Indexer .....	38
5.1.2	Key Manager.....	40
5.1.3	Encryption Utilities.....	40
5.1.4	Searcher.....	40
5.1.5	Data Manager.....	40
5.2	Integration Implementation.....	41
5.2.1	Columba Client Integration.....	41
5.2.2	Maildir Integration .....	42
5.3	Encrypted Email Generator.....	43
6	Secure Searcher.....	44
6.1	Implementation Library .....	44
6.1.1	Indexer .....	44
6.1.2	Searcher.....	44
6.1.3	Key Manager.....	45
6.1.4	Decrypter.....	45
6.2	Columba Search Functionality.....	46
6.2.1	Indexing .....	46
6.2.2	Searching.....	47
6.2.3	Text Viewer .....	48
6.2.4	Key store Configuration.....	49
7	Evaluation .....	50
7.1	Objectives .....	50
7.2	Development Strategy.....	50
7.3	Testing Strategy .....	51

7.4	Functionality .....	51
7.4.1	Implementation Library .....	51
7.4.2	Integrated Email Client .....	52
7.4.3	Critical Functionality Analysis .....	52
7.5	Performance .....	55
7.5.1	Analysis Environment Setup.....	55
7.5.2	Indexing Analysis .....	55
7.5.3	Searching Analysis.....	56
7.6	Security .....	59
7.6.1	Security model of the Final implementation.....	59
7.6.2	Enhanced Security Model .....	60
7.6.3	Vulnerability Model.....	60
7.7	Summary.....	60
8	Conclusions and Future Work.....	62
8.1	Conclusions.....	62
8.2	Summary of Challenges .....	62
8.3	Future Works and Improvements.....	63
A	Testing Summary .....	64
B	Changes to Original Specification.....	66
C	Project Plan .....	67
D	User Manual.....	68
D.1	Columba Client Configuration.....	68
D.1.1	Email Account Configuration .....	68
D.1.2	Enable IMAP on Gmail.....	70
D.2	Secure Searcher Configuration .....	71
D.2.1	Key store Configuration.....	71
D.2.2	Indexing .....	72
D.2.3	Searching.....	73
D.3	Demo Account Configuration .....	73
D.3.1	Demo Account Configuration Details.....	73
	References.....	75

## List of Figures

Figure 2-1 OpenPGP decrypted message using MailPile .....	14
Figure 2-2 Mailpile search on encrypted email .....	14
Figure 2-3 Mailvelope mail compose editor with Gmail .....	15
Figure 2-4 OpenPGP encrypted message using Mailvelope .....	15
Figure 2-5 Thunderbird S/MIME Encryption with Certificate Manager .....	16
Figure 2-6 Encryption using Enigmail with Key Management .....	17
Figure 2-7 OpenPGP decryption using Enigmail .....	17
Figure 2-8 Pooka Email Client.....	22
Figure 2-9 Columba Email Client.....	22
Figure 4-1 Secure Searcher Logical Design .....	26
Figure 4-2 Secure Searcher Deployment Design .....	28
Figure 4-3 Secure Searcher Remote Server Alternative Deployment Design .....	29
Figure 4-4 Secure Searcher Client Server Alternative Deployment Design .....	29
Figure 4-5 Secure Searcher Remote Storage Alternative Deployment Design .....	31
Figure 4-6 Secure Searcher Logical View .....	32
Figure 4-7 Decrypter Logical Design - Chain of Responsibility .....	33
Figure 4-8 Indexing Runtime View .....	35
Figure 4-9 Searching Runtime View .....	36
Figure 4-10 Indexing Integration Design.....	36
Figure 4-11 Searching Integration Design .....	37
Figure 6-1 Columba Indexing Dialog .....	46
Figure 6-2 Gmail S/MIME message .....	47
Figure 6-3 Columba S/MIME Message .....	47
Figure 6-4 Gmail OpenPGP Message .....	48
Figure 6-5 Columba OpenPGP Message .....	48
Figure 6-6 Columba KeyStore Configuration.....	49
Figure 7-1 JUnit Test Coverage Report .....	51
Figure 7-2 Messages Indexing Time vs Number of Messages .....	56
Figure 7-3 Index Records Search time with Message Id Encryption.....	57
Figure 7-4 Final Index Records Search Time vs Number of Index Records .....	58
Figure 7-5 Index Record Search Time vs Record Length.....	58
Figure A-1 JUnit Test Execution Report .....	64
Figure D-1 Columba New Account Wizard .....	68
Figure D-2 Columba Server Properties Wizard.....	69
Figure D-3 Columba Account Preferences Dialog .....	69
Figure D-4 Columba IMAP Connection Settings .....	70
Figure D-5 Gmail Enable IMAP.....	70
Figure D-6 Gmail Less Secure Access .....	71
Figure D-7 Columba KeyStore Configuration.....	71
Figure D-8 Enter Import Key and Secure Search Password Dialogs.....	72
Figure D-9 Columba Index All Encrypted Messages Dialog .....	72
Figure D-10 Columba Encrypted Message Search .....	73

# 1. Introduction

## 1.1 Overview

Email (Electronic mail) has become a common mode of communication for confidential personal as well as business needs. Email messages are composed and sent using a local email client which could be a browser based thin client software like Gmail web interface [Google 2015] or thick clients like Thunderbird [Mozilla 2015] or Outlook [Microsoft 2015]. The composed email messages are sent to the mail servers which facilitate the transfer of messages to the corresponding recipients. The messages are typically encrypted to and from mail servers using a transport layer security protocol (TLS) [Dierks & Rescorla 2008]. But the mail server has access to the un-encrypted plain text form of the message contents.

Even though mail service providers may protect their data servers, recent leaks [Guardian News and Media Limited 2015] regarding the surveillance of data servers by government agencies and the number of possible attacks [Nanavati et al. 2014] on the data handled by the servers emphasises a lack of security and privacy of the user's data stored on an un-trusted cloud storage. This has motivated the use of encrypted email services, in particular when communicating confidential information.

A common approach for sending an encrypted message through email is to send encrypted attachments [Adobe Systems Incorporated. 2015; WinZip Computing 2013] as a password protected ZIP or PDF files. This method is commonly used in sending e-financial statements by banks or income tax statements by tax departments. This technique is used due to a lack of sophisticated support for sending encrypted email message, instead of an encrypted attachment. And it provides a reasonable guarantee that only the intended recipient could view the attachment.

There are many encryption protocols that specify a mechanism to send and receive encrypted emails, the most commonly used encryption protocols include OpenPGP [Torto et al. 2015] and S/MIME [Ramsdell & Turner 2010]. Some mail servers [Hush Communications Canada Inc. 2015; Trancecrypt Inc. 2014] that support secure mail transfer have the encryptions performed at the mail server, so the mail server and the storage needs to be completely trusted. A security that the data could be read only by the intended recipient at the receiver side is possible only with the help of end-to-end encrypted email transfer.

End-to-End encryption requires keys used to encrypt the messages be managed at the client sides i.e., sender and the receiver. Sender encrypts the message using a public key and this encrypted message could be decrypted only by the receiver's private key. With end-to-end encryption, only the encrypted messages can be seen by the mail servers and thus the default search service provided by the mail servers is not usable.

## 1.2 Motivation

Mailvelope [Oberndörfer 2014], GPGTools [GPGTools 2015] and Enigmail [Brunschwig 2015] are some of the tools that offers functionalities for sending and receiving end-to-end encrypted email messages. Most of the tools are plug-ins or extensions to existing email clients. Moreover, ProtonMail [Yen et al. 2015], a mail service provider, offers mail services promising a full pledged end-to-end encrypted service. But the common problem with these tools is that once the



data is encrypted, the encrypted email messages are stored in the mail provider storage and the default search for email does not work on these encrypted messages. This project would enable end-to-end encrypted email service providers to implement a full-pledged service with a secure search capability.

Searching on encrypted data is discussed in some cryptography papers including [Xiaodong et al. 2000; Shmueli et al. n.d.; Goh 2004; Thian et al. 2005] . But the feasibility of an implementation of the techniques proposed by these research papers using the existing encryption libraries as well as the suitability of the technique with email communication is not clear. This project aims to identify an approach for which an implementation is possible with the existing cryptographic libraries. Furthermore, development of a tool using the approach would provide evidence of its suitability for searching encrypted email messages.

A naive approach would be to decrypt all the encrypted messages and store them in a local storage to support searching. But storing the decrypted data is inefficient with respect to storage and portability, as a copy of the email is accessible only on the machine where decrypted emails are stored. In addition, it increases the security risk if the storage is compromised. This project aims to store a secure index of the encrypted emails, promising security and space efficiency. Moreover this project evaluates the feasibility of storing the index on a remote server which supports network search from any machine.

## 1.3 Objectives

The main goal of the project is to identify a mechanism to search encrypted email messages that complements an end-to-end encrypted email transfer between a sender and receiver. And this project focuses on the commonly used protocols for email encryption: OpenPGP and S/MIME. The security and feasibility of such an approach should be evaluated with the help of a tool that supports searching. Moreover, the scalability and usability of the tool needs to be evaluated.

### 1.3.1 Primary objectives

- Identify an approach for securely searching encrypted email messages.
- Develop a tool that supports creating a secure index of the email messages encrypted using the OpenPGP encryption protocol.
- Add a search capability to the tool resulting in pointers to the indexed email messages.

### 1.3.2 Secondary objectives

- Enhance the tool to support indexing of email messages encrypted using S/MIME protocol.
- Extend the tool to support incremental indexing for incorporating new email messages.

### 1.3.3 Tertiary objectives

- Integrate the library with an open source Java email client to demonstrate the functionality in a live environment.

## 1.4 Project outcome

The project's outcomes are a Java library that could be integrated into an existing email client for securely searching email and an email client with a beta support reflecting usability and security.

Moreover, it proposes architecture for adding secure search functionality in the current email communication model.

The implementation satisfies the primary and secondary objectives with capabilities to index and search encrypted email messages using S/MIME and OpenPGP protocols. The implementation makes use of the existing Bouncy Castle [Bouncy Castle Inc 2013] cryptographic libraries and adopts a layered design supporting future extensions. This project recommends an integration model for securely searching email using the implemented library and analyses alternative designs for security. Finally the security and scalability of the technique and implementation is evaluated.

## 1.5 Dissertation Organisation

This dissertation document organisation is outlined as follows.

**Context Survey** section provides a background of end-to-end email encryption standards and describes the related tools and techniques that influence this project.

**Strategy** section presents the design, development and evaluation strategies adopted for this project.

**Design** section provides an overview of high level design of the overall solution. It is followed by different architectural views of the implemented library and the proposed integration design of the library with the existing email clients.

**Implementation** section details the implementation strategy, key decisions made and challenges encountered during the implementation of the library.

**Secure Searcher** section provides an overview of the functionality provided by the implemented library.

**Evaluation** section provides a critical analysis of the technique and implementation with respect to the security, adopted strategy and scalability.

**Conclusions and Future Work** section summarises the achieved goals, encountered challenges and possible future works on the project.

This document concludes with a series of appendices describing **A Testing Summary**, **B Changes to Original Specification**, **C Project Plan** and **D User Manual**.

## 2 Context Survey

This chapter includes the details and influence of the related works that were considered for the development of the tool. This chapter begins with a background section including an overview of existing end-to-end encryption techniques followed by the related works section providing an overview of end-to-end encryption tools, indexing and search techniques.

### 2.1 Background

This section outlines the background of basic encryption and end-to-end encryption techniques.

#### 2.1.1 Encryption basic techniques

There are two classes of basic encryption techniques. One is referred to as symmetric encryption where the data is encrypted using a shared secret key or password phrase and the data could be decrypted only using the same secret key used for encryption. Another class of encryption refers to the public key encryption, where two keys (private and public key) are maintained. The private key is confidential and known only to the owner where as the public key could be announced liberally. The messages encrypted using one key could be decrypted only by another key of the key pair respectively.

#### 2.1.2 End-to-End Encryption Techniques

The end-to-end encryption refers to a class of techniques where the encryption and decryption are performed at the end user clients without involvement of a centralised encryption server. The techniques that are considered popular and widely used/implemented for email communication [Internet Mail Consortium 2015] are listed as follows.

##### 2.1.2.1 PGP

PGP (Pretty Good Privacy) [Atkins et al. 1996] refers to a class of systems developed to support encrypted email communications using a combination of symmetric encryption and public key encryption technique. It was created by Philip Zimmermann in 1991 and the subsequent versions were released with his guidance and are owned by PGP Corporation. The major services offered by PGP includes the following

- Digital signature service offered by PGP involves sender creating a hash code of the email message and encrypting it with the private key. The receiver could decrypt the hash code with the corresponding public key and compare it with the hash code of the received message for authenticity. The signatures are normally attached to the message although it could also be sent as a separate entity after the message.
- Confidentiality of the email message is supported by the public key encryption. Sender generates a random key called session key and encrypts the message using that session key. The session key is then encrypted with the receiver's public key. The encrypted session key and the message are sent to the receiver, where the session key is decrypted using the receiver's private key. The decrypted session key is then used to decrypt the message. If the message includes a signature, both the signature and message are encrypted using the session key at the sender's side.

### 2.1.2.2 OpenPGP/MIME

OpenPGP [Callas et al. 2007] refers to a standard for open source security implementation based on PGP standard 5.x. The initial standard for Internet text messages [Crocker 1982] supported only the transfer of text messages through email. Other formats like images or audio needed to be transformed to reversible textual byte code for transmission using the initial standard. MIME (Multipurpose Internet Mail Extensions) [Freed & Borenstein 1996] specifies a mechanism to transmit data including text, images and audio files. Email messages with MIME headers could be broken down into multiple parts each having its own content type such as text, html and image.

Initially PGP was integrated with MIME using a special content type ‘application/pgp’ where the signed messages are included into the body part. But the email clients were not able to decode the messages without separating a specific PGP implementation details as part of the signature. OpenPGP/MIME [Torto et al. 2015] specifies a mechanism to separate signature and the message body so as to send encrypted messages using MIME. OpenPGP/MIME mechanism includes the introduction of following content types

- The content type “application/pgp-signature” is used to denote signature part of the message’s multi-parts.
- The encrypted message is included as a two-part multipart with content type "multipart/encrypted" and protocol parameter value "application/pgp-encrypted". The first part contains the version number while the second part contains the actual encrypted data.
- The content type “application/pgp-keys” is used to transmit public keys as part of the email message.

### 2.1.2.3 S/MIME

S/MIME (Secure/Multipurpose Internet Mail Extensions) [Ramsdell & Turner 2010] specifies a mechanism to transfer secure MIME data using HTTP in addition to other supported protocols. S/MIME introduces a content type “application/pkcs7-mime” to specify the secured MIME parts. The messages are secured using Cryptographic Message Syntax (CMS) based on PKCS7 (Public Key Cryptography Standard 7) [Kaliski 1998].

The user needs a key (private key) and certificate pair to use S/MIME mechanism. The certificate contains the public key information with a signature using private key. The certificate is entitled to an individual and should be obtained from a certificate authority (CA). The free certificates can be obtained from online security entities such as StartSSL [StarCom Ltd. 2011] and Comodo [Comodo CA Limited 2015]. The sender signs the message using the private key and encryption is performed using the receiver’s certificate. The receiver could authenticate the origin signature using the sender’s public certificate and the message could be decrypted using the receiver’s private key.

## 2.2 Related Work

This section includes the existing end-to-end encryption tools and the encryption search techniques that were researched during development.

## 2.2.1 End-to-End Email Encryption Tools

The following end-to-end email encryption tools were investigated during the development of the library.

### 2.2.1.1 Mutt

Mutt [Elkins & Blosser 2014] is a Linux based email client that uses GPG to provide support for OpenPGP/MIME email communication. GPG (GNU Privacy Guard) [Koch et al. 2015] is a free implementation of the OpenPGP standard and is developed as part of the GNU project [Free Software Foundation 2015]. Moreover, Mutt includes a support for S/MIME encrypted emails as well.

Mutt provides support for searching encrypted email messages [Gilles 2012], by decrypting each email and searching in the decrypted content. In addition to being highly inefficient when searching email, the Mutt tool works only on Linux environment. Notmuch [Notmuch 2014] is a Linux based mail indexing and searching tool that could be integrated with Mutt. Whenever a search is made, the results are stored and indexed in a special directory to support future search rendering the messages vulnerable to attack if the indexed directory is compromised. This project identifies an approach to build a secure index that supports efficient search using a platform independent Java library.

### 2.2.1.2 ProtonMail

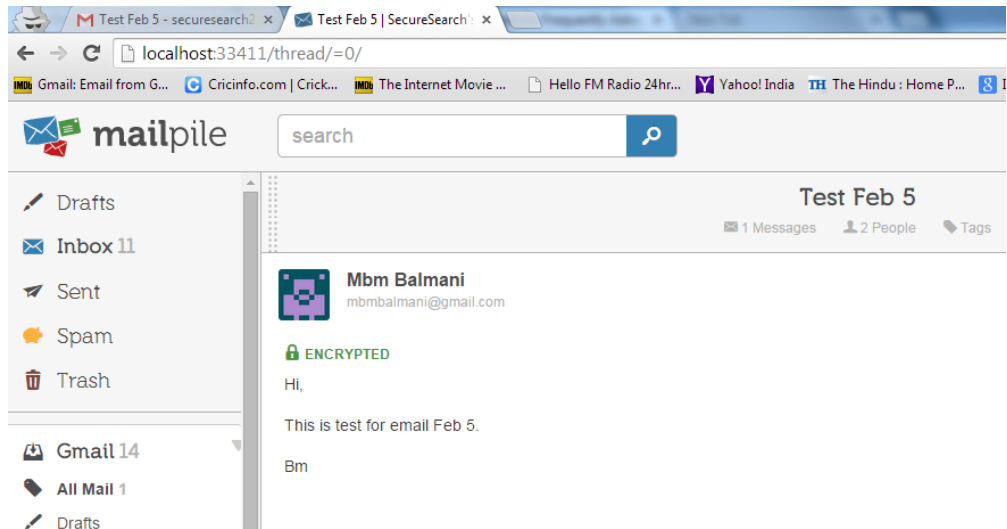
ProtonMail Beta [Yen et al. 2015] provides a web based mail service like Gmail except with the support for end-to-end encryption ensuring privacy. ProtonMail servers are deployed at Switzerland thereby protected by strict Swiss based laws for data privacy.

ProtonMail uses OpenPGP standard and open source cryptographic libraries for encrypted email communication, where all the encryptions and decryptions take place at the end user system. The private and public keys are generated in the client's system when a new user signs up for a ProtonMail account. All public keys are stored directly in the server whereas the private keys are encrypted using the user's password on the client's system and the encrypted private key is stored on the server.

In addition to OpenPGP communication, users can send regular un-encrypted email or email encrypted using symmetric encryption to other users without encryption support. In the later case, the receiver would receive a link and upon entering the shared password (symmetric key) the message is decrypted. ProtonMail is still in beta stage, and is not clear if there is search support for the encrypted email messages.

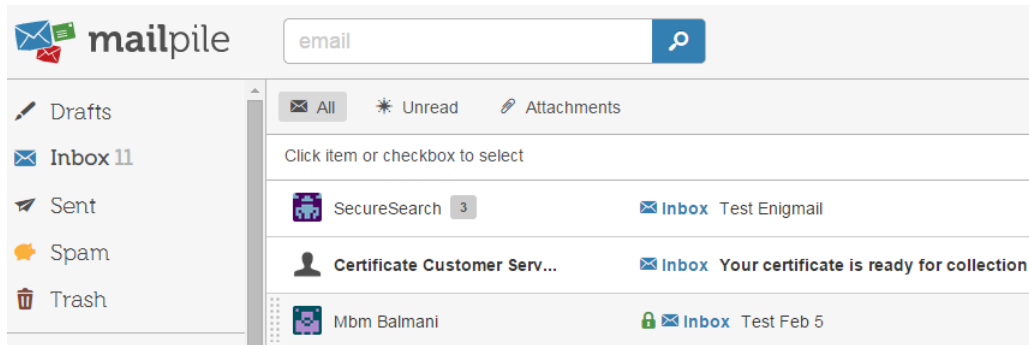
### 2.2.1.3 Mailpile

Mailpile [Einarsson et al. 2015] is an open source software written in Python that could be installed on the user system. It provides a web interface on top of existing mail servers and it enhances the services with the functionality to send and receive OpenPGP encrypted messages. The encrypted messages are automatically decrypted for viewing as shown in the next figure.



**Figure 2-1 OpenPGP decrypted message using MailPile**

Mailpile is still in beta stage, where the connection to Gmail server was intermittent during the analysis of the tool as part of this project. Moreover, it is not clear how the key management is supported by the tool. Interestingly, Mailpile supports searching of the OpenPGP encrypted messages. As an answer to one of the FAQs (Frequently Asked Questions), Mailpile’s reply regarding the security of the search support is “The search index is stored encrypted, and the terms are stored as hashes. Good enough?” The figure below shows the search results including the encrypted email (one with green lock icon) for search word ‘email’.



**Figure 2-2 Mailpile search on encrypted email**

The encryption algorithms used and data structures used to store index or the technique to ascertain the security of search is not documented and one with Python expertise could review these aspects as part of the source code. If the created index is an inverted index with index pointer and search terms encrypted, then it is vulnerable to frequency analysis attack where depending on number of indexes (messages) a hashed search term occurs, the attacker could identify the hashed search term. This project identifies an approach that is analysed for security attacks including frequency analysis attack and moreover the performance of the search operation is evaluated.

### 2.2.1.4 Mailvelope

Mailvelope [Oberndörfer 2014] is an extension plug-in for browsers based on OpenPGP.js java script library to support OpenPGP encrypted email communication. The plug-in interface is woven into the existing web interface provided by the popular web email providers such as Yahoo, Gmail and GMX. In addition, Mailvelope provides support for key management where a user could generate, import or export keys for OpenPGP encryption.

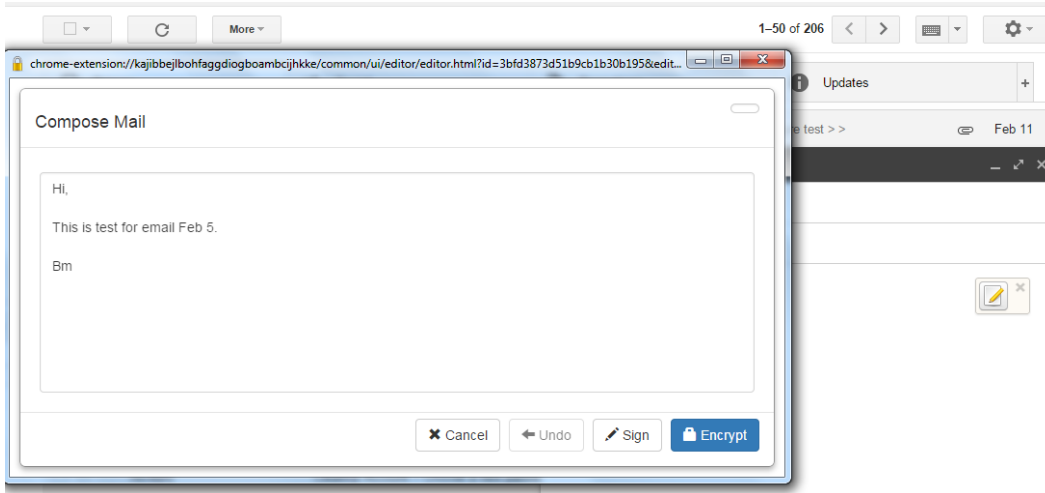


Figure 2-3 Mailvelope mail compose editor with Gmail

The figure above shows the compose editor provided by Mailvelope and is weaved together with the Gmail composer. After encryption, the message could be transferred to the Gmail editor and sent to the intended recipients. The figure below shows the corresponding encrypted message which is shown with a lock sign and decrypted with a click if the receiver has the private key with Mailvelope installed. Since the email messages are encrypted, the default Gmail search would not work on them and Mailvelope has no support to search such encrypted messages as well.

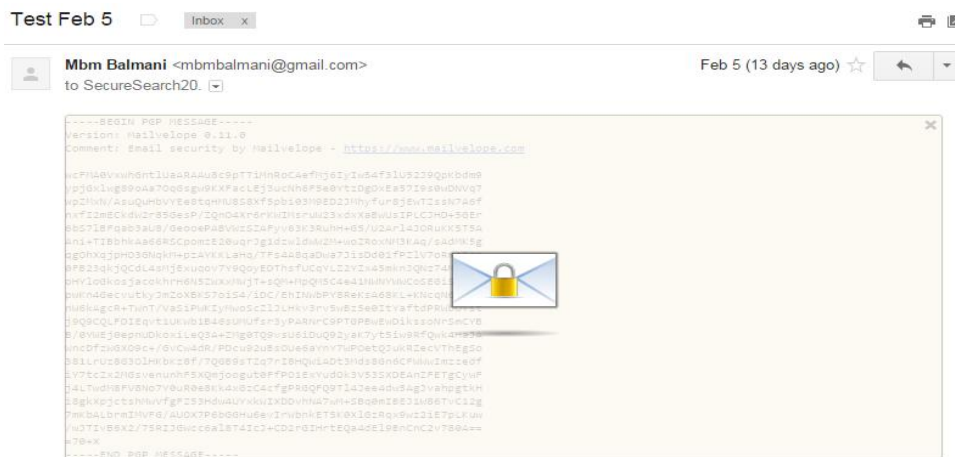


Figure 2-4 OpenPGP encrypted message using Mailvelope



### 2.2.1.5 Thunderbird

Thunderbird email client [Mozilla 2015] provides support for S/MIME encrypted email communication. Moreover, it includes a Certificate Manager to store sender's certificate and import other people's public certificate. This is depicted in the People tab of the figure below with an imported public certificate of [mbmbalmani@gmail.com](mailto:mbmbalmani@gmail.com). Similar support is provided by outlook email client [Microsoft 2015] as well.

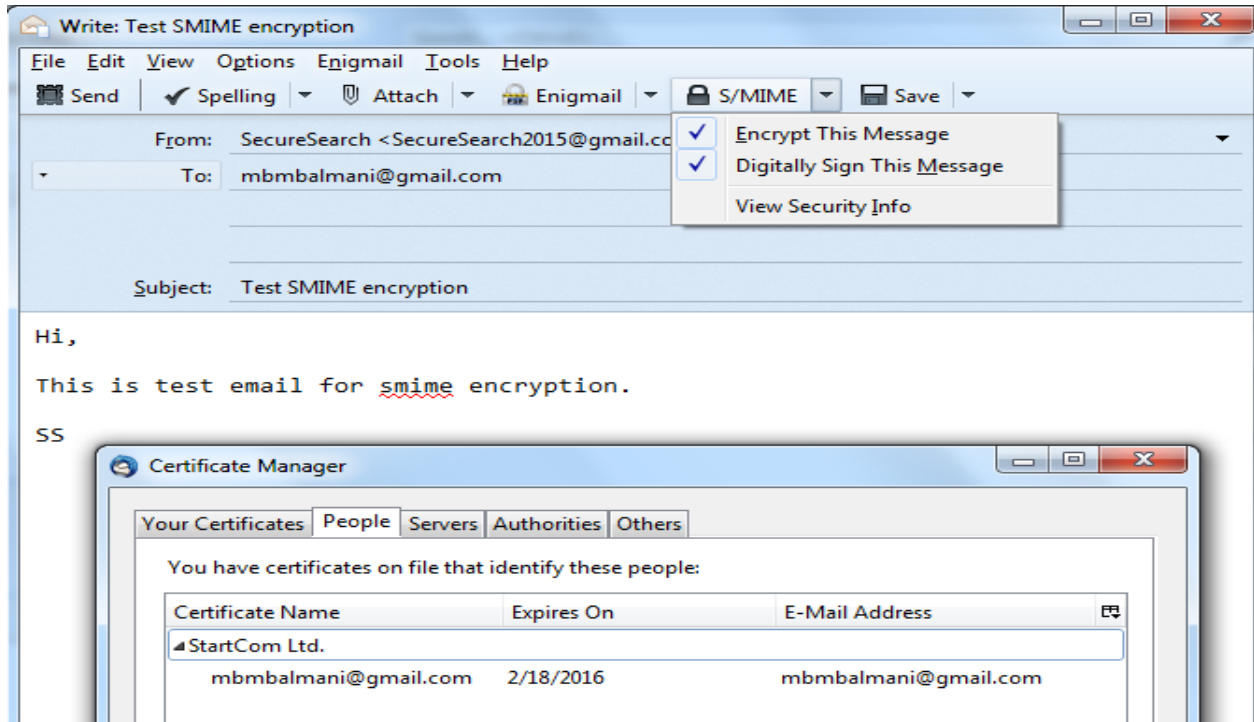


Figure 2-5 Thunderbird S/MIME Encryption with Certificate Manager

An enhancement [Chang 2005] to support search on the encrypted messages communicated using Thunderbird was filed initially on 2005. The enhancement initially suggests storing the decrypted messages in local folder for search capability and link both encrypted and decrypted messages. Owing to the open source nature of Thunderbird implementation and extensions made in S/MIME standard during the ten year span, the original enhancement still remains in the NEW status and is yet to be implemented. This project identifies an approach to store only the secure index for searching instead of storing decrypted form of all messages. Moreover, the secure index could be stored in a remote un-trusted storage and shared with other local machines.

### 2.2.1.6 Enigmail

Enigmail [Brunschwig 2015] is an extension to Thunderbird to support OpenPGP secure email communications. As shown in the next figure, Enigmail has a key management facility to generate and import keys.



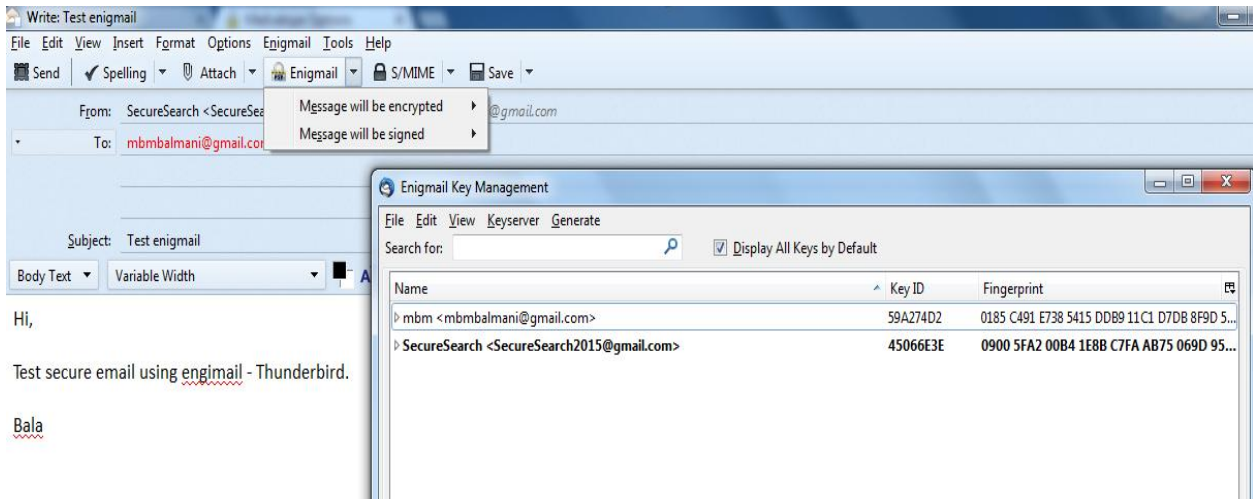


Figure 2-6 Encryption using Enigmail with Key Management

The encrypted messages are automatically decrypted on view with Thunderbird, provided the corresponding private key is available in the key store. For the first time, Thunderbird prompts for the password of the private key as shown in the figure below. The private key is then stored in cache for limited time during which other encrypted messages are automatically decrypted on view.

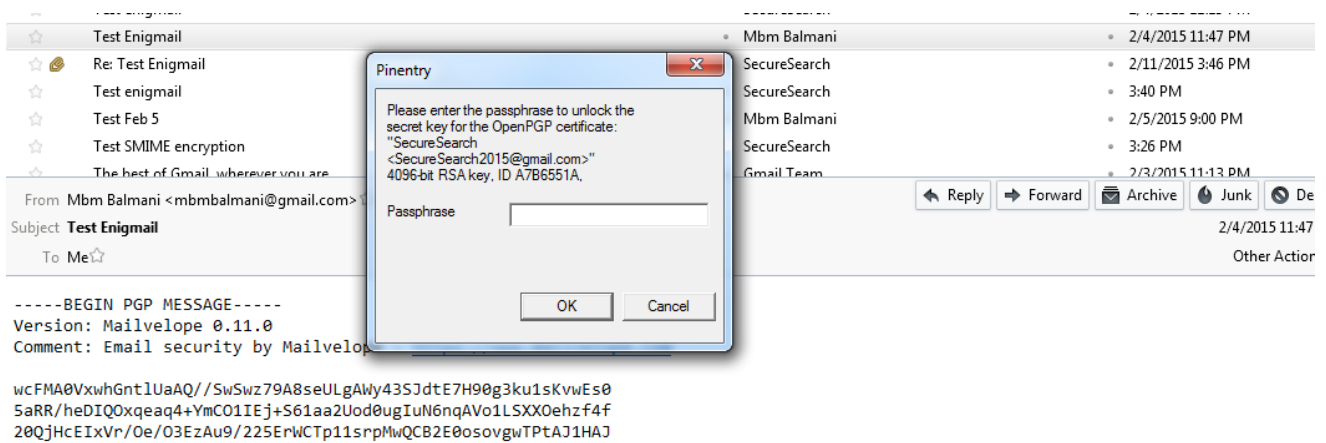


Figure 2-7 OpenPGP decryption using Enigmail

## 2.2.2 Search Techniques on Encrypted data

Storing data in the encrypted form requires trade off in terms of other functionalities such as searching data which becomes computationally or spatially inefficient. The following encrypted search techniques are analysed for its applicability as part of the end-to-end email encryption. The following sections outline the overview of each technique and analyses the applicability as part of the project.

### 2.2.2.1 Forward Index

Informally, Forward index refers to the indexing technique for documents where each document's unique id points to a set of words contained in that document. Hence the search for a word using forward index would require sequentially scanning each document and comparing

against all unique words in each document. If  $N$  is the number of documents and  $M$  is the average number of unique words in a document then  $O(N.M)$  comparisons would be required.

#### 2.2.2.1.1 Applicability

Forward index technique is inefficient in terms of space required to store the data and number of computations required to search. It imitates how the documents are normally stored and is not typically used for searching. This project uses a cryptographically secure technique similar to the forward index approach to create a secure index where the encrypted document id points to a secure Bloom filter representing hashed set of words, instead of storing the words themselves.

#### 2.2.2.2 Homomorphic Encryption

Homomorphic encryption refers to a class of encryption techniques that allows a range of computations to be performed on the encrypted data without decryption. This project is interested in a homomorphic encryption technique that enables secure and efficient searching on the encrypted data. Searchable homomorphic encryption technique is diagonally opposite to the naive approach of decrypting all the data and searching over it.

Xiaodong et al. [2000] discusses an approach where each word ( $W_i$ ) is XORed (Exclusive Or) with a pseudo random bits ( $T_i$ ) generated using  $W_i$  and the location  $i$ . The XORed values (ciphers  $C_i$ ) are stored in the database. To search for a word ( $W_s$ ),  $W_s$  and pseudo random bits for that word ( $T_s$ ) are input to the searcher which compares  $T_s$  with XOR of each cipher word ( $C_i$ ) and  $W_s$ . This technique allows searching for a word as well as recovering the document (decrypting all words) if needed.

#### 2.2.2.2.1 Applicability

There are number of similar search techniques [Thian et al. 2005; Chang & Mitzenmacher 2005], which require storing the data in a new searchable encryption form. Homomorphic encryption techniques are a current area of research in many industries especially the cloud service providers. But this project identifies a searching technique that is built on top of currently used encryption protocols for email communications such as OpenPGP and S/MIME. Moreover, the indexing techniques are analysed for applicability, where it is not necessary to recover the words from index.

#### 2.2.2.3 Inverted Index

A typical indexing technique is an inverted index tree, where each word ( $W$ ) points to a set of document pointers (document's unique id) whose documents contain the word ( $W$ ). Hence the search for a word using inverted index would require scanning of all the unique words across all documents. This approach is space efficient by storing only the unique words across all documents and is typically stored in a tree structure requiring  $O(\log N)$  comparisons where  $N$  is the number of unique words across all documents.

Xiaodong et al. [2000] discusses a possible way to securely store and search the inverted index. This would involve encrypting the words and document pointers associated with each word. This is vulnerable to [frequency analysis attack](#) as it allows easy analysis of number of occurrences of a word. To tackle against the frequency analysis attack, the document pointer list could be maintained with identical size for all the words. This is possible by including additional dummy document pointers to the documenter pointer list of the words with fewer occurrences.

#### 2.2.2.3.1 Applicability

This project agrees with Xiaodong et al. [2000] that it would be a good research area to identify how an addition of a new document could be securely handled. If a new document (D) is added and the document pointer list of an existing word (W) in the inverted index is not updated, then it implies that the word (W) is not contained in the document (D).

This project visualises one possible way to securely store an inverted index of N documents is to have each word in the index point to a document pointer list with size N, indicating the presence of the word in all the documents. Hence an addition of a new document would require adding one entry or updating the document pointer list of all words in the index tree. Even then, if there is a new word ( $W_N$ ) in the added document (D), then the index would need to add the word ( $W_N$ ) with document pointer list thus leaking the information a new word ( $W_N$ ) is not contained in all the previous documents, but only in the new document (D). Thus creating a secure index using this technique would require all words stored in the index initially which is inefficient and impractical.

#### 2.2.2.4 Secure Index based on Bloom Filter

Bloom filter [Bloom 1970] represents an array of bits of length M used to store a set elements N. Each element  $N_i$  is represented using r set bits in the Bloom filter. The mapping of an element to the Bloom filter is performed with the help of r hash functions ( $h_1, h_2 \dots h_r$ ) each of which maps an element to an index 1 to M of the Bloom filter.

To search for an element  $E_s$  in the Bloom filter, r hash functions are applied on the element  $E_s$  and the r set bits in the Bloom filter at the indices resulted from r hash functions indicate  $E_s$  may be present in the set. As with any hashed mapping, there is a possibility of overlapping and a probability for false positive i.e., the search for an element  $E_n$  which is not in the original set could be perceived as present as the bits for the  $E_n$  could have been set by the other elements. But if any of the r bits is not set in the Bloom filter then  $E_s$  is definitely not present in the set.

Eu-Jin Goh [2004] proposes an approach to build a secure index using Bloom filter. This involves maintaining a Bloom filter for each document. To index a document, all the words from that document are extracted and hashed using a key. Each hashed word is then used as a key to hash that document's unique id, so that the presence of a same word in multiple documents is not leaked into the corresponding Bloom filter. That is if a same word is present in multiple documents, different r bits will be set in those document's Bloom filter owing to their unique document id. Then the r hash functions are applied on the final hashed word to populate the Bloom filter.

To search for a word in the index, the search word is first hashed using the same key used for indexing. This hashed word is called trap door as only an authorised person with access to the key could generate the trap door and proceed further for search. For each document, the trap door is used as a key to hash the document id and then the final hashed word is searched for its occurrence in the corresponding Bloom filter with the help of r hash functions.

#### 2.2.2.4.1 Applicability

Goh's Bloom filter technique is secure with maintaining an index for each document. It is faster requiring  $O(N)$  to search for a word in N documents. The Bloom filter does not enable retrieval of the indexed words which is not necessary for indexing email messages as part of this project.

The implementation of Bloom filter is space efficient with a BitSet data structure where each bit could represent a presence of an entry (word) in the set (document).

The major drawback with the Bloom filter technique is the false positive rate associated with the technique. The false positive rate can be controlled, in order to decrease the false positive percentage, the number of bits  $M$  or the size of the Bloom filter needs to be increased which tradeoffs with the space efficiency. The false positive association with email search is acceptable from the view of this project as a user could always view the message and ignore the result. This project aims to set the false positive rate configurable, so that the indexing is usable for all users with different space constraints.

#### **2.2.2.5 Hybrid scheme for search**

This technique [Thian et al. 2005] is a hybrid of Bloom filter [Goh 2004], encrypted sequential scan and encrypted inverted index techniques [Xiaodong et al. 2000] that are discussed in the previous sections. Hybrid scheme aims to combine the good properties of the previous techniques such as easy integration with the support for indexing any data of the Bloom filter technique and preserving the stored words for decryption of Xiaodong et al. [2000].

The scheme involves maintaining a hash table as an index per document similar to the Bloom filter model. At first words from the document are extracted and encrypted. Then hash of each encrypted word and the document id of  $D$  is used as the key to  $D$ 's hash table. The encrypted word XORed with pseudo random bits as in the technique described by Xiaodong et al. [2000] is used as the corresponding value in the hash table. The search for the word ( $W_s$ ) is possible by encrypting the search word and then hashing with the document id to check for the  $W_s$  occurrence in that document's hash table.

Hence this scheme uses hash table technique from Bloom filter for indexing each document along with encryption and decryption technique from Xiaodong et al. [2000] for preserving the indexed words for decryption.

##### **2.2.2.5.1 Applicability**

Thian et al. [2005] performs a comparison of this hybrid scheme with the parent techniques. Hybrid scheme has an average processing run time compared with other techniques as the Bloom filter and inverted index techniques are much faster than the hybrid scheme. This project perceives the encryption and decryption of each word would increase the indexing and searching time considerably. This technique would be a good alternative if it is necessary to retrieve the indexed words from the index storage.

### **2.2.3 Email Client Integration**

The library that is developed as part of the implementation needs to be integrated with existing email clients to depict the suitability for practical use. The following sections briefly summarises the class of email clients and the possible integration approaches that were considered for the integration of the developed library.

#### **2.2.3.1 Thin Email Clients – Web Browser**

The web interface in the browsers connected to email server acts as a thin email client for accessing email messages. The implemented library should complement the existing browser plug-in tools that support end-to-end encryption. The integration is possible with a separate

trusted index server that uses the exposed searching and indexing APIs (Application Programming Interfaces).

For indexing, this project foresees the following two alternatives

- End-to-end encryption plug-in tools for browsers could be extended to connect to the index server and then send the decrypted message contents for indexing whenever the plug-in tool is invoked to view/decrypt the encrypted message. The key management is handled as part of the end-to-end encryption plug-in tools.
- The index server periodically queries the mail server for new messages and indexes them. In this case, the index server should handle key management.

For searching, the index server needs to be queried for the result message ids and the mail server needs to be invoked to display the search results (messages). This could be part of the existing end-to-end encryption plug-in tool or a separate extension if the latter option is used for the indexing process.

Consequently this approach requires an additional user authentication by the implemented library for connecting to the mail server either as part of the browser plug-in tool extension or from the index server.

### ***2.2.3.2 Thick email clients - Thunderbird***

Enhancement [Chang 2005] to support search for the encrypted email communication is still in NEW Status. This project perceives the discussions in that enhancement and yet to be solved enhancement as a proof for non-trivial nature of implementing such a facility as part of the core thunderbird client.

Therefore this project examined the feasibility of developing a separate plug-in to support encrypted email searching. Such a plug-in development is possible using Javascript and there is support for C++ or Python language implementations as well. But, the extension development documents specify plug-in development mechanism using Javascript [Mozilla Developer Network 2015b; MozillaZine 2013] extensively as it is applicable to both Firefox web browser and Thunderbird email client belonging to the same Mozilla group employing many common technologies. While it is possible to invoke Java APIs as part of the Javascript plug-in code or convert the Java implementation to Javascript [Mozilla Developer Network 2015a], there is no direct support for invoking a Java library.

### ***2.2.3.3 Open Source Java email Clients***

There are number of open source email clients implemented in Java [Java-Source 2015] primarily using Javamail [Oracle 2013b], an external library for sending and receiving messages with a connection to external mail server. This project initially explored for a Java client that has built-in support for S/MIME and OpenPGP encrypted email communication.

Pooka [Pilone et al. 2001] specification indicates that it has support for PGP and S/MIME encryption. The client has only alpha support for decryption and its key management is not fully documented. This project tried to configure the encryption setup but the keys could not be imported to the client. Hence this project could not examine the support provided by Pooka for

PGP and S/MIME. Pooka has a normal search support which could be extended with the implemented library.

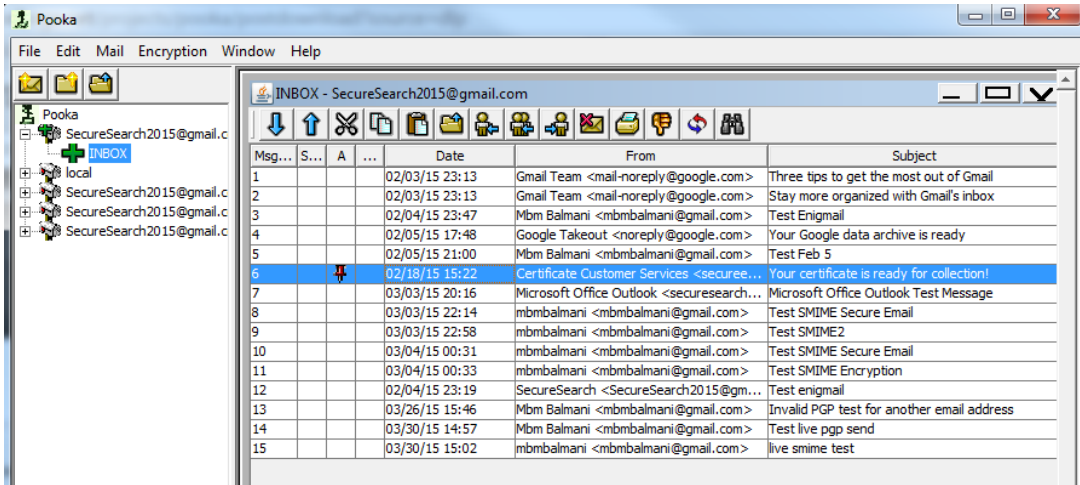


Figure 2-8 Pooka Email Client

Columba [Dietz et al. 2013] is another open source email client in Java with an user friendly interface. Though it has no built-in support for S/MIME or PGP encryption, the plug-in architecture of the client enables easy integration of new features. Despite the fact that the open source Java clients are slow owing to the Java swing UI, this project aims to identify the integration points that help with integration of the implemented library with any email client.

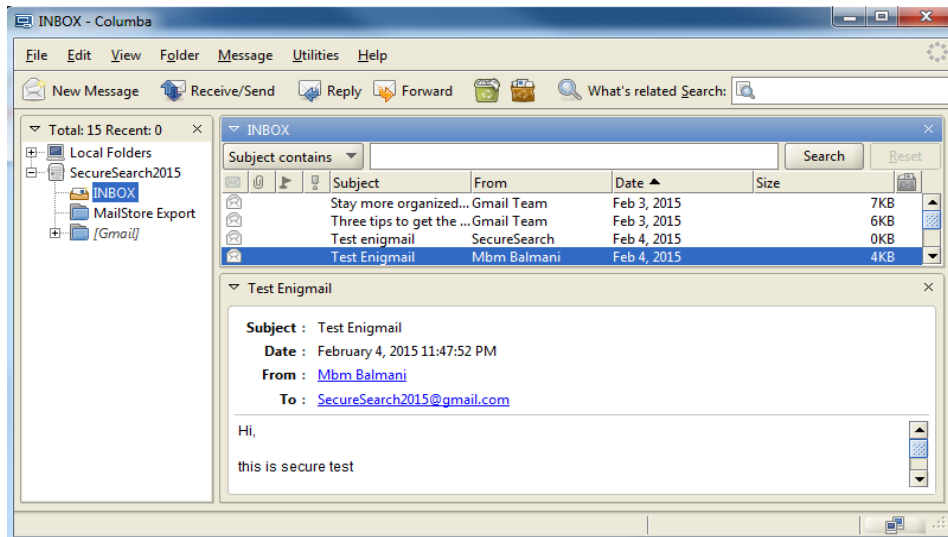


Figure 2-9 Columba Email Client



## 3 Strategy

The project's goal is to identify a technique that supports securely searching encrypted email stored on un-trusted storage. This involves researching the existing cryptography techniques and deciding an approach suitable for email messages. The chosen approach needs to be implemented and integrated with an email client to depict the suitability of that approach in practice.

The following sections describe the development and testing strategies employed for this project.

### 3.1 Development strategy

This section outlines the strategies employed for the selection of technique and the corresponding implementation. As described in the following sections, the selection and choice adopted for one influence the other.

#### 3.1.1 Technique selection

The following strategies were adopted in the selection of cryptography technique and they are listed in the order of importance with higher order first.

- The primary factor influencing the selection is the feasibility of implementation of the technique using the libraries that are currently available.
- The technique should support securely storing the index data on un-trusted storage. The index data should not reveal any detail that is not deducible by an attacker with access to the corresponding encrypted email.
- This project deals with the end-to-end encrypted email communication with the encrypted email messages already being stored in the mail server; hence the index data may not be a replication of the already stored encrypted data. This implies that data stored as part of the index may need to be lesser than the original encrypted email message.
- It may not be possible to assess the run time requirements based on the technique description, as the library used affects the run time for the cryptographic operations. The following two things may be assessed
  - A typical unsecure index requires only one communication with index where a search query is sent and the results are retrieved. The upper bound of the number of communication needed for storing or securely searching index may be linear in terms of number of messages or search results.
  - The number of comparisons needed for searching should not exceed the naive comparison limit i.e., the number of comparisons where each word in the encrypted data is matched against the search word.

#### 3.1.2 Implementation strategy

The following lists outline the strategy adopted for the implementation.

- This project follows an incremental prototype based approach where an initial prototype is developed which is extended to the final implementation.
- The cryptography library chosen should be stable with some evidence of its usage in successful projects. Moreover, the support for issues should readily exist with some answered questions.

- This project may choose offline mail storage for its initial development and further evaluation but there should be some evidence of its usability in a live environment.
- Any research project has its uncertainty with the implementation time line; this project may stick with the original plan being broken down to individual tasks after consultation with the supervisor. And there may not be a blocker issue more than a week without seeking advice from supervisor.

## **3.2 Testing strategy**

This project employs JUnit testing, scenario based testing and scalability tests as the techniques for evaluating the project.

### **3.2.1 JUnit and Scenario based testing**

JUnit tests shall be used for testing each functional module in the implementation. The effectiveness of such tests shall be measured through a code coverage tool and shall have at least 80 percent code coverage. The tests shall be part of the same project as the source code or a different test project with dependencies on the source project.

This project shall employ manual scenario based testing for testing end functionality which cannot be tested through unit tests. These tests would involve manual execution of operations in a live environment requiring human interpretation.

### **3.2.2 Performance**

The performance of the implementation shall be tested on a corpus of encrypted email messages. The performance shall be measured in terms of time required for indexing and searching as well as space required for storing indexed messages. The measured time and space shall be plotted against the number of messages and length of the messages. This evaluation should aim to identify an upper bound against number and length of messages and identify areas of performance improvement or bottlenecks in the implementation.



## 4 Design

This chapter describes the overall design proposed by this project for securely searching encrypted email followed by the logical design and design decisions made for the implementation of different functionalities. This chapter concludes with the runtime and deployment view of the implemented library.

### 4.1 Solution Design

This section describes the overall design for securely searching encrypted email and its prerequisite is an end-to-end encrypted email communication setup. Therefore the existing setup should enable sending and receiving encrypted email messages and the implementation of this project complements the existing setup with secure search functionality.

Initially this section describes the core technique employed in the design followed by the notation used to represent the design diagrams. And then the logical and deployment views proposed by this project are outlined. All the design views presume that there exists an end-to-end encrypted email communication setup between User1 and User2 and the views from the perspective of User1 with the secure searcher functionality are described.

#### 4.1.1 Core Technique

This project uses [Goh's Bloom filter](#) [Goh 2004] as a core technique for creating a secure index given an email message (document) and unique message Id (document Id). This project suggests extensions to Goh's approach and recommends a practical design for securely searching encrypted email messages. The design proposed by this project includes the overall solution design, design of the implemented library and the integration design describing how the implementation could be integrated to an existing email client.

##### 4.1.1.1 Modification to the Technique

During the indexing process, each word in the document is hashed using a key and this hashed word is referred as a trap door for that word. The trap door is again used as a key to hash the document id and then the final hashed word is converted into a set of  $r$  indices and corresponding bits are set in the Bloom filter. Finally, instead of storing the Bloom filter and document id as an index, as proposed by the original technique, this project stores the Bloom filter and encrypted document id as an index.

During the search process, for the searched word the corresponding trap door word is created. The original technique proposes retrieving the index and using the trap door again as a key to hash the document id to search for that word in the Bloom filter. Because of the modification during indexing, now the searcher has to first decrypt the document id and hash the decrypted document id with trap door as the key to search for that word in the Bloom filter.

Let's say there are  $U$  unique words in a document comprising  $N$  words (words repeat in a document for example, the, a, and, are). Goh [2004] recommends adding  $X$  random words ( $X = \text{Total number of words (N)} - \text{Number of unique words (U)}$ ) to be [IND-CKA](#) semantically secure. Therefore the Bloom filter indexes of two identical length documents have same number of words mapped to the index irrespective of number of unique words in the documents. This project modifies the technique in the view of additional functionality while promising same security. Instead of mapping  $X$  random words, the repeated occurrence information of the word

can be mapped; for example, the word ‘this2’ is added to bloom filter index for the second occurrence of word ‘this’. The ‘Word<N>’, where N is the nth repeated occurrence of the word is mapped as this would lay the basis for future enhancement where the searcher could return search results ordered by the number of occurrences of the search word.

#### 4.1.1.1.1 Analysis

During the indexing process, storing an encrypted document id strengthens the security, as the document id information is not leaked with the index being stored on un-trusted storage. While this may look like an additional precaution, the main reason this project recommends this is to strengthen the index against security attacks during the search process. This design strengthens the security of the index against **Replay attack** and **Brute force attack** models. **Security** and the **performance tradeoffs** associated with this modification is analysed further in the Evaluation section.

#### 4.1.2 Design Notation

This project follows the boxes and arrow style for representing components in a design and the connections between them. Moreover, a magnetic disc symbol is used to represent a storage unit and the physical machines are labelled with ‘Machine’ in their name. In Logical view the boxes represent the logical components and the arrows represent the dependencies between the components with arrow pointing towards the depended component. In the runtime view, boxes represent executable components with arrows representing flow of control or flow of data.

#### 4.1.3 Logical View

This section describes the high level interaction between the logical components of Secure Searcher and an existing email client with end-to-end encryption setup. The functionality of each logical component is detailed below.

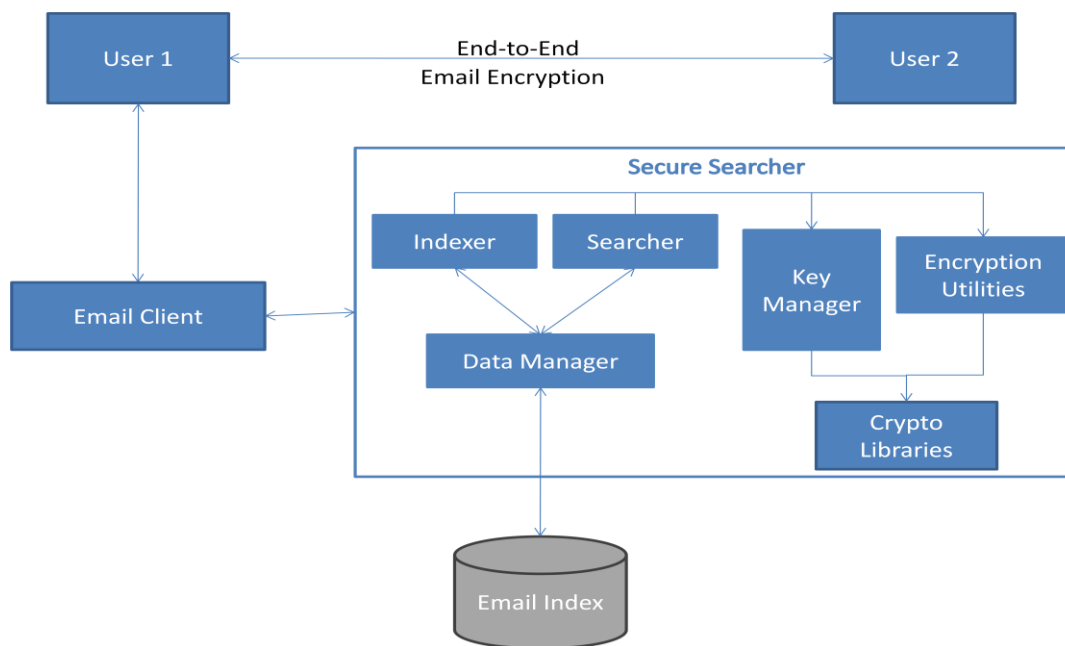


Figure 4-1 Secure Searcher Logical Design

#### 4.1.3.1 Indexer

*Email Client* interacts with the *Indexer* component for indexing encrypted email messages. This could be a live interaction i.e., the client invokes the *Indexer* whenever it receives new messages or it could be an offline interaction where the client periodically (hourly or daily) indexes the messages. Moreover, *Email Client* may pass the encrypted message to the *Indexer* or the client could decrypt the message and send only the plain text message for indexing.

The indexing is based on unique message Id associated with each email message and the index is stored in *Email Index* (database) with the help of *Data Manager*.

#### 4.1.3.2 Searcher

Email client interacts with *Searcher* component for searching encrypted email messages. *Searcher* accepts the search word and returns the unique message Ids whose message bodies may contain the searched word.

#### 4.1.3.3 Data Manager

*Data Manager* decouples the data store implementation type from the *Indexer* and *Searcher* components. *Data Manager* manages the connection to the data store and enables storing and retrieving the index data to and from the data store.

#### 4.1.3.4 Key Manager

*Key Manager* enables *Indexer* and *Searcher* components to retrieve the indexing key used for creating a secure email index. *Key Manager* exposes the functionality to import and securely store this indexing key with the help of *Crypto Libraries*.

*Email Client* may interact with *Key Manager* to store the keys used for encryption/decryption purpose as part of the end-to-end email communication. This process may not be needed if the *Email Client* handles the decryption process and sends only the plain text message for indexing. But if the *Email Client* needs the *Indexer* to index encrypted messages then the *Indexer* requests the *Key Manager* for corresponding decryption key. Hence in the latter case the decryption key need to be stored by *Email Client* using *Key Manager*.

#### 4.1.3.5 Encryption Utilities

*Encryption Utilities* exposes hashing and encryption/decryption functionalities with the help of *Crypto Libraries*. *Crypto Libraries* are collection open source standard libraries enabling full strength cryptographic operations.

### 4.1.4 Deployment View

This section describes the recommended deployment approach for the secure search functionality. The next figure shows User 1 having access to two machines Local Machine 1 and Local Machine 2 to perform email operations using a same email account.

User 1 logs in to the email client on *Local Machine 1* and checks for new messages. Any new messages are indexed in the *Email Index store* on the *Local Machine 1*. If there is a need for backup or portability, the stored secure Email Index could be backed up in a remote storage on the cloud.

To illustrate the portability of the functionality, let's say the User1 logs in to the *Email Client* from another machine *Local Machine 2*. The designed solution requires merely a synchronisation between the local *Email Index* and the remote cloud *Email Index* store. This negates the need to download and index all the already indexed email messages again on the *Local Machine 2*. *Local Machine 2* could be a new machine with *Email Client* and *Secure Searcher* installed or an alternate machine that is used by the User1 on regular basis.

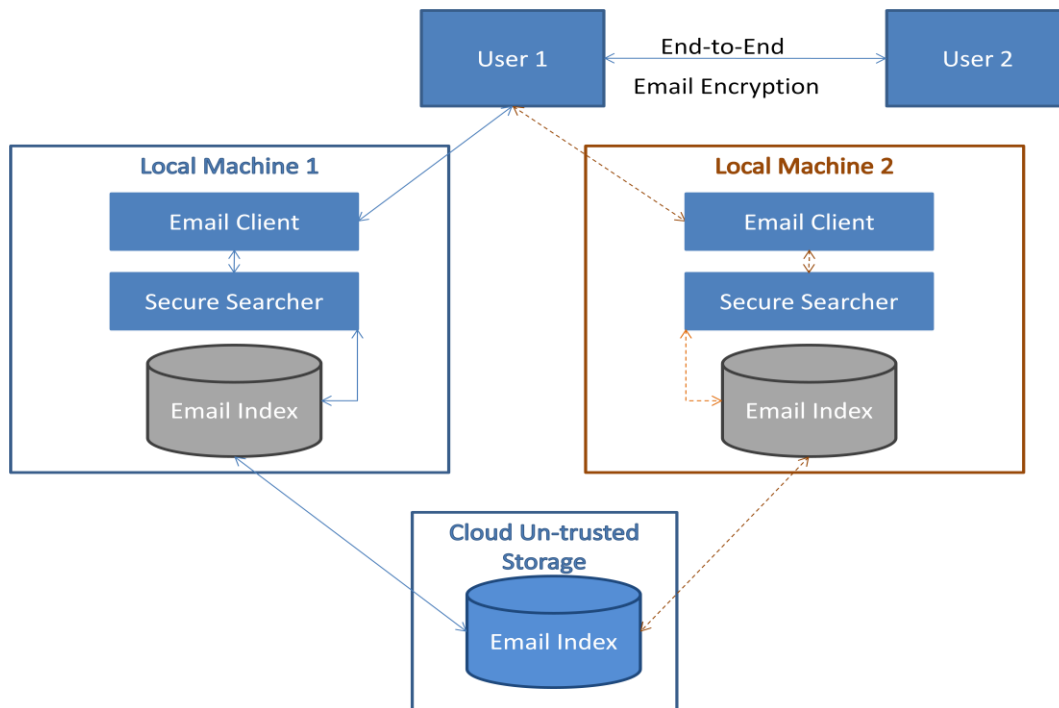


Figure 4-2 Secure Searcher Deployment Design

Though the description uses two machines to illustrate portability of the solution, the secure searcher portability could be extended to N machines.

#### 4.1.4.1 Alternative Approaches

The above approach is recommended owing to the simplicity of deployment coupled with the [security](#) of the search mechanism. This section describes the alternative approaches that were considered and reasons their limitations.

##### 4.1.4.1.1 Remote Secure Search Server

The deployment approach described in this section requires a single remote search server deployed on the cloud with which email clients could interact for indexing and searching.

This approach has the advantage of having a single deployment of *Secure Searcher* as opposed to the recommended approach's requirement of secure searcher configuration on all local machines used for email communication. A serious limitation of this approach is that the *Secure Searcher* needs access to the decrypted messages for indexing purpose. Moreover, any attacker having access to the *Secure Searcher* could perform search operation on the stored *Email Index*.

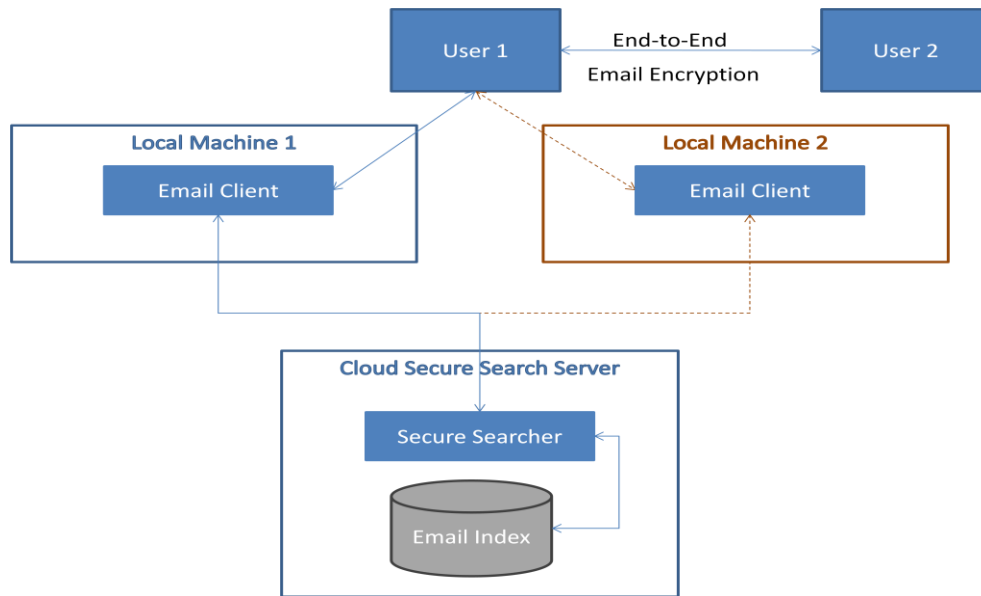


Figure 4-3 Secure Searcher Remote Server Alternative Deployment Design

Hence the remote Secure Search server needs to be a trusted server which restricts the access only to the authenticated users and protects the Secure Searcher from the **In-Memory attack** i.e., the keys and decrypted messages are available in memory of the executing program during indexing and searching purpose and is vulnerable to attack.

#### 4.1.4.1.2 Secure Searcher Client and Server

This deployment approach tries to minimise the security vulnerability in the previous Remote Secure Search Server approach by splitting the Secure Searcher functionality between the Local Machine and remote server. The solution design is depicted in the figure below.

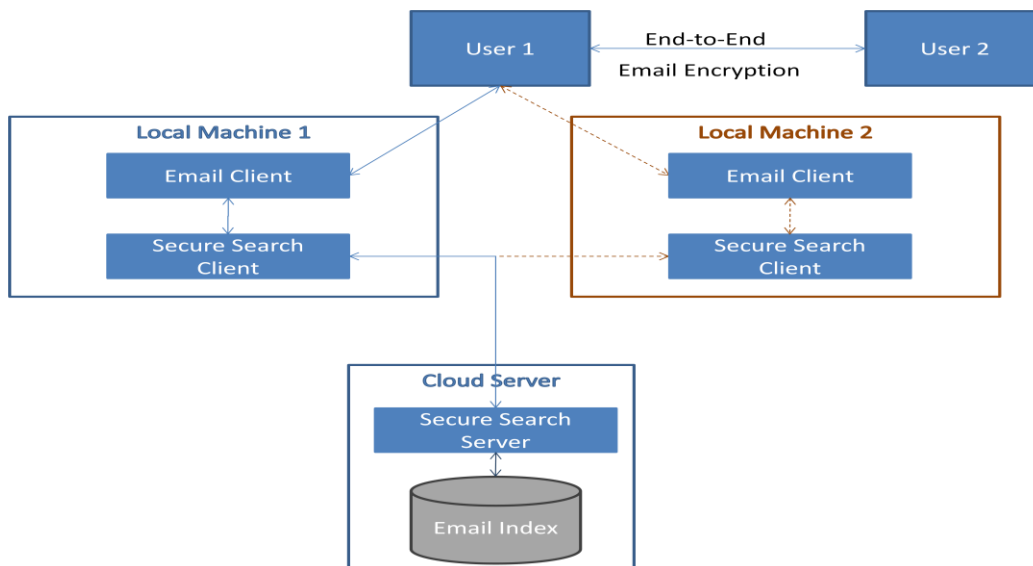


Figure 4-4 Secure Searcher Client Server Alternative Deployment Design

The functionality of Secure Search client and the server during indexing and searching process are described in the following sections.

#### *4.1.4.1.2.1 Indexing*

When a new message needs to be indexed by the Secure Search Client, the client indexes the message and sends the index (encrypted message Id and Bloom filter) to the Secure Search Server. Secure Search Server on receipt of the index, stores them in the Email Index store.

#### *4.1.4.1.2.2 Searching*

During the Search process, Secure Search Client creates a trap door for the received search word. Trap door is a hashed word created using a keyed hash function where only the client could create the corresponding hashed value while it would be hard for an attacker without the key.

The trap door is then sent to the Secure Search Server which performs the search and returns the message Ids of messages containing the searched word.

#### *4.1.4.1.2.3 Analysis*

This approach is strong against the In-Memory attack vulnerability of the previous approach as the Secure Search Server does not have access to the keys or plain text messages during indexing. Moreover, only the Secure Search Client can perform the search operation as only the client can generate trap door with a key.

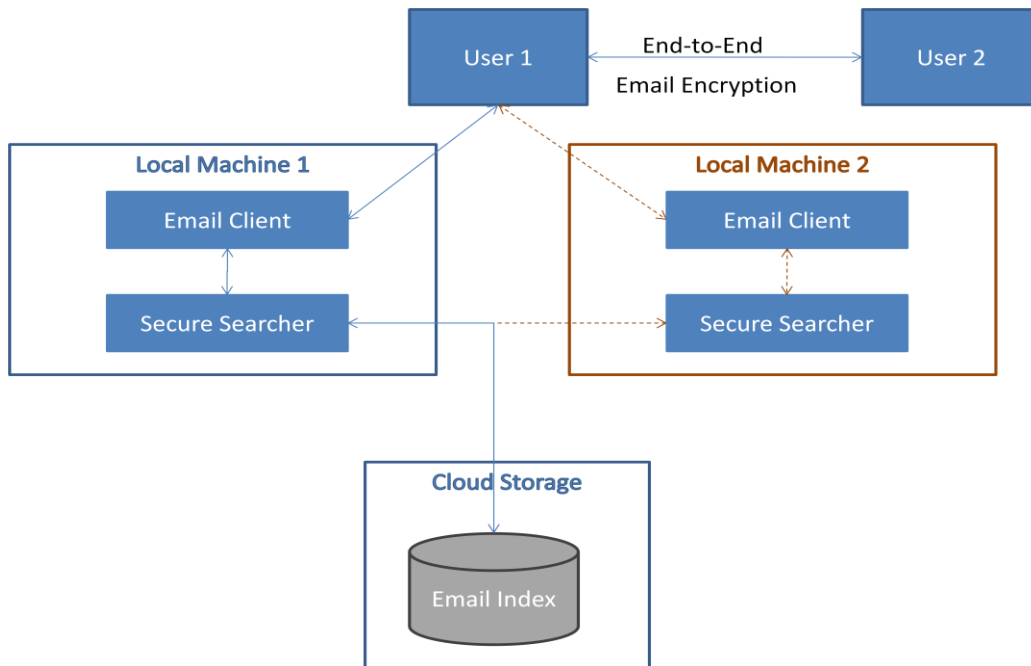
But the major limitations of this approach are during search operation, where the deployed Secure Search Server is vulnerable to frequency analysis attack. Suppose the user searches for a word 'SearchWord' then the Secure Search Client creates a trapdoor for 'SearchWord' let's say 'TrapDoorWord'. Secure Search Server receives the trapdoor 'TrapDoorWord' performs the search and returns the result message Ids. Now the attacker having access to the Secure Search Server could analyse that a certain word is possibly found in the email messages based on the number of results. The number of search results acts as a vulnerability to perform [frequency analysis attack](#). Having access to history of such searched 'TrapDoorWords' and results, the attacker could determine the corresponding 'SearchWord' for the 'TrapDoorWord' leaking the information that 'SearchWord' is present in the resulted messages.

Another serious limitation with this approach is it suffers from the [Replay attack](#). If the attacker gets access to a 'TrapDoorWord' and the remote server, the same trap door could be used in future to check for the presence of the corresponding 'SearchWord'.

Hence the remote Cloud server where the Secure Server is deployed needs to be a trusted server. Moreover, a remote un-trusted server could block or modify the search results or the index asserting the requirement of a trusted remote server.

#### *4.1.4.1.3 Single Remote Email Index Data Store*

This deployment approach is stronger than the previous split approach with only the data store being stored in the remote un-trusted cloud storage.



**Figure 4-5 Secure Searcher Remote Storage Alternative Deployment Design**

When a new message needs to be indexed, the Secure Searcher sends the encrypted message Id and corresponding index to store in the database. While searching, the stored indices are retrieved from the Email Index data store sequentially or in batch and searched for the presence of search word.

#### *4.1.4.1.3.1 Analysis*

Even though the search results or search word are not known to the remote storage, the data access history during the search operation leaks information that a certain word may be found/not found in accessed rows. Let's say the default search configuration displays 20 search results for initial search and only the first 20 rows of the Email Index data store are accessed. Then this data access pattern leaks the information that a certain word is found in each of the first 20 email index. Similarly if the search operation accesses all the rows, then the information that the corresponding search word may not be found in all index is leaked. This could act as a basis of frequency analysis attack which could be avoided using an oblivious data store that hides the history of access patterns.

This approach may be slightly more vulnerable than the recommended deployment approach. But the key reason that this project recommends a local index store in synchronisation with the remote data store is to increase the performance. The usage of cryptography libraries in the Secure Searcher produces considerable delay in the search operation. Using a remote index store for search operation might increase the delay further. But with current faster network services and the designed space efficient index, this may be negligible and needs to be asserted with further experiments.

## 4.2 Library Design

This section describes the high level design of the implemented library. It begins with the logical view followed by the description of run time interactions between the components.

### 4.2.1 Logical View

This section describes the logical components of the implementation along with the corresponding mapping to the implementation packages in the source code environment. The design follows the layered approach with the responsibilities divided among three layers Client, Server and Data Store. Each layer exposes APIs (Application Programming Interfaces) that are consumed by the layer left or above to it.

The next figure shows the major logical components in the implemented library and their dependencies. Each logical component and its functionalities are described as follows.

#### 4.2.1.1 Secure Search Client

This component encompasses the Client side functionality described in the [Secure Searcher Client and Server](#) deployment approach. It corresponds to the package `org.maynooth.client` in the implementation structure.

A separate logical component for client and server is used, even though this project recommends only a two-tier deployment approach comprising Secure Searcher with a local database on local machine and a remote database on cloud storage. The separation of logical client and server provide scope for future extensions and logical separation of functionalities.

The *Secure Search Client* handles integration with the email client and performs core of the indexing functionality where as *Secure Search Server* handles integration with *Data Store* and the core of searching functionality.

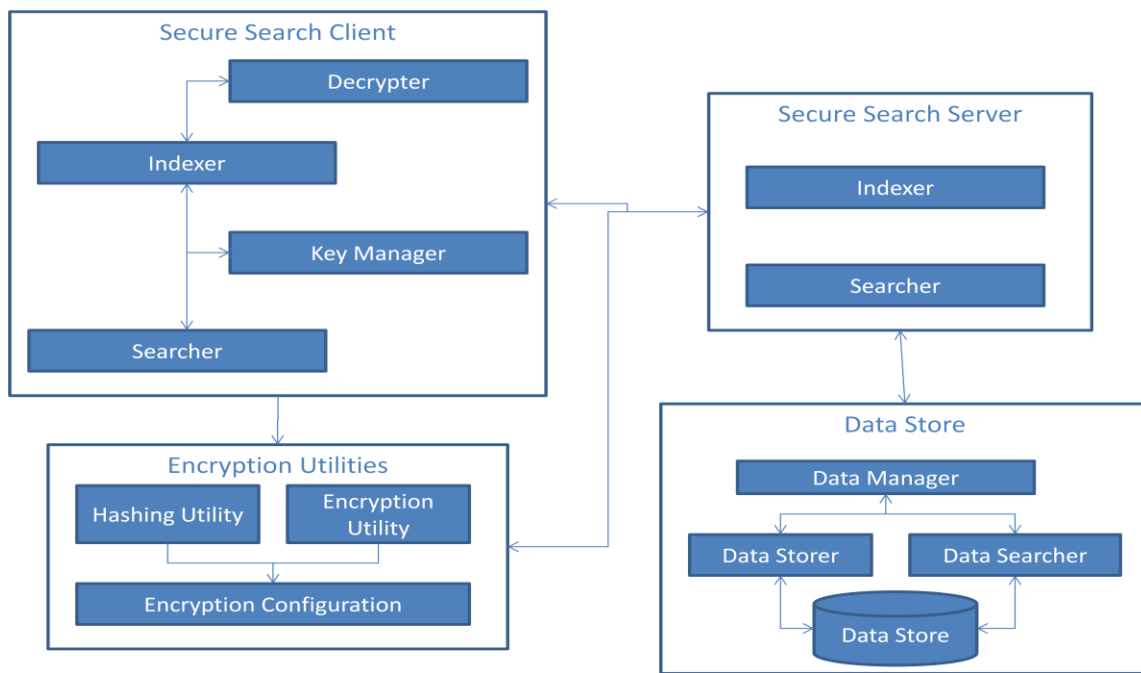


Figure 4-6 Secure Searcher Logical View



#### 4.2.1.1.1 Indexer

Client Indexer exposes APIs for indexing both encrypted and plain messages to the data store. For indexing an encrypted message it utilises the *Decrypter* component to decrypt the contents and it creates a Bloom filter data structure representing the index of the message. It passes the index information to the *Server Indexer* for storage. This component corresponds to *org.maynooth.client.indexer* package in the implementation structure.

#### 4.2.1.1.2 Searcher

Client Searcher exposes APIs for searching in the indexed messages and returns a list of message Ids whose contents may contain the searched word. Furthermore, it exposes API to search for words in a particular message and returns a Boolean value indicating presence or absence of the searched words.

For the input search word, it creates a trap door search word and passes the trap door to *Server Searcher* for searching. The maximum number of results returned depends on the Searcher configuration settings. This component corresponds to *org.maynooth.client.searcher* package in the implementation structure

#### 4.2.1.1.3 Decrypter

Indexer on receiving encrypted messages for indexing invokes the Decrypter component which employs Chain of Responsibility design pattern to perform the decryption functionality. Decrypter corresponds to *org.maynooth.client.indexer.decrypter* package in the implementation structure.

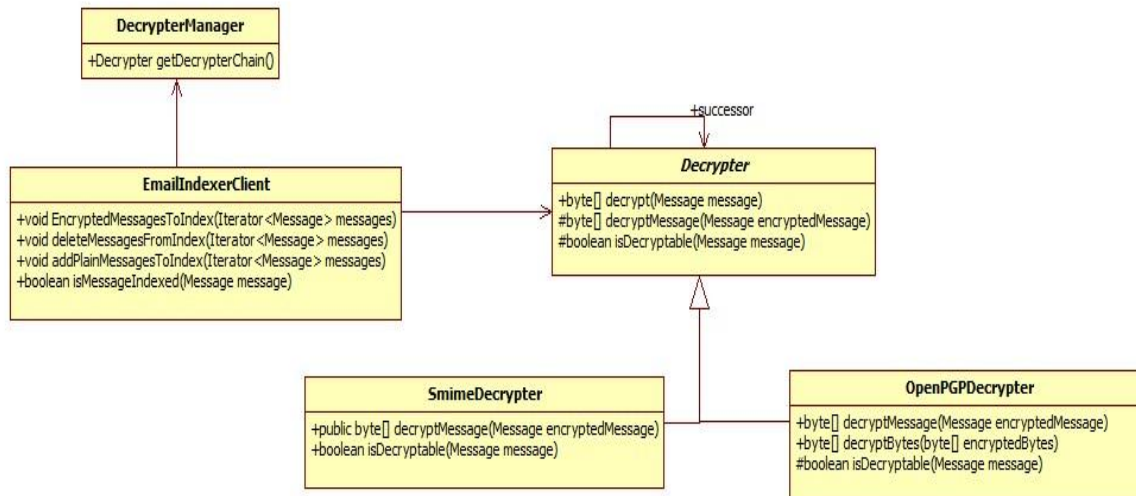


Figure 4-7 Decrypter Logical Design - Chain of Responsibility

As depicted in the class diagram above, Indexer client gets the *Decrypter* chain from *DecrypterManager* and invokes `decrypt` on the message. *Decrypter* chain first checks the *OpenPGPDecrypter* for decryption, if the message is not *OpenPGP* encrypted or if the corresponding key is not present in the key store the responsibility then goes to the next

successor *SmimeDecrypter*. If none of the *Decrypter* in the chain could handle the message then the message is not indexed.

Chain of Responsibility pattern enables easy integration for future extension of new *decrypter* functionality. This would involve specialising the *Decrypter* and modifying *DecrypterManager* to add the new specialised *Decrypter* to the chain.

#### 4.2.1.1.4 Key Manager

Key Manager corresponds to *org.maynooth.client.keystore* package in the implementation structure. It consists of three major components described as follows.

- *KeyStoreManager* enables creation of a new key store and deletion of the existing key store. It also exposes API to retrieve the key used for indexing purpose.
- *PGPKeyManager* enables importing and retrieving keys used for OpenPGP decryption.
- *SmimeKeyManager* enables importing and retrieving keys used for S/MIME decryption.

#### 4.2.1.2 Secure Search Server

This component encompasses the Server side functionality described in the [Secure Searcher Client and Server](#) deployment approach. It handles the majority of searching functionality and corresponds to the package *org.maynooth.server* in the implementation structure. The logical components of the server are detailed as follows.

##### 4.2.1.2.1 Indexer

Server Indexer acts as a connector to the data store for passing the encrypted message Id and the corresponding Bloom filter index for storage. This component corresponds to *org.maynooth.server.indexer* package in the implementation structure.

##### 4.2.1.2.2 Searcher

Server Searcher on receiving the trapdoor invokes the data store to retrieve indices and performs the requested search. This component corresponds to *org.maynooth.server.searcher* package in the implementation structure.

#### 4.2.1.3 Encryption Utilities

Encryption Utilities exposes two functionalities; a keyed hash function known as HMAC [Krawczyk et al. 1997] function and a symmetric encryption functionality. The algorithm or standard used for hashing and encryption depends on the *Encryption Configuration* settings. This component corresponds to *org.maynooth.encryptionUtil* package in the implementation structure.

#### 4.2.1.4 Data Store

Data Store corresponds to *org.maynooth.datastore* package in the implementation structure. It consists of three major components described as follows

- *Data Manager* creates and closes the connection to the database. The connection parameters depend on the *Data Store Configuration* settings.
- *Data Storer* exposes APIs for indexing purpose which includes inserting a new index, removing an existing index and querying if the index already exists.

- *Data Searcher* exposes APIs for searching the data store and returns the indices containing message id and Bloom filter index upon retrieve index operation by *Server Searcher*.

## 4.2.2 Runtime View

This section describes the run time interaction flow between the components of the library during indexing and searching process.

### 4.2.2.1 Indexing Runtime View

Indexing runtime view describes run time interaction between the logical components of Secure Searcher during a typical indexing process where a new message is indexed by the Secure Searcher.

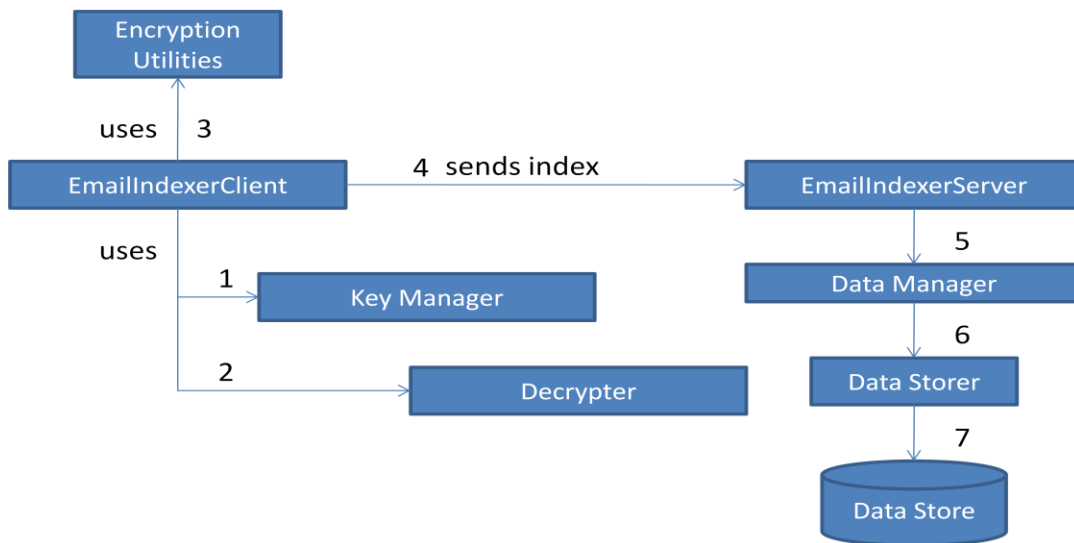


Figure 4-8 Indexing Runtime View

*EmailIndexerClient* on receiving a request to index email messages retrieves the indexing key from the *Key Manager* and then uses *Decryper* to decrypt the email message if it is an encrypted message. *EmailIndexerClient* then uses *Encryption Utilities* to create an index for the message and sends it to the server for indexing. *EmailIndexerServer* invokes *Data Manager* for getting a connection and then stores the index to the *Data Store*.

### 4.2.2.2 Searching Runtime View

Searching runtime view describes the run time interactions between the logical components of Secure Searcher for a typical search operation where the Searcher is queried for messages containing the search word.

*EmailSearcherClient* on receiving a search request invokes *Key Manager* to get the indexing key and then uses *Encryption Utilities* to create a trap door word corresponding to the search word. Trap door word is then sent to *EmailSearcherServer* which retrieves the index records from *Data Store* to search for the records that satisfy the requested operation. *EmailSearcherServer* uses *Encryption Utilities* and *Key Manager* for the search operation and returns the message Ids of messages containing the searched word to the *EmailSearcherClient*.

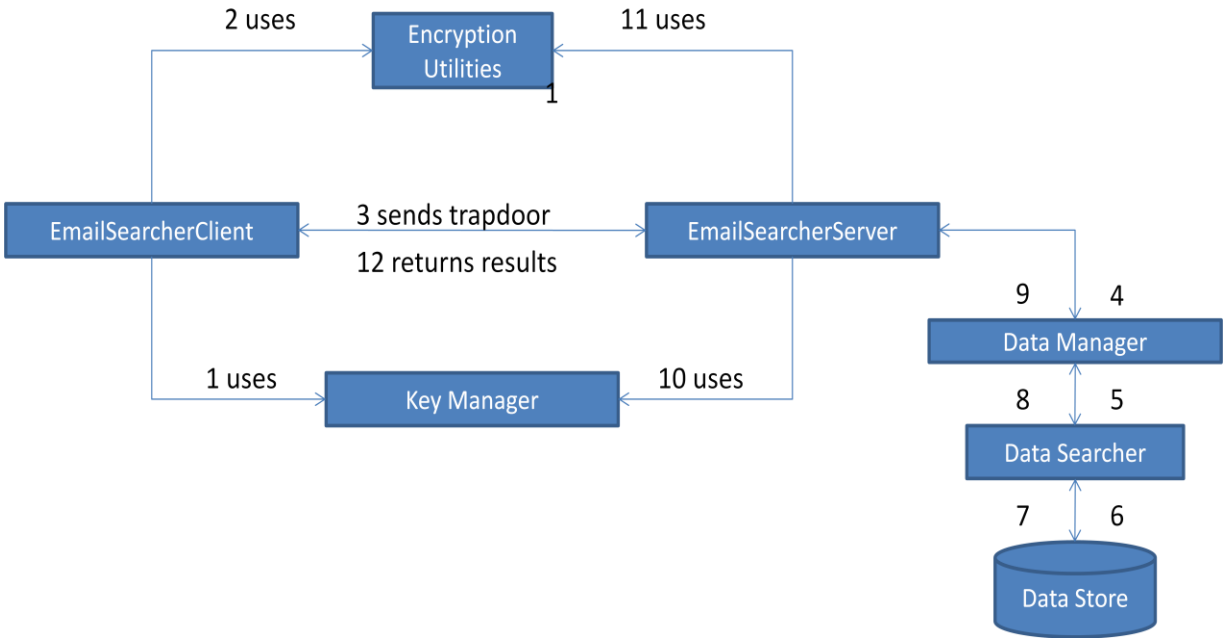


Figure 4-9 Searching Runtime View

### 4.3 Integration Design

This section describes the high level design for the integration of the implemented library with an existing email client. This section includes the integration design for indexing operation followed by the design for search operation.

#### 4.3.1.1 Indexing Integration

Integration design proposed by this project for indexing purpose using the Secure Searcher is depicted in the next figure. As shown in the design diagram, there are two high level indexing operations supported by this design numbered as 1 and 2.

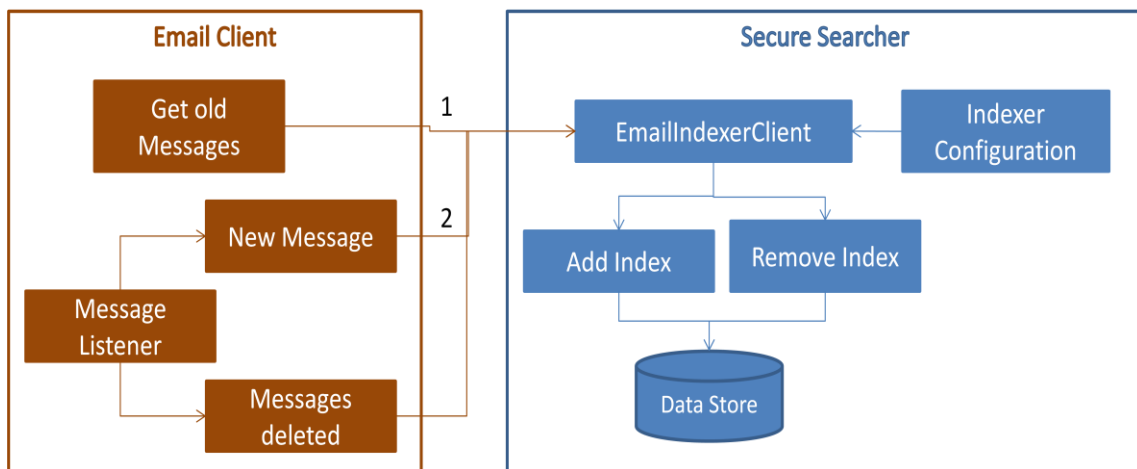


Figure 4-10 Indexing Integration Design

The first one is a support for full or first time indexing operation, where all the existing old messages in the email client need to be indexed by Secure Searcher. Email client retrieves all the messages for the configured email account and sends it to the *EmailIndexerClient*. *EmailIndexerClient* indexes the messaged based on retrieved indexer configuration property settings such as key store, database, Bloom filter and encryption settings.

The second one is a support for an incremental index operation, where new messages are indexed and the deleted messages are removed from the index store. Email client listens for any new messages or deletion of messages and invokes *EmailIndexerClient* to index or remove an existing index.

#### 4.3.1.2 Searching Integration

Integration design proposed by this project for search operation using Secure Searcher is described in this section.

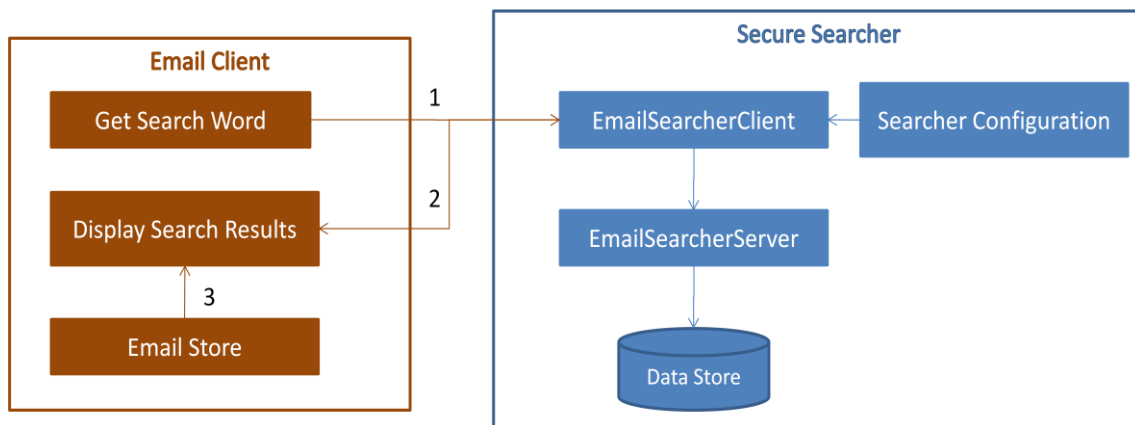


Figure 4-11 Searching Integration Design

Email client on receiving a search request from the user, invokes the search operation on *EmailSearcherClient*. *EmailSearcherClient* performs the search operation utilising the default Searcher configuration settings such as key store, searcher, database, Bloom filter and encryption settings and then returns the message ids of messages that may contain the searched word. The results are then processed by the email client and the corresponding email message data is retrieved from the Email Store for display. Email Store may be a local cache of messages stored by email client or the remote mail server containing the messages.

## 5 Implementation

This chapter details the implementation approach adhered to by the project. It begins with the core library implementation details followed by the email client integration implementation details and concludes with the encrypted email generator approach. Each section includes a summary of key decisions made and the challenges faced during the implementation.

### 5.1 Core Library Implementation

This section describes the implementation details of the core library developed by this project. At first, indexer implementation details are described, followed by the key manager and encryption utilities information. Then the approach for searcher implementation is detailed and this section concludes with the data manager details.

#### 5.1.1 Indexer

Indexer extracts the message Id from the message and encrypts it with the indexing key obtained from key manager and encryption utilities. Then it parses the message content and creates a Bloom filter as the index of the message, the index creation details are outlined as follows.

##### 5.1.1.1 Index Content

Indexer decrypts the message body and splits the message body into individual words. The subject of the email message and attachments are not indexed owing to the following rationales.

- Subject lines are not considered as they are not encrypted by the end-to-end encryption tools. This is due to the fact that the spam filters on the mail servers works on the basis of subject line of the message as well. Hence the subject line of the email message could be searched by the default search service provided by the mail service provider.
- Current tools that support end-to-end encrypted email communication do not have support for encryption of attachments and hence indexing such attachments is considered out of scope for the project.

For splitting the message body into individual words, a regular expression "[^A-Za-z0-9]+" is used. Therefore any character other than the alphabets and numbers is considered as a delimiter to split the message. Alternative approach would be to use white space characters as delimiter but if employed then the words like “Thanks,” will be indexed including comma in it. A more sophisticated delimiter would anticipate all such possibilities in words depending on the user expectations. This project sticks with the initial regular expression and translates each word to a case insensitive word for simplicity and usability.

##### 5.1.1.2 Bloom Filter

This project uses Java BitSet [Oracle 2014] data structure to represent Bloom filter. The size of the Bloom filter, number of words mapped to the Bloom filter and number of hash functions used for mappings during the indexing process determines the false positive rate associated with the search process. The number of words (N) extracted from the message body multiplied by the Bloom filter configuration parameter ‘NumberOfBitsPerWord’ (B) is taken as the Bloom filter size (M). This project uses ‘NumberOfHashFunctions’ configuration parameter to determine the number of hash functions used for mapping words to the Bloom filter.

Bloom filter size  $M = (\text{Number of words in the message } N) * (\text{Bits per word configuration } B)$

These configuration parameters are the properties files in the 'configuration' folder of the project. This could be overridden by the integrated email client and changed at the run time. More specifically, all changes to the configuration properties in the Secure Searcher should be made before running the first or full index operation. If the parameters are changed after the index operation then the search operation may not work. As the search operation expects the configuration to remain the same as it was during the indexing stage.

Each word is hashed using encryption utilities with a pseudo key as the key and then the hashed word is again used as the key to hash the message id. Then the final hashed word is mapped to the Bloom filter using  $r$  hash functions, details of the mappings are described in the next section. A pseudo key is a random byte array (32 byte) generated using Secure Random with indexing key as the key. Pseudo keys are used to generate  $n$  keys from a single key (indexing key) or for the key with required key length (32 byte) to make the implementation independent of the size of the indexing key used. It is recommended that key size be at least equal to the block size the data processed by the hashing/encryption function [Krawczyk et al. 1997], which is 32 byte (256/8) for the default hash function (HmacSHA256). 256 bits is set as the default hashing key size in the encryption configuration, though a smaller key size would be padded up with 0s on the right to get the right key size.

#### **5.1.1.3 Challenges in Indexer implementation**

A challenge during the indexing was to identify  $r$  hash functions depending on the Bloom filter configuration parameters. The initial idea was to use inbuilt hash functions with  $r$  pseudo keys generated based on the original key. But then each hashed value which is a 256 bit length value (HmacSHA256) needs to be mapped to the Bloom filter index from 0 to  $M$ .

This project decided to adapt the implementation approach used by an existing Bloom filter implementation [MagnusS 2011]. Here instead of using  $r$  pseudo keys for  $r$  hash functions, first a pseudo key is created and then the hashed value is split into 4 byte blocks (32 bits) and the absolute value is taken as an integer. This integer mod the Bloom filter size is used to map to the index 0 to  $M$ . So a single hashed 256 bit value could be mapped to 8 (256/32) Bloom filter index. If more hash functions are needed i.e.,  $r > 8$  the next pseudo key is created to generate the next hashed value. This provides a simple and light weight mapping instead of generating  $r$  pseudo keys and using  $r$  keyed hash functions which are both costly operations.

#### **5.1.1.4 Decrypter**

The major challenge with the implementation is decrypting the initial encrypted message for indexing. There are simple examples for decrypting OpenPGP and S/MIME encrypted contents using bouncy castle library [Wiki ServiceNow 2012b; Bouncy Castle 2015]. While it was straight forward to extract OpenPGP encrypted body content, extracting the message content from the S/MIME encrypted mail message was the major challenge. For S/MIME messages the default `message.getContent()` returns mime multi-parts which needs to be parsed to extract the encrypted content. The extracted content contained varied headers in the content depending on the client or library (bouncy castle) used to create S/MIME encrypted message. This project adopted an approach to get the raw input stream of the message part and parse it manually. The raw input stream of the message body part parsing needed to be modified to adopt for mail dir message format and multi-part (plain and html) message bodies.



### 5.1.2 Key Manager

Key manager provides functionalities to import OpenPGP and S/MIME keys that could be used for indexing or decryption. The major challenge was to decide on how to store the keys on the client machine. Initially password protected S/MIME pfx (Personal Information Exchange) file and the default encrypted OpenPGP key was used by the Key Manager. While S/MIME could be imported to a JKS (Java Key Store), this project could not find a standard way to store OpenPGP keys that is usable by a Java library. This implementation converted the imported OpenPGP key to a key and certificate pair so that it could be stored in JKS [Bouncy Castle Inc 2014]. This approach was used so there is a common key store for both the keys and the key store is protected by a single password. This promotes usability requiring user to input a single password for indexing or decryption of both OpenPGP and S/MIME messages, irrespective of the passwords used to protect the source S/MIME pfx file and OpenPGP keys.

### 5.1.3 Encryption Utilities

The encryption utilities provides two functionalities, a keyed hashing function called HMAC [Krawczyk et al. 1997] and a symmetric encryption function. The bouncy castle crypto library [Bouncy Castle Inc 2013] is used for the implementation of these functions. Bouncy castle was used as it provides support for both OpenPGP and S/MIME from older release of JDK 1.3 and there is good support online indicating the use of library on other successful products. While there are number of libraries for GPG [Yemini 2010; daniele athome et al. 2015], One other library considered for the project is Cryptix [Cryptix 2005] which claimed to provide good support for PGP but was deactivated on 2005.

The default configuration employs HmacSHA256 and AES256 as keyed hashing and symmetric encryption functions respectively. These standards are approved by NIST (National Institute of Standards and Technology) and the standard used is configurable by changing the encryption configuration properties settings, provided the bouncy castle library supports them.

Other NIST approved keyed hash functions are SHA-224, SHA-384, SHA-512, SHA-512/224 and SHA-512/256 and the approved encryption standards are Triple DES and Skipjack [NIPS 2015; NIST 2014].

### 5.1.4 Searcher

Searcher follows the same operations performed by the indexer for each word in message but on the search word and invokes Data Manager to get the index and then for each message index it decrypts the message id and uses the decrypted message id to check for the membership of the word in that Bloom filter message index.

### 5.1.5 Data Manager

Data Manager provides connection to the database for storing and retrieving indices. This project uses JDBC (Java Data Base Connectivity) driver with the configurable connection parameters to allow runtime portability across different database types. Data Manager is a singleton class, providing a single connection to the database while it could be extended to manage a pool of connections for improving performance. Initially this project used JavaDB or Apache Derby [Oracle 2015d] as a local file database due to the straight forward support from within Java. But this project noticed a considerable delay in database operations. Hence this project moved to use



HSQLDB [HyperSQL 2014] as a local file database, as it promised to be another light weight database written in Java guaranteeing comparatively increased performance.

## 5.2 Integration Implementation

This section describes the integration details with Columba email client for a live integration and integration with Maildir for offline testing. Both the integrations accepts default configuration parameters provided as part of the library.

### 5.2.1 Columba Client Integration

This section begins with the indexing integration details followed by the searcher and text viewer details. For integration, the Secure Searcher library eclipse project is added as a depended project to the Columba email client project [Dietz et al. 2013].

#### 5.2.1.1 Indexing Integration

As described in the [integration design](#) section the implementation includes two indexing support.

The first or full time indexing implementation involved creating a Utility Menu ‘Index All Encrypted Messages...’ and displaying an Index dialog corresponding to the implementation file ‘*org.columba.mail.gui.action.IndexMailDialog*’. The implementation is similar to the existing functionality provided by Columba for exporting email messages. In a nut shell, instead of exporting email with an *ExportFolderCommand* implementation, Index email support invokes the indexing command ‘*org.columba.mail.folder.command.IndexFolderCommand*’ where each email message is retrieved and *EmailClientIndexer* of the Secure Searcher is invoked to index the message. Before starting the indexing process, the data store is cleared as it is a full index operation.

For incremental indexing, initially this project tried to add ‘*IMailCheckingListener*’ to check for new messages. But the notification of new messages with this listener was inconsistent; moreover there was no notification for deleted messages. So this project added a listener on Inbox folder implementing the Columba ‘*IFolderListener*’ interface called ‘*org.maynooth.indexer.integration.MailIndexerListener*’ as it provided consistent notification for both new and deleted messages.

#### 5.2.1.2 Searcher Integration

For searcher integration, a new search type ‘Encrypted Body Contains’ is added to the search filter combo menu in ‘*org.columba.mail.gui.filtertoolbar.FilterToolbar*’. And the ‘*DefaultSearchEngine*’ is modified to invoke the new filter plug-in for the encrypted message search ‘*org.columba.mail.filter.plugins.EncryptedBodyFilter*’.

The encrypted body filter gets the filter search pattern, the source folder (Inbox) and message id as input. And it returns a Boolean value indicating whether the search pattern is found in the message body of the message invoking ‘*EmailSearcherClient*’. It is possible to extend it to a more efficient implementation that invokes ‘*EmailSearcherClient*’ once to get all message ids, instead of invoking the Searcher for each message id. This project decided to follow the former approach for a standard conformance with other filter plug-ins developed within Columba.

#### 5.2.1.2.1 Challenges in Searcher Integration

Columba has two search facilities. One is a quick search that was integrated with this project and another one is a detailed search. The detailed search supports conjunctions ‘And’, ‘Or’ among searches on multiple search fields such as subject, to and cc fields of the email message. This project initially attempted to integrate with detailed search but the work flow of detailed search could not be easily identified and it repeatedly crossed over with the quick search implementation. This project then chose to integrate with quick search and the detailed search integration is moved as future work.

#### 5.2.1.3 Text Viewer

Columba does not have an inbuilt support to decrypt the encrypted message. So this project modified the ‘*org.columba.mail.gui.message.viewer.TextViewer*’ to display the encrypted message. This involved identifying the encrypted message body and invoking the ‘*Decrypter*’ implementation to get the decrypted contents. This integration demonstrates an additional capability supported by the implemented library.

### 5.2.2 Maildir Integration

This section describes the implementation details of the integration with Maildir for offline testing. At first, the mail storage format details are described followed by the mail server and client simulator details.

#### 5.2.2.1 Mail Storage

This project uses Maildir [Qmail 2015] as an offline storage mechanism where each email message is represented as a separate Maildir file. The JavaMaildir [Zhukov 2002] library is used for parsing and reading the messages.

Initially, this project investigated IMAP (Internet Message Access Protocol) [Crispin 2003] and Gmail Rest APIs [Google 2015] for retrieving email messages for indexing. But then, it decided to use an offline mechanism to enable better support for the development environment. Initial storage format used was Mbox [Hall 2005] where all messages are stored in a single Mbox file. Gmail supports exporting email messages in Mbox format and the file is parsed using Mstor Java library [Fortuna 2014]. But the major issue during development phase was the complexity associated with modifying the original Mbox file for adding or deleting messages. So this project moved to Maildir format where adding/removing messages would imply adding or removing (renaming) the corresponding Maildir files in the Maildir source folder.

#### 5.2.2.2 Mail Server Simulation

For offline simulation, this project created an email client implementation in the ‘*org.maynooth.mailreader.client*’ source package. The client registers two listeners with the server and is used to listen for new and deleted messages. Server uses a polling mechanism to poll periodically for the message changes (addition/deletion) and notifies the registered client listeners.

A major challenge in the simulation was that JavaMaildir [Zhukov 2002] library used for parsing Maildir files had full support for UNIX format files and it had issues in Windows development environment used for the project. For example, for deletion of messages JavaMaildir renames the corresponding file with “:2ST” at the end indicating the message has been seen and trashed. But

Windows environment does not allow the special character colon “:” in a file name. So this project used a work around of creating a new folder (deletedmessages) whose new messages were interpreted as deleted messages.

This integration could also act as a core for full pledged offline indexer and searcher that allows searching encrypted email messages stored or exported in Mbox or Maildir format. In view of the future extensions, the initial Mbox reader implementation is retained in the project even though they are not used for the project’s offline testing.

### **5.3 Encrypted Email Generator**

This section details the encrypted email generator implemented for the performance testing of the library. The generator generates encrypted email from an input plain text data; this includes OpenPGP and S/MIME email generation.

The implementations belong to the package ‘*org.maynooth.mailgenerator*’. These two were the challenging parts next to the decryption implementation discussed earlier. While S/MIME encryption was straightforward from S/MIME examples, the corresponding decryption has to be modified so that it handles the S/MIME generated email messages as well. Writing OpenPGP encrypted email message was about finding a right OpenPGP encryption example [Archive 2015]. The final example was not easily searchable for reference and the examples that were tried initially [Wiki ServiceNow 2012a; Fastpicket 2012; Sloanseaman 2012] were incomplete or using old version of Bouncy Castle APIs.

## 6 Secure Searcher

This chapter describes the final functionalities exposed by the project. Due to the performance loss described in the next section, the symmetric encryption/decryption utility functionality is turned off in the final implementation. Other functionalities are exposed as per the design and implementation chapters. At first, details of the APIs (Application Programming Interfaces) exposed by the implemented library are described, followed by the functionalities that were assessed using a live integration with the Columba email client.

### 6.1 Implementation Library

This section details the APIs exposed by the implemented library for consumption by external email clients. At first, two major APIs (Indexer and Searcher) are outlined which are followed by Key manager and Decrypter APIs.

#### 6.1.1 Indexer

The indexing APIs are exposed as part of the `'org.maynooth.client.indexer.EmailIndexerClient'` interface implementation. The major APIs are listed as follows and the Java docs contain the detailed documentation of the APIs.

```
void addEncryptedMessagesToIndex(Iterator<Message> messages)
```

```
void addPlainMessagesToIndex(Iterator<Message> messages)
```

```
void deleteMessagesFromIndex(Iterator<Message> messages)
```

```
boolean isMessageIndexed(Message message)
```

The message objects accepted as part of the APIs are of type `javax.mail.Message` [Oracle 2013a]. The interface can be instantiated with the corresponding `'EmailIndexerClientImpl'` implementation for usage.

The implemented functionality delegates the responsibility of possible exceptions while parsing message or using cryptographic APIs to the email clients. Therefore the email client is responsible for notifying users or taking appropriate roll back options; this strategy is used for all other exposed functionalities.

In addition to the above APIs, similar APIs that accepts an additional message id parameter is introduced to support email client integration. This is due to the possibility that the email clients may store messages using their custom message Id instead of the message id header as part of the email message. For example, Columba email client employed numbers for message ids while the message id that is part of an email message needs to conform to a recommended format with ankle brackets [Crocker 1982].

#### 6.1.2 Searcher

The Searcher APIs are exposed as part of `'org.maynooth.client.searcher.EmailSearcherClient'` interface implementation. The major APIs are listed as follows and the Java docs contain the detailed documentation of the APIs.

*List<byte[]> searchAllMessages(String searchWord)*

*List<byte[]> searchAllMessagesForAnded(List<String> searchWordList)*

*List<byte[]> searchAllMessagesForOred(List<String> searchWordList)*

The result values are a list of message ids returned as a byte array for security and neutral character encoding implementation. Similar to Indexer, Searcher contains additional APIs with message id as an additional parameter to search within a specific message and they return a Boolean value indicating the presence or absence of the searched word. Searcher can be instantiated with the '*EmailSearcherClientImpl*' implementation for usage.

### 6.1.3 Key Manager

The major key manager APIs exposed as part of the package '*org.maynooth.client.keystore*' for external email clients are listed as follows.

*KeyStoreManager.createKeyStore(char[] password)*

*KeyStoreManager.generateSessionKey(char[] keyStorePassword)*

*KeyStoreManager.deleteSessionKey()*

Above APIs use the key store configuration property setting to get the key store file location. In addition to the above APIs there are overloaded APIs that accepts key store file path and password as additional parameters.

Email client can generate the key for session; this is the indexing key needed for indexing or searching using Secure Searcher. Email client is responsible for maintaining the session time during which the key is stored in memory of the indexer. After the session time out, email client may delete the key. Secure Searcher uses lazy loading of the key and the key is loaded during the first actual search or index operation.

If the integrated email client needs Secure Searcher to index OpenPGP or S/MIME encrypted messages then the corresponding keys needs to be loaded into the key store. Email client needs to load the PGP and/or S/MIME keys before the indexing process so that the Secure Searcher could decrypt the messages for indexing.

*PGPKeyManager.loadPGPKeyToKeyStore(String pgpFile, char[] pgpPassword)*

*SmimeKeyManager.loadPfxfileToKeystore(String pfxFile, char[] pfxPassword)*

Secure Searcher uses *IndexKeyAlias* and *SmimeAlias* key store configuration settings for loading the appropriate key from the key store. For OpenPGP decryption, the public key Id part of the encrypted data is used to search for the corresponding key in the key store. This enables multiple PGP keys and one S/MIME key per user as the S/MIME certificate is typically allocated to the user (email address) by CA (Certificate Authority).

### 6.1.4 Decrypter

The major decrypter APIs exposed as part of the '*org.maynooth.client.indexer.decrypter*' package that are usable by an external email client are listed below. These can be used for decrypting the message (Java mail) or decrypting the content.

`byte[] DecrypterManager.getDecrypterChain().decrypt(Message message)`

The above API decrypts the input message and returns the decrypted message body.

`byte[] decryptBytes(byte[] encryptedBytes)`

The above API exposed in the *OpenPGPDecrypter* and *SmimeDecrypter* can be used by the email client as part of its implementation to decrypt the respective encrypted contents.

## 6.2 Columba Search Functionality

This section details the search functionality implemented as part of the integration with Columba email client. It begins with the indexing functionality followed by searching and then concludes with the text viewer functionality.

### 6.2.1 Indexing

To perform initial indexing of already existing email messages, a utility menu is created which indexes all existing encrypted emails. This option would clear the index store and load new indices to the data store. The figure below shows the full index dialog box where the folder to be indexed needs to be selected.

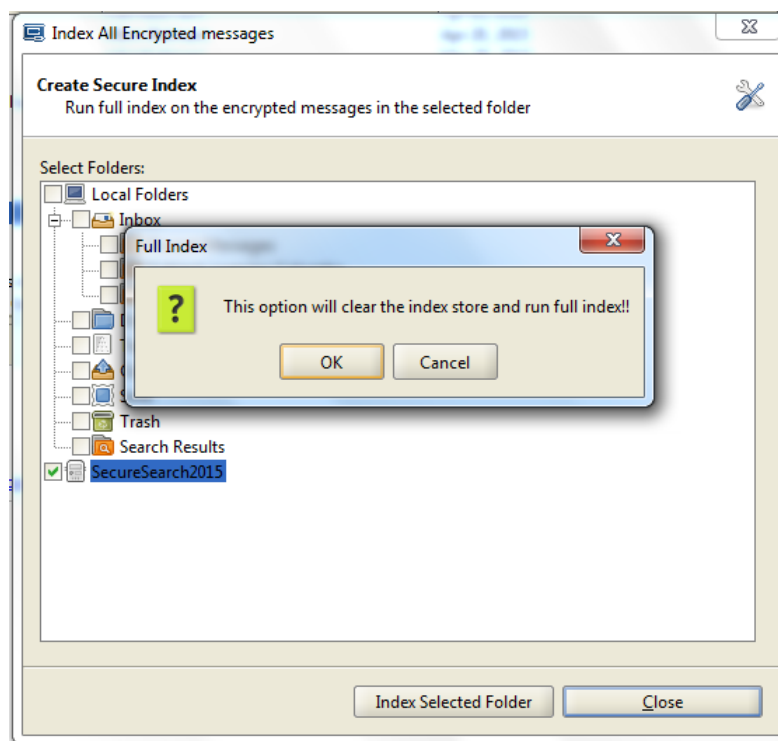


Figure 6-1 Columba Indexing Dialog

Whenever a new message arrives at the Columba client or if a message is deleted in the Columba client, the incremental indexer would automatically perform the corresponding index or remove operation at the backend.

## 6.2.2 Searching

To illustrate the search functionality, consider an S/MIME encrypted message sent to the setup email account ([SecureSearch2015@gmail.com](mailto:SecureSearch2015@gmail.com)). The figure below shows the Gmail web viewer which does not recognise the encrypted email and is shown as an attachment smime.p7m.

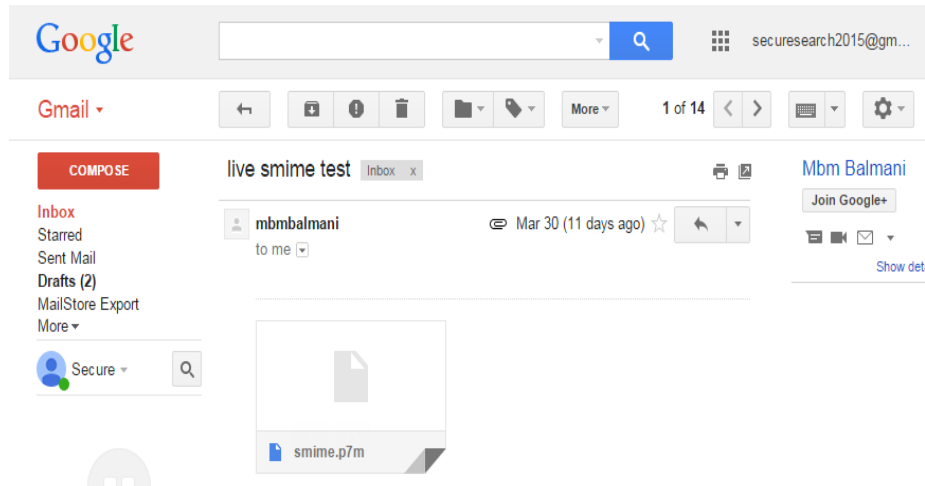


Figure 6-2 Gmail S/MIME message

The same message is received using the integrated Columba email client and the encrypted message body was searched for the word in the message body 'livesmime'. The figure below shows the search result indicating the corresponding encrypted message. The same functionality is implemented for OpenPGP encrypted messages as well.

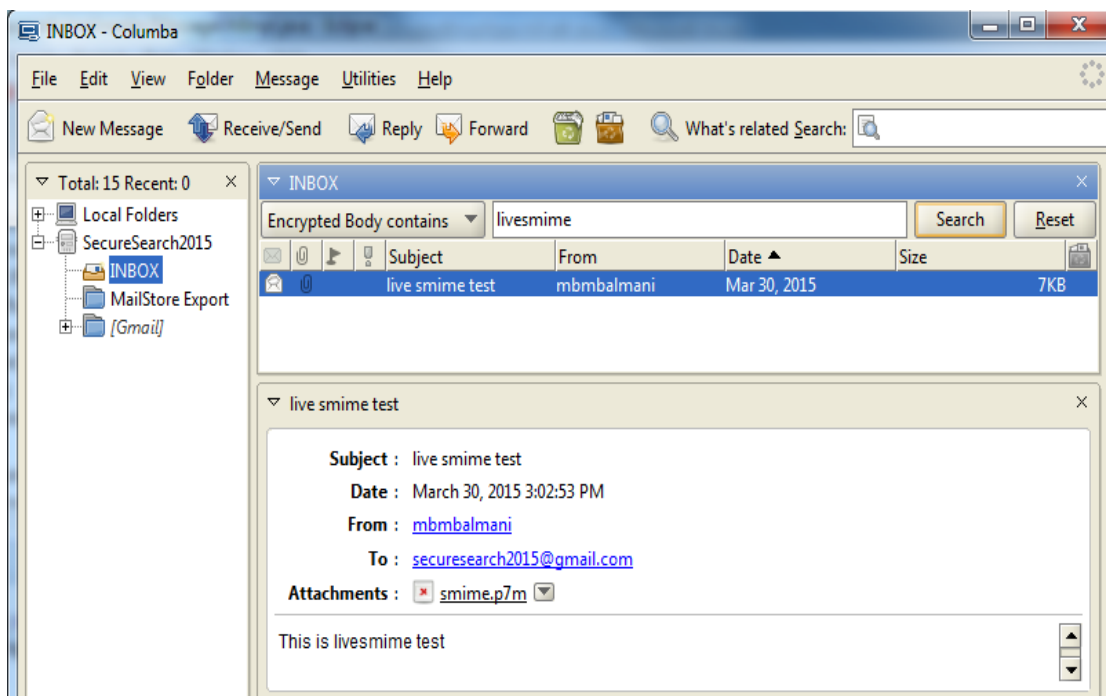


Figure 6-3 Columba S/MIME Message



### 6.2.3 Text Viewer

Columba does not have a default support for viewing encrypted messages. The Decrypter APIs exposed by the library is used to view the encrypted message content in Columba's text viewer. The previous figure shows the corresponding functionality where an S/MIME encrypted message is viewed in the Columba's text viewer.

To illustrate OpenPGP decryption functionality, consider an OpenPGP encrypted message sent to the setup email account ([SecureSearch2015@gmail.com](mailto:SecureSearch2015@gmail.com)). The figure below shows the Gmail web viewer with the encrypted message content.

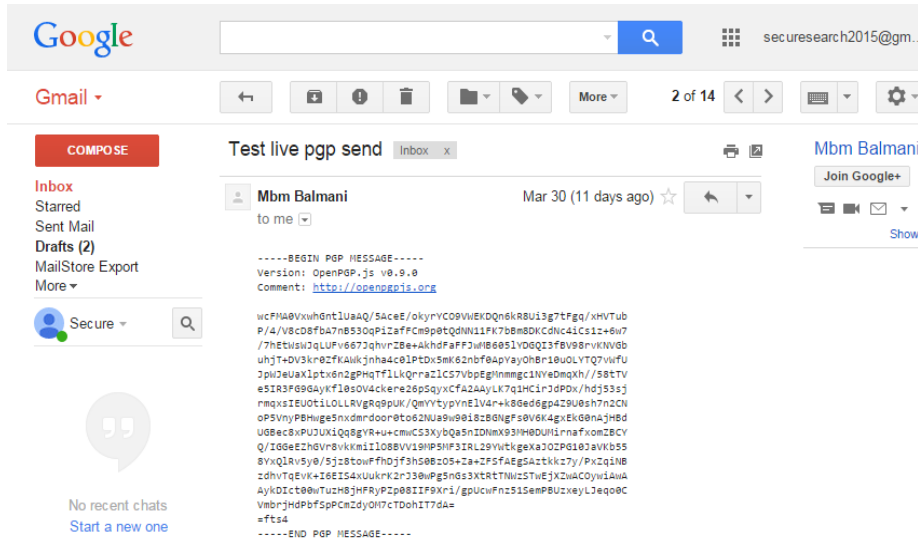


Figure 6-4 Gmail OpenPGP Message

The same email when viewed using the Columba client displays the decrypted message content as depicted in the next figure.

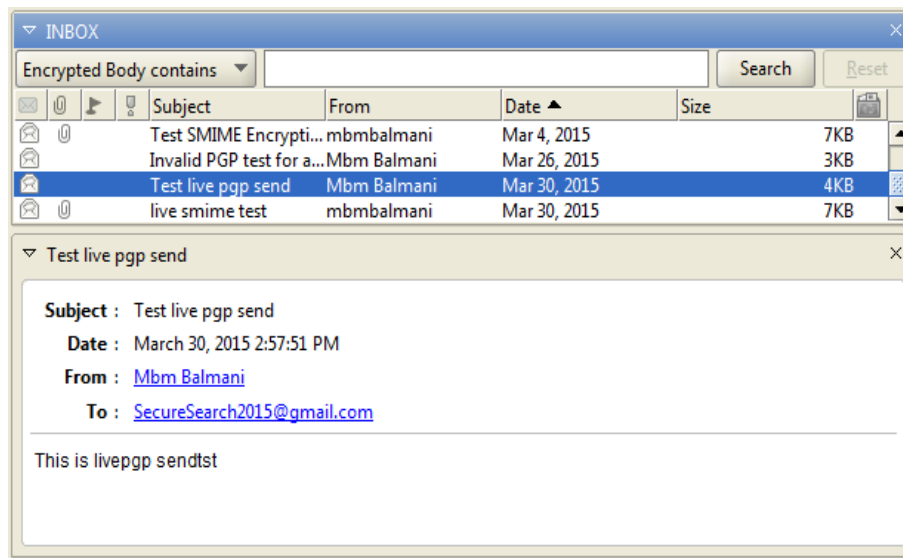


Figure 6-5 Columba OpenPGP Message



## 6.2.4 Key store Configuration

The key store configuration dialog enables importing PGP and S/MIME private keys to the key store. These keys are used for decrypting the encrypted messages for indexing. By default S/MIME key is used as the indexing key, this setting along with other default configurations can be modified in the indexConfiguration.xml at Columba home directory.

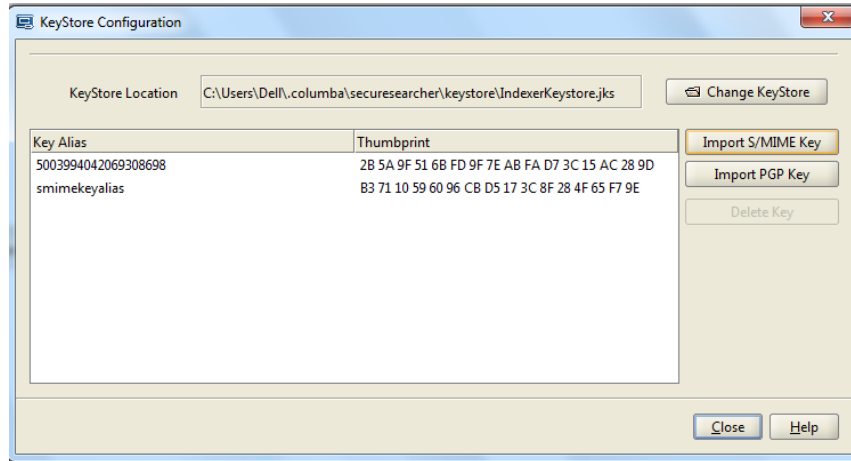


Figure 6-6 Columba KeyStore Configuration

## 7 Evaluation

This chapter outlines how the developed system satisfies the objectives and conforms to the strategies that were initially adopted. And then it critically analyses the functionality, performance and security of the implementation.

### 7.1 Objectives

This project answers the core research question regarding the feasibility of securely searching encrypted email messages with different design approaches and corresponding security tradeoffs in the [Design](#) chapter. A Java library is developed which exposes APIs (Application Programming Interfaces) for securely searching and indexing encrypted as well as plain text messages. The implemented library is then integrated with Columba, an open source Java email client and the secure search functionality is realised using the implemented library.

This project proposes a design for securely searching encrypted email messages with Goh's Bloom filter technique at its core to index the email messages. A Java library is developed using Java Cryptographic library and Bouncy Castle Cryptographic library for indexing and searching OpenPGP encrypted messages, thus realising the primary objective of the project.

The implemented library is extended to support S/MIME encrypted email messages to realise the secondary objectives. The responsibility of differentiating whether the indexing process is a first time indexing of all old email messages or an incremental indexing of new messages is delegated to the invoker of the designed library by design. But as part of the realisation of the tertiary objective to integrate the library with an email client, the integration with the email client was seamless for both first time and incremental indexing. Thus the project realises both the primary and secondary objectives along with the first part of tertiary objectives.

The secondary part of the tertiary objective to deploy a remote data server that synchronises with the local index data store is not demonstrated due to time constraints. This would involve utilising a data base replication and synchronisation technique which is typically used in applications where data resides on cloud storage.

### 7.2 Development Strategy

The selection of Goh's Bloom filter as a core technique for Secure Searcher satisfies the initial technique selection strategy. The implemented library asserts the feasibility of the implementation. The size of the created index is compact with each word being indexed uses 8 bits and this could be further reduced depending on the false positive rate tolerance. Thus the selection satisfies the first two major strategies for technique selection. The security and performance of the technique is analysed as part of the respective sections in this chapter.

This project strictly adhered with the initial development strategy of incremental development where a core implementation is extended to realise additional objectives. The implementation is extensible with the layered architecture asserting each layer is suitable for future extensions. Moreover, the integrated Columba email client follows plug-in architecture where the integrated search functionality can be extended further.

The technique and the cryptographic libraries are selected after an initial prototype and an investigation in terms of search support for issues faced during prototyping. Mail dir is chosen as

a mail format for offline testing as it allows seamless addition and deletion of messages. Furthermore, all existing encrypted email messages can be exported as ‘.eml’ (Electronic Mail file) or Mail dir format and can be indexed and securely searched with the implemented library without any integration to the email client. This would need additional UI on top of the existing Mail dir server implementation.

### 7.3 Testing Strategy

JUnit tests are written to test the functional modules in each layer including the final exposed APIs. The effectiveness of the written tests is measured with EclEmma [Hoffmann et al. 2015] a code coverage analysis tool. The figure below shows that the tests written cover 86.7% of the implemented library. This is well above the adopted testing strategy to cover 80% of the source code. Apart from some exceptional cases, the code to handle S/MIME decryption for various mail message formats as part of *mailreader* affected the coverage.













Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
src	 86.7 %	4,765	729	5,494
org.maynooth.server.searcher	 88.5 %	1,155	150	1,305
org.maynooth.mailreader	 39.0 %	89	139	228
org.maynooth.client.keystore	 83.9 %	675	130	805
org.maynooth.client.indexer.decrypter	 75.5 %	394	128	522
org.maynooth.datastore	 88.1 %	653	88	741
org.maynooth.encryptionUtil	 88.7 %	290	37	327
org.maynooth.client.searcher	 95.2 %	550	28	578
org.maynooth.client.indexer	 97.3 %	585	16	601
org.maynooth.client.indexer.bloomFilter	 96.9 %	217	7	224
org.maynooth.configuration	 93.8 %	45	3	48
org.maynooth.server.indexer	 97.4 %	112	3	115

Figure 7-1 JUnit Test Coverage Report

The suitability of the developed library for its use with a live email client is assessed with its integration to the Columba email client. Additional scenario testing performed as part of the evaluation of operations in the user manual complements the existing JUnit tests. Testing the capability of Bouncy Castle library to decrypt OpenPGP and S/MIME encrypted messages is out of the scope of this project. But the scenario testing used Mailvelope and Enigmail for sending OpenPGP encrypted messages whereas Microsoft Outlook email client is used to send S/MIME encrypted messages. Detailed list of tests summary is available as part of the appendix [A Testing Summary](#).

### 7.4 Functionality

This section discusses the functionalities exposed as part of the implemented library and with the integration to the Columba email client. It concludes with a critical analysis of missing functionalities.

#### 7.4.1 Implementation Library

This section evaluates the functionality or APIs exposed as part of the implementation library which includes both the indexing and searching functionalities.

### **7.4.1.1 Indexing**

Indexing APIs provide functionality to index both encrypted (OpenPGP and S/MIME) and plain text messages. In addition, an API to check whether the given message is already indexed is exposed. The APIs exposed accepts an Iterator [Oracle 2015a] of Java mail Message [Oracle 2013a] instances allowing multiple email messages to be sent to the indexer. It is trivial for an email client to convert a message body to Java mail Message; the encrypted email generation functionality implemented as part of the [secure email generator](#) could also be used as a reference.

As part of the integration with Columba, additional APIs that accepts a separate message id parameter was introduced. This is due to the fact that the Message Id header values as part of a Java mail Message needs to be of particular format within ankle brackets [Crocker 1982]. But, the email clients use a local message id typically starting with a number 1 for caching the messages locally and it is nontrivial to change the Message Id header of an existing Java mail Message instance [StackOverflow 2013]. So the additional APIs enable seamless integration with email client.

### **7.4.1.2 Searching**

For searching, the implementation exposes APIs to perform case insensitive search for a word in all the messages or in a particular message. Searcher configuration settings limit the number of search results returned by the searcher. The next batch search functionality where the searcher searches in the next batch after the previous search results assert the suitability in a typical email client search, where the results are displayed in batches of 20 in a page.

Moreover, Searcher exposes APIs for a compound search where multiple words must be present in the message ('AND') or one of the words must be present in the message ('OR'). These two APIs can be used to extend the search functionality to provide a more sophisticated search where some words may be present, other words must be present and other combinations.

## **7.4.2 Integrated Email Client**

Email client integration shows both the indexing and search capability of the secure searcher. Moreover, the functionality exposed by Key Manager APIs for importing keys is also integrated to the client. A full key manager UI implementation to export or view the keys in the key store or import keys sent as part of the email is considered out of the scope of this project.

## **7.4.3 Critical Functionality Analysis**

This section analyses the missing functionalities and discusses the feasibility of implementing such functionality and tradeoffs associated with them.

### **7.4.3.1 Searching Subject or Attachments**

The Secure Searcher indexes only the message body of the email messages. It is trivial to integrate the functionality to search subject or attachments in the message with the addition of additional Bloom filter indices or adding to data to a single index. It is not implemented as the end-to-end email encryption tools do not encrypt the email message subject. This is due to the fact that the subject of the message is parsed by the spam detection services in the mail servers. Moreover, the email encryption tools do not support encryption of attachments and even a typical email client do not offer search support on attachments. Hence searching attachments (encrypted or plain) is considered out of scope of this project.

#### 7.4.3.2 Context information

Context sensitive search refers to searching for a word with respect to the context of another word i.e., a 'SearchWord1' occurs before or after a 'SearchWord2'. This functionality requires identifying the location of the searched word in the email message as its core.

Implementing this would require adding the location information to the word before mapping it to the Bloom filter index. For example, instead of mapping a 'word', 'wordL' is mapped to the Bloom filter where L is the location in the message with N words [1-N]. But during the search process, the searcher has to search for all locations [1-N] in the message, making the search complexity polynomial  $O(N*M)$ , where N is the average number of words in a message and M is the total number of messages indexed.

Another way would be to store all the words in the message and compare against each word as in [homomorphic encryption](#) with the search capability. This would also require polynomial comparisons and it may not be possible with the investigated techniques that have linear search time as the implemented one.

#### 7.4.3.3 Number of occurrences

This functionality refers to searching in terms of number of occurrences of search word i.e., the message contains 'SearchWord' N times. The existing indexing process indexes the repeated occurrence information as per the proposed [design modifications](#). During the search process, search queries on the indices of the search results needs to be implemented to sort based on repeated occurrence information. Implementing this functionality is considered as a future enhancement.

#### 7.4.3.4 Stemming

The current implementation supports only exact word searching i.e., if the indexed word is 'stemming', search for the word 'stem' or 'stems' or 'stemmer' or 'stemmed' would not return the index. The two possible approaches to implement this functionality are outlined below.

During indexing, all the stemmed words could be indexed as well. In addition to increase in the indexing time, this would require increase in the Bloom filter index size or increase in the false positive rate.

An alternative would be to index using the default approach and search for all the stemmed words. This would increase the search time depending on number of stemmed words.

#### 7.4.3.5 Technique Complexity

Indexer needs to scan the email at least once to index it, while the current technique scans the input words twice for determining the total number of words in the message and then indexing the words.

The technique used needs to process N records for searching N indexed messages demanding  $O(N)$  search complexity. The index record for each message needs to be stored separately to guarantee the security against frequency analysis attack described later in this chapter.

If the security threat model do not require protection against frequency analysis attack then the search time could be made close to  $O(\log N)$  using the current technique. As described as an

efficient search mechanism in Goh's Bloom filter technique [Goh 2004], for indexing each word, hash that word (trap door) but skip the next additional step where the hashed word is again used as the key to hash the message id. The trap door is then mapped to the Bloom filter index. Since the word mappings are no longer unique to each message, a Bloom filter tree could be created by computing bitwise OR of the two Bloom filter sets. Thus a non-existent word may be ruled out by searching against the root index in the Bloom filter tree. The cost of searching and indexing using the technique are analysed further in the Goh's report which are both in  $O(\log N)$ .

Moreover, the [inverted index technique](#) could even scale the complexity close to  $O(1)$ . But this project considers frequency analysis attack as a basic security attack especially if the created index is stored on un-trusted storage.

#### **7.4.3.6 Lost Indexing Key**

As with any encryption requirements, the key used for indexing the encrypted email messages needs to be stored securely. Moreover, similar to the fact that the encrypted email messages could not be decrypted if the corresponding decryption key is lost, the created secure index is not usable if the corresponding indexing key is lost.

#### **7.4.3.7 False Positive Rate**

This project's default Bloom filter configuration employs 8 bits per word in the message body with 6 hash functions for mapping each word to the index. These parameters are described as ones that produce an optimal false positive probability of 0.0215 [Fan et al. 2000] asserting the selection as default parameters. Evaluating this theoretical false positive rate in the Secure Searcher implementation is considered out of scope of the project.

It is possible to further reduce the false positive rate by increasing the number of bits per word configuration, requiring a space efficiency trade off. This is configurable with the help of configuration property settings in the implemented library. This project considers 0.0215 error rate as acceptable as this is non-critical search functionality. Moreover, the end user could view the email message and see that the word is not present in the email.

#### **7.4.3.8 Maximum word limit**

Maximum number of words that a message can contain so that it could be indexed by the current technique is represented below

Maximum number of words =  $(\text{java.lang.Integer.MAX\_VALUE} / \text{BitsPerWord configuration})$

With the default configuration the above expression calculates to 26,84,35,456 maximum allowable words per message. Note that this is the supported maximum number of words by the implemented library, while the database Bloom filter index column would also need to be expanded to hold the large index. The default database configuration as part of the integration implementation supports a maximum limit of 32,768 words in a message. This project views this as an acceptable limit for the body of email messages used in typical communication.

## 7.5 Performance

This section analyses the run time of secure searcher for indexing and searching test email messages generated using [Encrypted email generator](#). The environment used for the analysis followed by indexing and searching time analyses is described in the following sections.

### 7.5.1 Analysis Environment Setup

Runtime analysis is carried on a Dell Inspiron 1525 Laptop with 2 GHz Intel Core 2 Duo T7250 processor and 4GB Transcend DDR2 SDRAM. The indexing process is tested from an Eclipse Luna development environment installed on Windows 7 64 bit operating system.

Initially this project considered using the spam or advertisement promotion email messages as the test source. But these email messages mostly contain images while this project requires a text based content for analysis. This project uses luindex [Blackburn et al. 2006] text data comprising 1230 text documents utilised by DaCapo benchmark suite for benchmarking Apache Lucene [Apache Software Foundation 2012], an open source indexer and searcher for text documents. The text documents are converted to Open PGP, S/MIME and Plain messages using Encrypted email generator and the resulting 3690 email messages are used for analysing indexing and searching time.

### 7.5.2 Indexing Analysis

This analysis focuses on the following two analysis questions

- What is the time taken to Index a message based on the number of words in a message?
- How does the OpenPGP or S/MIME encryption affects the message indexing time?

Plain, OpenPGP and S/MIME messages are indexed separately and the indexing process is repeated 5 times with the median value being plotted in the graphs below. The detailed values are included as part of the *maildirtestresources/evaluationresults* directory for further reference.

The analysis graph shows the indexing time of approximately 1 second for indexing 3000 words and it increases with increase in the number of words with a maximum value of 5.5 seconds for indexing 32000 words in an OpenPGP encrypted message. Compared to plain messages, S/MIME and OpenPGP messages require additional time at the beginning for decryption process owing to key retrieval process. The following observations could be made from the corresponding analysis graph.

Interestingly the indexing time for S/MIME and plain messages remain almost same while OpenPGP messages take comparatively more time. S/MIME decryption implementation uses predefined 'SmimeKeyAlias' to identify the private key for decryption of S/MIME messages rendering the key lookup process negligible. OpenPGP decryption implementation scans the message payload to identify the encrypted public key id and requests the key store for respective OpenPGP private key. This project examined this additional scanning to be one of the causes for the slight increase in decryption time.



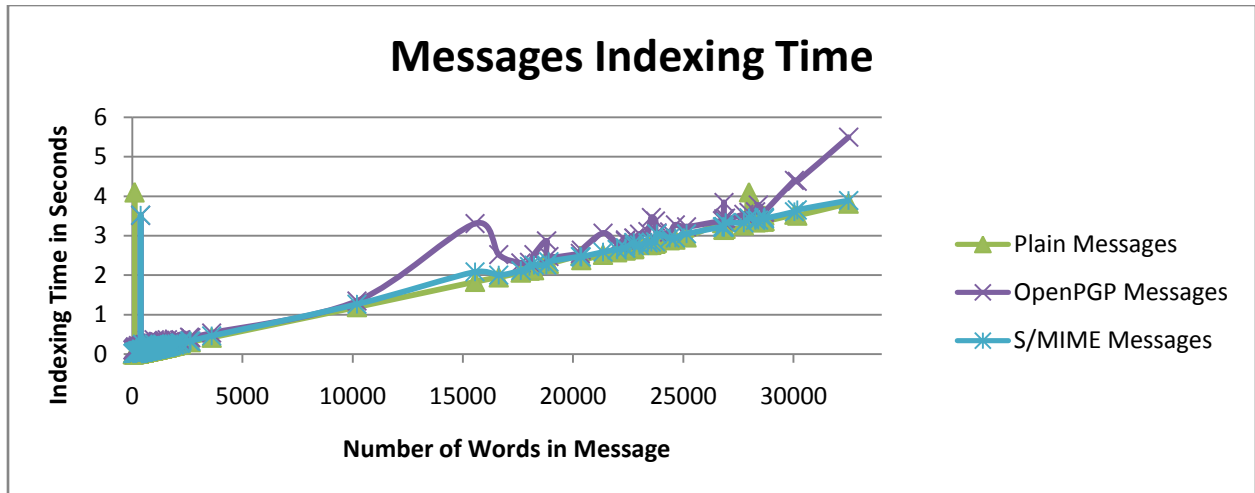


Figure 7-2 Messages Indexing Time vs Number of Messages

The project’s implementation could be modified to decrease the OpenPGP key look up process with a preconfigured key alias like ‘OpenPGPKeyAlias’. But this project adheres to the default implementation as S/MIME keys are unique certificates issued by Certificate Authority to a person (email address). Hence an email address typically has one S/MIME key associated with it, whereas a user could have multiple PGP keys employed for different purposes and the current implementation supports storing multiple PGP keys per user.

Another important observation is the initial time of up to 4 seconds for indexing only 100 words approximately. This is the first email message considered for indexing during which the SecureSearcher initialises encryption services and pseudo key generator. This delay could be negated by performing the initialisation as part of a backend process after the client is loaded. Moreover, the indexing of email messages is typically handled by a backend process in the email client. Hence this project does not analyse further to improve the complexity and performance of the indexing process.

### 7.5.3 Searching Analysis

This analysis focuses on the analysis question of measuring the search time against the number of index records to be processed. The results from five searches with different search words are used for the analysis.

This analysis does not consider the length of the search word as any search word is hashed into 256 bit hashed word (Trap door) for further searching. In addition, success or failure of the presence of a search word in the message accounts for a negligible processing time difference during search operation. This due to the fact that the additional processing involved is only on checking the additional indices in the Bloom filter for set bits.

#### 7.5.3.1 Search time with Message Id Encryption

The graph presented next indicates the initial time of 8 seconds for searching 20 index records, which increases up to 30 seconds for searching 100 index records and requires around 6 minutes for searching 1260 index records.



Search time for the implementation is viewed unacceptable for practical purposes. On further investigation this project deduced that the decryption of message id during the search process of each index record took around 2 seconds. Note that the current implementation uses a single thread, with a single connection to database searching each record iteratively. Each step could be optimised to compensate for the performance loss during searching.

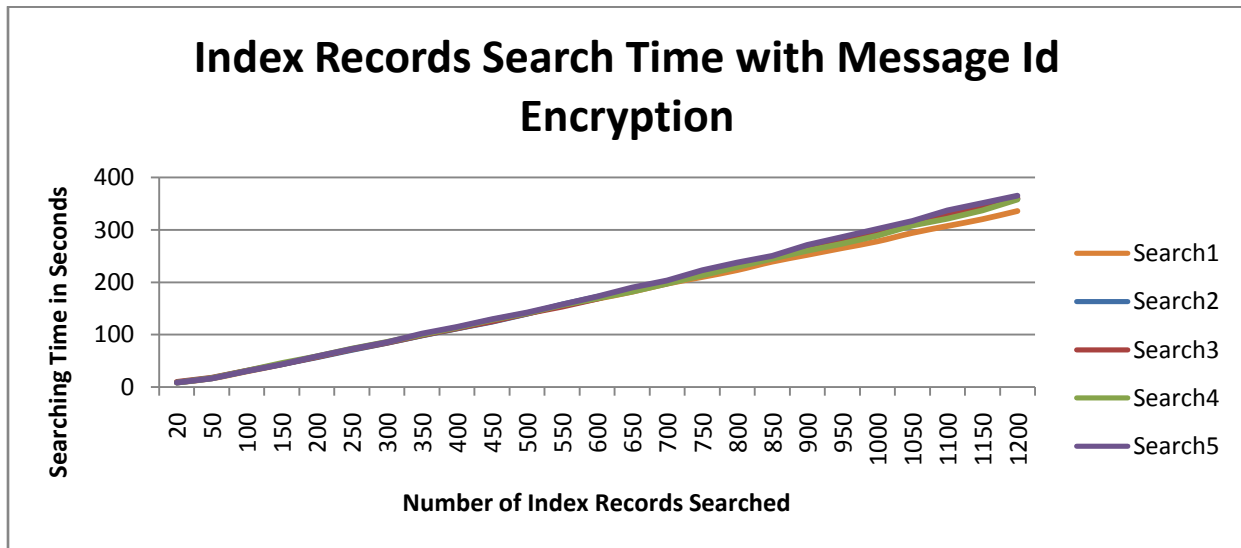


Figure 7-3 Index Records Search time with Message Id Encryption

#### 7.5.3.1.1 Impact on Final Functionality

This project modified the final functionality to use the plain message id, to better accommodate the integrated implementation with the Columba client. The final index stored on the un-trusted storage is still secure as per the original Goh’s Bloom filter technique. But the attacker model scope is modified, which is discussed in the next [security evaluation](#) section. More importantly, this project considers no information is leaked based on the plain message id information stored as part of the index. After the implementation of performance improvement changes, the encryption of message id could be turned on by changing flags in the *SymmetricEncryptionUtil* implementation.

An alternative secure approach would be to store only the plain message id instead of encrypted message id. But instead of hashing the message id with trap door (initial word hashed with key) as the key, the message id could be hashed with the indexing key used for trap door. And then the hashed message id could be hashed with the trap door as the key before mapping the word to the Bloom filter. This guarantees the same attacker model as the initial design proposed by the project, by replacing decryption process during search process of record with a hashing process. This would be a good alternative and needs to be analysed further for the time taken to search with a less costly hashing process instead of decryption.

#### 7.5.3.2 Final Search time without Message Id Encryption

This section analyses the modified final functionality without decryption or encryption of message id. As depicted in the analysis graph below, the search time is within 3.5 seconds for searching 3690 index records. The initial 2 second delay is due to the index key generation and

the trap door generation for the search word. This project considers the final delays as practical and acceptable as there is more scope for further performance improvements.

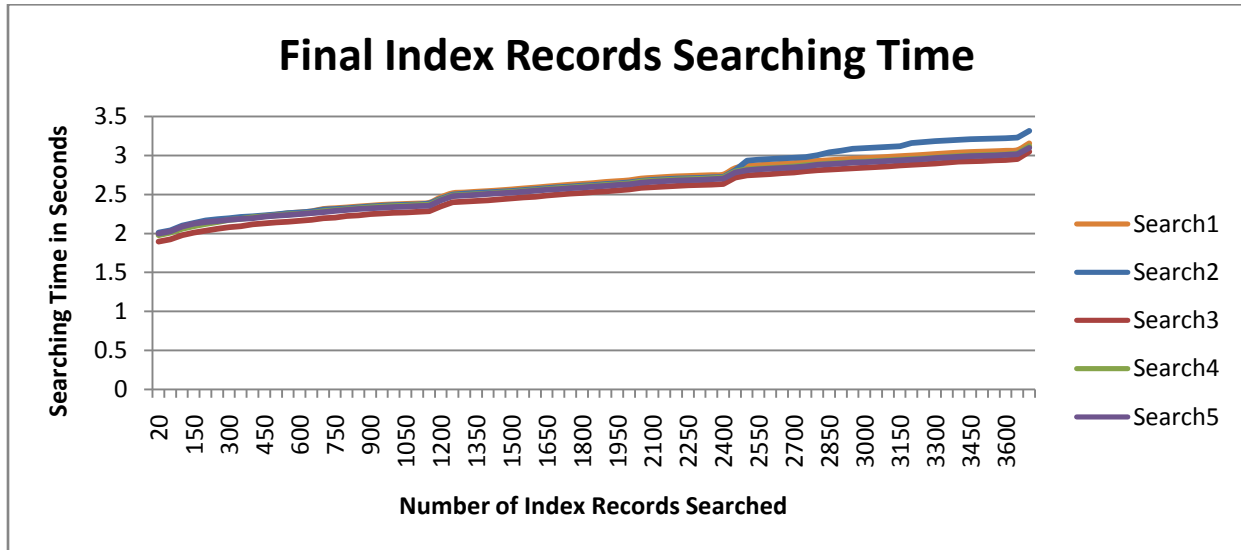


Figure 7-4 Final Index Records Search Time vs Number of Index Records

### 7.5.3.3 Search time against Bloom filter index size

This analysis focuses on the analysis question to determine if the search time depends on the length of the index record being processed. The initial spikes with maximum of 80 milliseconds are owing to the generation of pseudo key and the trap door word generation during each search. In summary, as depicted in the graph the search time increases with the length of the index record but the increase is negligible from 0 milliseconds to 4 milliseconds for lengths from 1000 to 250000 Bloom filter size index record.

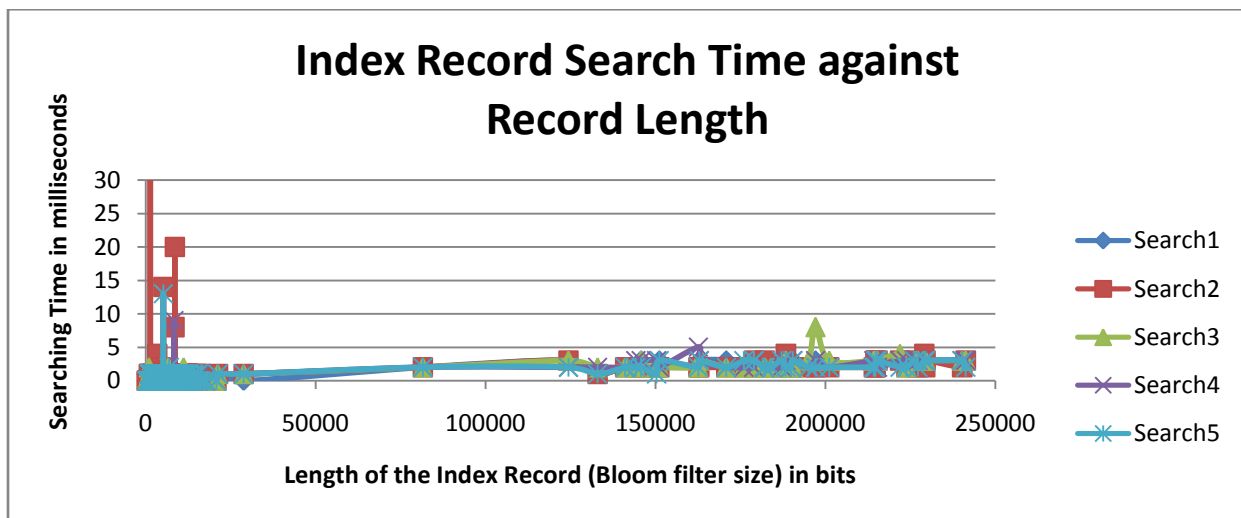


Figure 7-5 Index Record Search Time vs Record Length

## 7.6 Security

This section describes the threat model that the index stands against on un-trusted storage. Then it describes the enhanced attacker model with the proposed design alternative which needed further performance improvements. Finally the attacker model that this design does not withstand is described at the end.

### 7.6.1 Security model of the Final implementation

This section describes the attacker model that the final implementation protects against.

#### 7.6.1.1 IND-CKA Semantic Security

The final implementation adheres to Goh's original secure index approach; hence it has IND-CKA (INDistinguishability against Chosen Keyword Attack) semantic security. The security is explained with a challenger and indexer response scenario as follows.

The challenger inputs the indexer with two identical length messages; the indexer returns two indices corresponding to the two messages as a response. The challenger would not be able to distinguish which index belongs to which message, no different than the probability of  $\frac{1}{2}$  (choice of picking one at random). Hence the index does not reveal the contents of the messages that are indexed. The implementation assumes no information is leaked from the message id information.

#### 7.6.1.2 Controlled Searching

To perform search operation, a trap door of the search word needs to be created using the indexing key. Only the user with access to the indexing key could perform the search on the index.

#### 7.6.1.3 Frequency Analysis attack

During the indexing process, hashing each unique message id again with the trap door word as the key renders the mapping unique to that index. I.e., if two messages have identical contents, then the resulting index would be different owing to the unique message id.

Moreover, the number of words mapped on to the Bloom filter index depends on the number of words in the message. Even if the message with N words contains only 1 unique word repeated N times. After mapping the unique word, N-1 words with repeated information i.e., word2, word3... wordN-1 are mapped to Bloom filter index as per the proposed modification.

#### 7.6.1.4 Brute force Attack with Known Plain Text

The trap door restricts an attacker from searching against the stored index i.e., only the user with access to the indexing key can generate the trap door. It does not prevent the attack against the original hashing technique used to generate the trap door. Because an attacker with a known plain message would be able to perform a brute force hashing of a known word and try different keys to search against the stored index.

The HMAC [Krawczyk et al. 1997] function is strong against the known plain text attack, with the default configuration (HmacSHA256) requiring attacker to try  $2^{128}$  key values. The recommended modification of encrypting the message id further strengthen the index against brute force attack as the attacker has to perform an additional brute force decryption of the message id on each index record.

## 7.6.2 Enhanced Security Model

This section describes how the current implementation suffers from the mentioned vulnerabilities and how the [proposed design modification](#) withstands them.

### 7.6.2.1 Replay Attack

If the attacker gets hold of a trap door, then the attacker could use the same trap door to repeatedly perform search on the index. This would be a serious security flaw as it would divulge the presence of words in the new indexed messages as well.

The additional encryption of message id by the proposed design modification strengthens the index against replay attack. The replay attack may not be possible as the attacker with the trap door word has to first decrypt the message id to search the index. But the cost of this secure modification is the time required to decrypt each message id during the search process.

## 7.6.3 Vulnerability Model

This section describes the threat model that both the final implementation and the enhanced security model could not withstand.

### 7.6.3.1 Key store

To begin with, the information regarding the length of the message indexed is leaked from the size of the index. Moreover, as with any security model, this model relies heavily on the secure storage of the indexing key. To reiterate, the default implementation provides an option to store keys in a password protected Java Key Store (JKS).

### 7.6.3.2 In-Memory attack

The design does not protect against an In-Memory attack where the attacker could view the In-Memory data during indexing or searching process. As the indexing keys and decrypted messages are available In-Memory, they are vulnerable to attack. But even the end-to-end encryption tools suffer from this attack as the input password for the key needs to be loaded in the memory for authentication. The tools typically create a security sandbox model over the application process to prevent such attacks. Preventing this attack is considered out of scope of this project as per the adopted strategy.

### 7.6.3.3 Confidentiality Integrity Availability (CIA)

Furthermore, if the index is stored on un-trusted external system then the current design of secure index ensures only the confidentiality of the email messages. The un-trusted system should ensure the availability and integrity of the stored index. While there is no design modification to ensure the availability of the index, this project sees the following option to enforce integrity. Each Bloom filter index and message id data could be combined and a signature hashing algorithm could be used to add signature as another column. This would ensure authenticity and integrity of the data. As checking for integrity during each search operation would increase the search time, integrity check could be carried out on demand as a back ground task.

## 7.7 Summary

This project satisfies the initial objectives and conforms to the strategies adopted initially. The code coverage analysis asserts the effectiveness of the JUnit tests. Moreover, this project evaluated the performance of the implementation in terms of length of the messages indexed.

The existing implementation is modified to compensate for the performance loss during the search time evaluation. In addition, the integration of the implemented library with an open source email client asserts the suitability of using the library in a real environment.

## 8 Conclusions and Future Work

This chapter provides a summary of achievements and the challenges faced during the course of the project. It concludes with a list of possible improvements and future works.

### 8.1 Conclusions

This project addresses the initial research question regarding the feasibility of securely searching encrypted email. Different secure search techniques and deployment architectures are discussed for its feasibility and security. This project recommends a deployment architecture having a cryptographically strong Goh's Bloom filter secure index technique at its core.

The core objectives are realised with an implementation library supporting indexing and searching of plain, OpenPGP and S/MIME encrypted email messages. This project proposes modifications to Goh's secure index technique in a view to enhance the security and usability of the library. The implemented modifications are selectively enabled depending on the assessment of performance in the previous chapter. Furthermore, this library is integrated with Columba, an open source Java email client to demonstrate the usability of the library in a live environment.

This project adhered closely to the initial adopted strategies for design, development and evaluation. Finally the [Evaluation](#) chapter provides a critical analysis on the missing functionalities, performance and security. These analyses include the feasibility of incorporating additional functionalities to the existing library and tradeoffs associated with the same. Furthermore, the developed library adopted the layered architecture to support future extensions. Finally the search functionality in the integrated email client can be extended further owing to the plug-in architecture of the email client.

### 8.2 Summary of Challenges

This section summarises the challenges faced during the different phases of this project.

Initial research phase involved the challenges in understanding the existing cryptographic [secure search techniques](#). The next challenge was in the design phase to identify a practical [deployment architecture](#) that enables the use of the adopted technique in a live environment.

The development challenge involved adopting an extensible design and [decrypting S/MIME](#) encrypted messages where `message.getContent()` did not result in expected encrypted message body. Hence, this project adopted a work around to get raw message stream and parse it manually. For the integration with email client, the architecture and usability of the existing email clients including [Pooka](#) and [Columba](#) needed to be analysed for extensibility. For the Columba client with plug-in architecture, the main challenge was to identify source code work flow and the extension points for integration.

Finally for evaluation, a sufficient source of encrypted messages needed to be identified. Initial choice of spam and promotion messages were rejected as they typically contain images and [Dacapo benchmark source](#) was finally selected. This text document source was then converted to encrypt email messages with the help of an [encrypted email generator](#) implementation. Moreover, this project moved from Mbox to Maildir format for offline email storage owing to the good support for adding or removing messages.

### 8.3 Future Works and Improvements

In addition to the [functionalities](#) that were analysed critically for incorporation with the implementation in the previous chapter, this project views the following as possible future extensions.

The current implementation searches index based on the indexed order. An extension could be to add functionality to search by ordered time or last accessed time or depending on the priority or the location folder of the message. Moreover, [the number of occurrences](#) of the searched word in a message could be used for sorting the search results as the repeated information is already indexed during the index operation.

Furthermore, the current implementation indexes email messages belonging to one email account in one index store. But a typical user has multiple accounts for different purposes such as university, work place and personal. Since a user would typically use same end-to-end encryption key for all the accounts, indexing all these emails in a single repository store would be an interesting approach.

Maildir client simulator could be further extended with a UI support enabling indexing and securely searching encrypted email messages offline. This would require exporting the email messages from email account in .eml or MailDir format and using the client simulator for searching the exported messages.

This project's approach for secure searcher library was to add indexing capability at the back end server or thick client. An alternative would be an implementation as part of a thin client browser UI using Java script cryptographic APIs. This would be an interesting design allowing extensive usage with typical web email interfaces.

The solution proposed in this project shows the feasibility of a secure search implementation for encrypted email messages. The adoption of the implemented library or the proposed design would enhance the usability of end-to-end encryption tools with the secure search functionality.

## A Testing Summary

JUnit tests were written for testing the functional modules written as part of the implementation library. The tests are located inside the ‘tests’ folder of the secure searcher project. The summary of tests written and their mapping to the source code are outlined as follows.

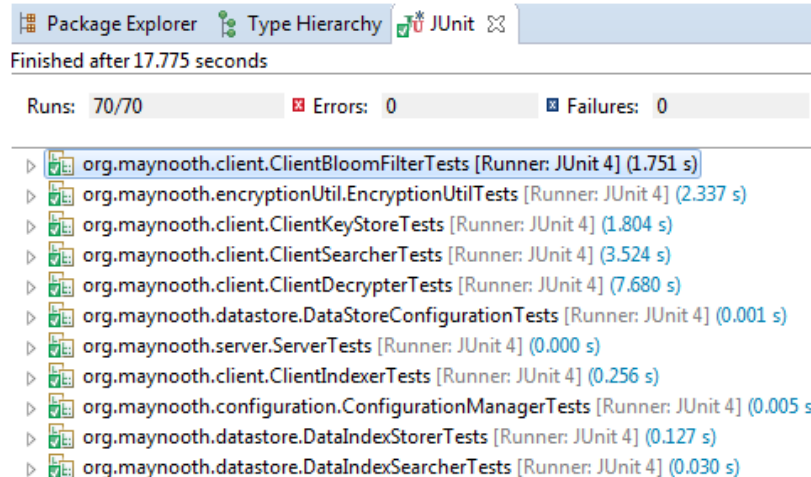


Figure A-1 JUnit Test Execution Report

### A.1 Client Tests

Tests written as part of ‘*org.maynooth.client*’ package includes indexer, key store and searcher client tests testing the APIs exposed as part of the implementation library.

Moreover, the package includes Bloom filter tests testing the Bloom filter index creation module and Decrypter tests testing the decrypter chain for testing the decrypter functionality for S/MIME and OpenPGP encrypted messages.

### A.2 Data store Tests

Tests written as part of ‘*org.maynooth.datastore*’ package include tests for DataIndexStorer and DataIndexSearcher that facilitate storing and retrieving the index. In addition, DataStoreConfigurationTests test the default data store configuration parameters.

### A.3 Encryption Utility Tests

Tests written as part of ‘*org.maynooth.encryptionUtil*’ package include tests for default symmetric encryption utilities and HMAC hashing utilities implemented as part of the library.

### A.4 Server Tests

Tests written as part of ‘*org.maynooth.server*’ package include tests for server side indexing and searching APIs.

### A.5 Performance Tests

The performance tests were carried out with the help of MailDirReader implementation contained as part of ‘*org.maynooth.mailreader.client*’ package in the ‘*maildirtest*’ source folder. The resources used by MailDirReader are contained in the ‘*maildirtestresources*’ directory. The



folder names mentioned in the remainder of this section will refer '*maildirtestresources*' as root directory.

The '*maildirectory/cur*' is the source maildir folder looked up by the MailDirReader for current messages. The '*messagecollection*' folder contains the OpenPGP, S/MIME and plain messages generated based on messages from Dacapo benchmark suite contained in 'luindex' folder. These messages were moved to '*cur*' of the maildirectory to measure the performance.

An alternative to MailDirReader is to use MailDirServer as part of '*org.maynooth.mailreader.server*' package which could be used to demonstrate incremental indexing capability by adding new messages to '*maildirectory/new*' folder.

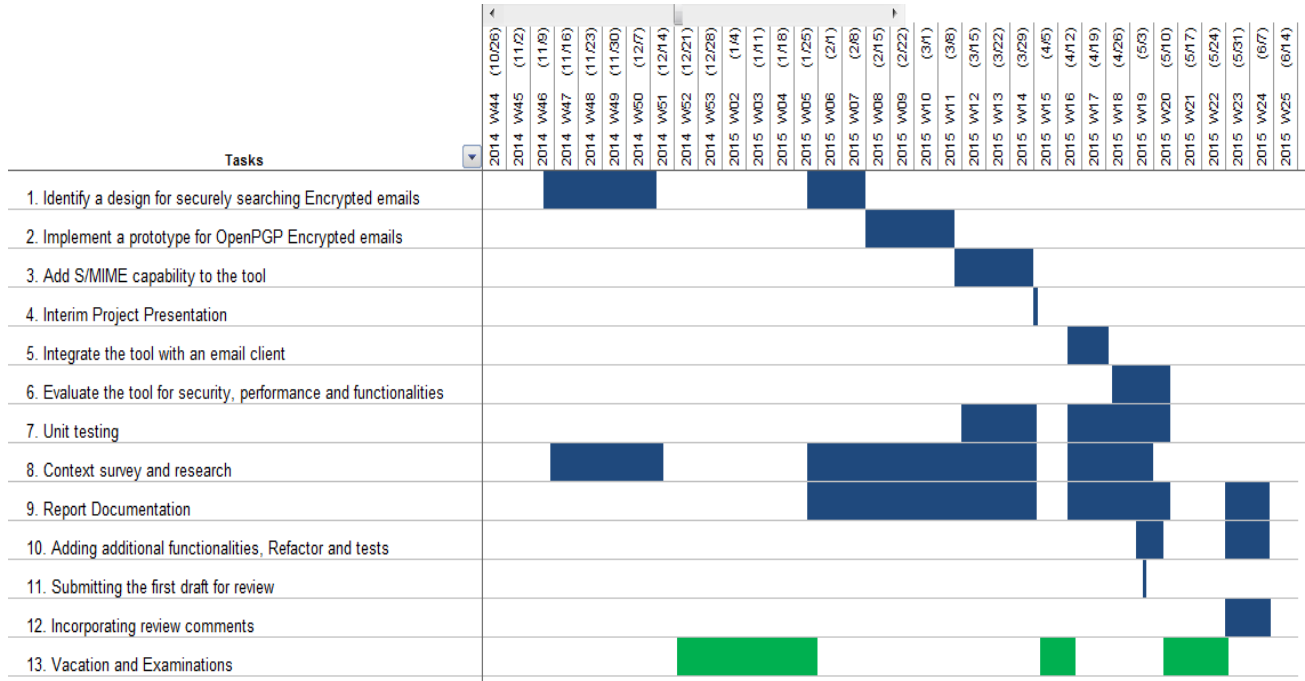
The evaluation logs are part of '*EvaluationResultLogs*' folder contain the logs obtained from each test runs. The final run values used to generate the charts are contained in FinalEvaluationReport.xlsx excel document.

## **B Changes to Original Specification**

The following changes were made to the original objectives after the initial submission of objectives as part of Thesis proposal.

- The initial tertiary objective to integrate with existing end-to-end encryption tools was removed as it was not feasible due to time constraints.
- As an alternative to the above tertiary objective, this project performed integration with a live client with a local index data store to demonstrate the functionalities.
- The secondary part of the tertiary objective to deploy a remote data server that synchronises with the local index data store is removed due to time constraints.
- In addition to the above tertiary objectives, Maildir server simulator was added for offline evaluation. This could be extended further to support offline indexing and searching of email messages.

# C Project Plan



## D User Manual

This section contains the manual for configuring Columba email client with secure searcher functionality. Moreover, this section includes the details to configure demo account SecureSearch2015 for evaluation purpose.

### D.1 Columba Client Configuration

The Secure Searcher with Columba client is available as an executable JAR (ColumbaSecureSearcher.jar) file under deployment directory. It could be run by double clicking the JAR or using the command line option ‘java -jar ColumbaSecureSearcher.jar’. Both these options require a JRE (Java Run time Environment) for execution. This project was tested using JRE versions 7 and 8 and JRE version 8 is recommended for execution.

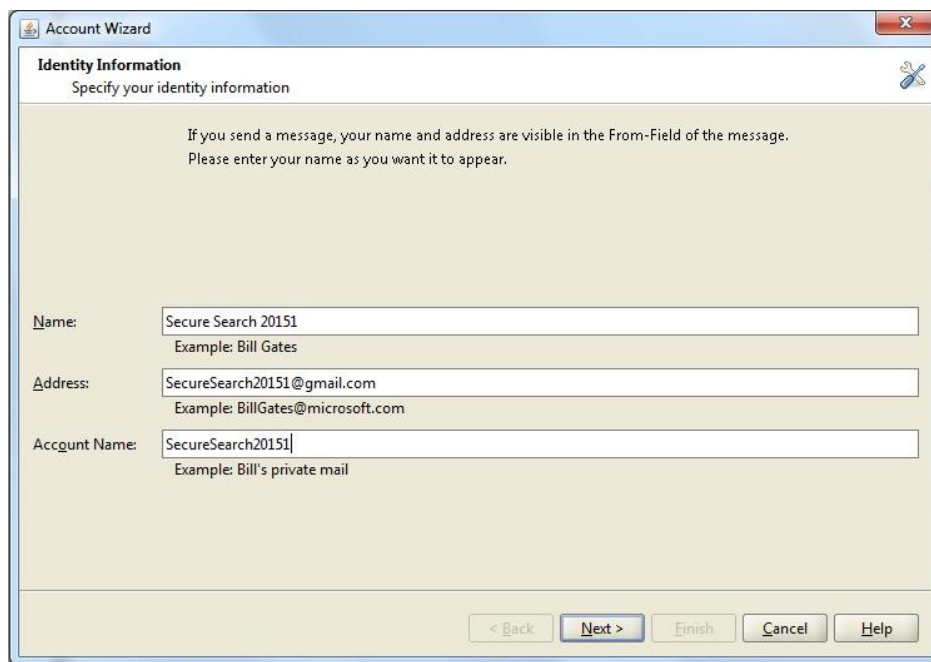
For indexing and decrypting S/MIME messages unlimited strength JCEs (Java Cryptography Extensions) needs to be enabled by replacing some files in the default JRE used for execution. These files are available at Oracle sites [Oracle 2015c; Oracle 2015b]. The files local\_policy.jar and US\_export\_policy.jar inside the <JREHome>\lib\security folder needs to be replaced with the corresponding downloaded files.

For demo or evaluation purpose the user could directly jump to [Section D.3](#). The indexing and searching functionalities are described as part of the [Columba search](#) functionality section.

#### D.1.1 Email Account Configuration

This section contains the details of a typical email client configuration for sending and receiving email messages. The steps followed for configuring the demo account are detailed as follows.

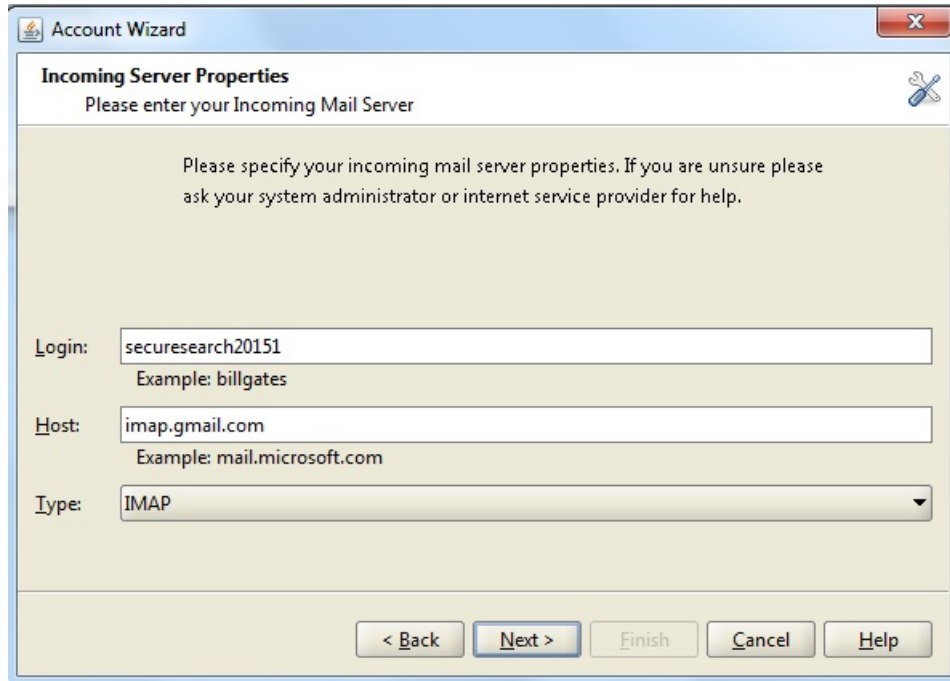
On executing the Columba client for the first time, a new account configuration wizard pops up. Enter the account details as depicted in the figure below.



The screenshot shows a window titled "Account Wizard" with a sub-header "Identity Information" and the instruction "Specify your identity information". A note states: "If you send a message, your name and address are visible in the From-Field of the message. Please enter your name as you want it to appear." Below this are three input fields: "Name" with the value "Secure Search 20151" and example "Example: Bill Gates"; "Address" with the value "SecureSearch20151@gmail.com" and example "Example: BillGates@microsoft.com"; and "Account Name" with the value "SecureSearch20151" and example "Example: Bill's private mail". At the bottom are buttons for "< Back", "Next >", "Finish", "Cancel", and "Help".

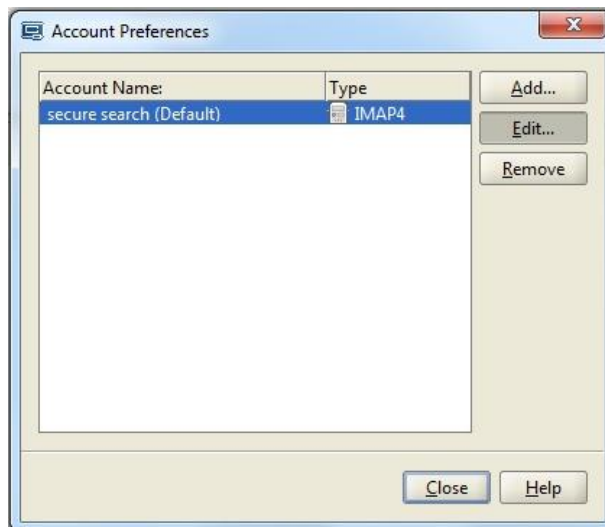
Figure D-1 Columba New Account Wizard

Then enter incoming and outgoing server details for receiving email messages. In the setup the incoming server is an IMAP server account.



**Figure D-2 Columba Server Properties Wizard**

Go to Edit -> Mail Account Settings... Check the input configuration and configure the connection port for IMAP server details.



**Figure D-3 Columba Account Preferences Dialog**

The connection type and the port use for IMAP can be changed for the configured account. This depends on the connected IMAP server details.

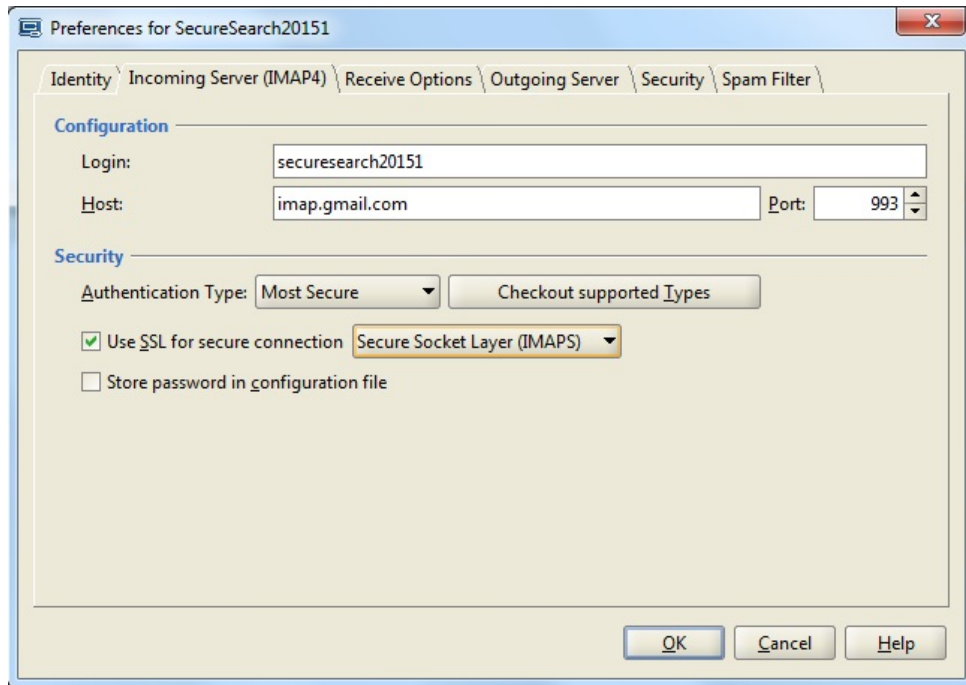


Figure D-4 Columba IMAP Connection Settings

### D.1.2 Enable IMAP on Gmail

If it is a Gmail account, connection to IMAP server for the account needs to be enabled as Gmail uses custom protocol for the connection. This needs to be turned on by following Settings -> Enable IMAP on the Gmail account logged in through a browser.

#### Settings



Figure D-5 Gmail Enable IMAP

Moreover, Gmail considers connections through IMAP server as less secure and therefore less secure connection has to be turned on. This is needed for the connection from Columba to Gmail account and is irrespective of whether Columba has an integrated secure searcher.

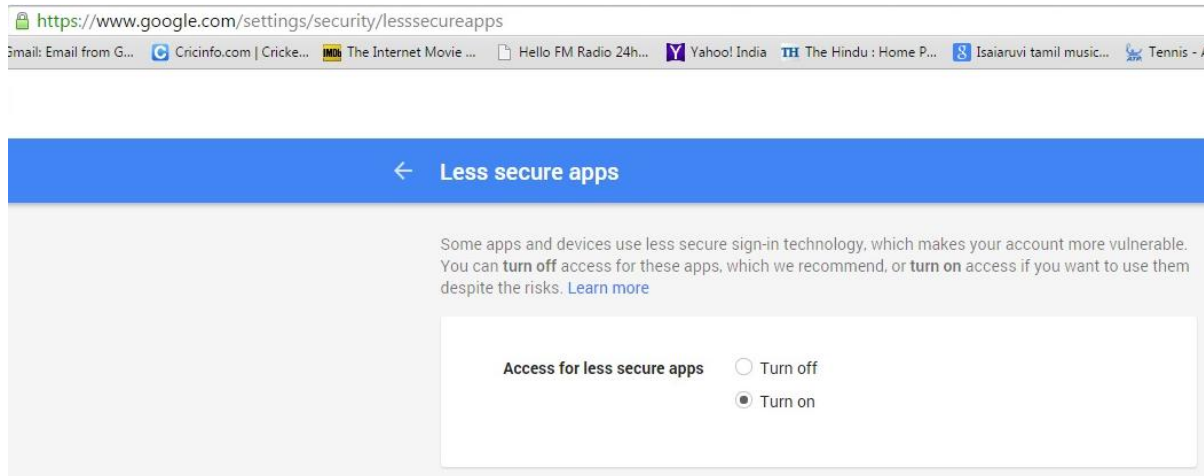


Figure D-6 Gmail Less Secure Access

## D.2 Secure Searcher Configuration

During the first time Columba initialisation, a default key store and HSQL index data store is created in the Columba home directory ' $\langle Home \rangle \backslash .columba$ ' under the subdirectory '*securesearcher*'. For example, ' $C:\Users\langle User \rangle \backslash .columba \backslash securesearcher$ ' is the directory in the Windows environment. All default configurations such as index configuration, key store and index data store can be changed in that directory.

### D.2.1 Key store Configuration

Before starting the indexing process, the PGP and S/MIME keys should be imported to the key store for decryption and indexing.

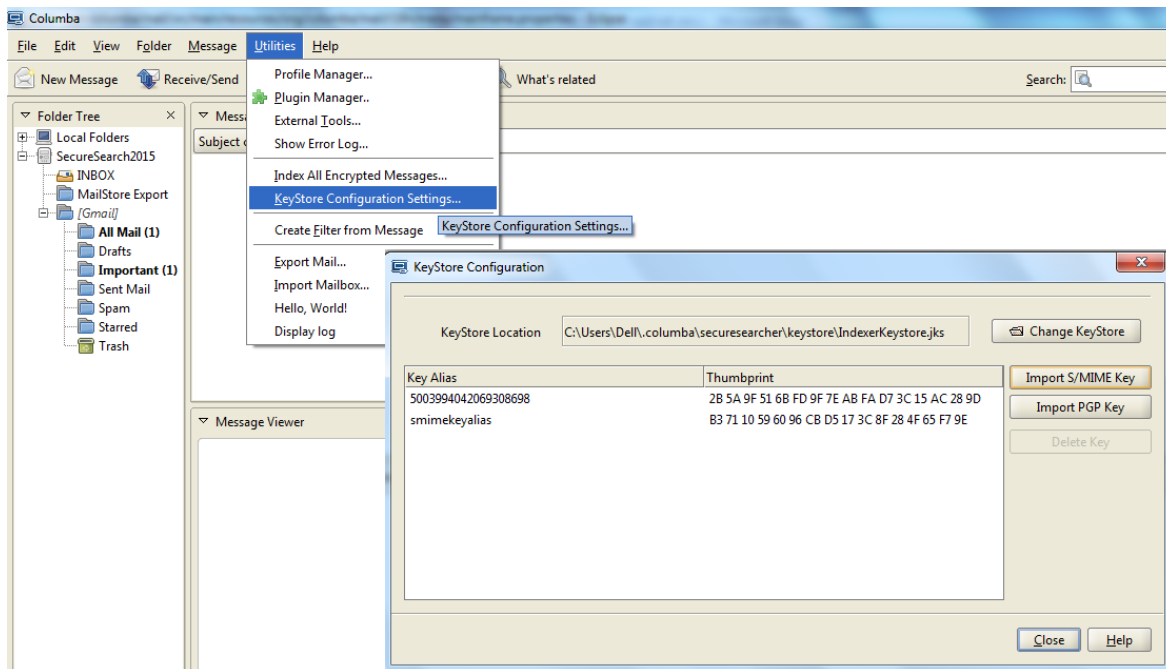


Figure D-7 Columba KeyStore Configuration

As depicted in the figure, go to *'Utilities -> KeyStore Configuration Settings...'* Key Store Configuration dialog allows the user to import one S/MIME private key (.pfx file) and multiple PGP private keys (ASCII format). While importing the key, the user needs to input the password with which the source private key is protected. After the import, all keys would be stored using a single master password of that of the stored key store.

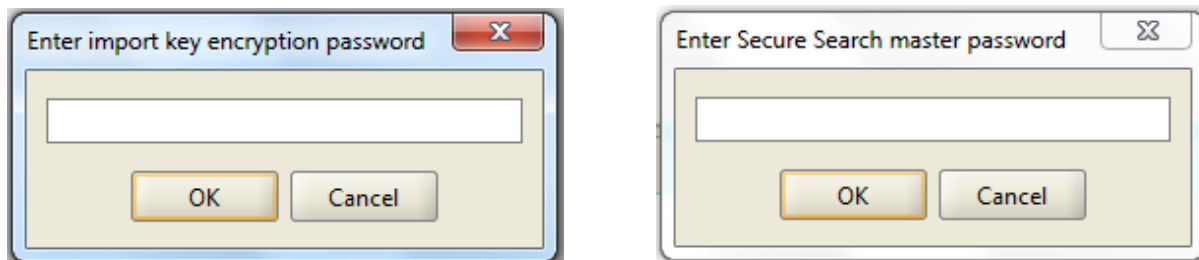


Figure D-8 Enter Import Key and Secure Search Password Dialogs

The default key store password is 'Welcome'. This should be entered as a master password for secure searcher. Moreover, the key store used for secure search can be changed using key store configuration dialog.

### D.2.2 Indexing

Go to *'Utilities -> Index All Encrypted Messages'* to select the Inbox folder on which full index should be run initially. This would clear the index store and create a secure index of all the encrypted messages in that folder.

Columba integration has online indexing feature where any new email messages are automatically indexed and deletion of any messages would result in removal of the corresponding index from the index store.

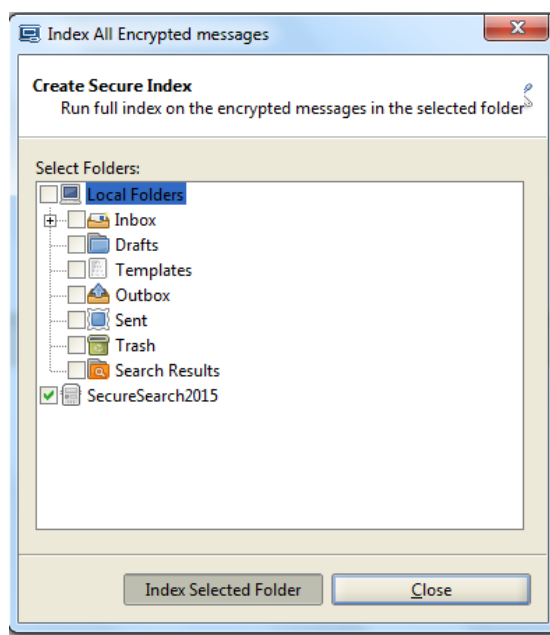


Figure D-9 Columba Index All Encrypted Messages Dialog



### D.2.3 Searching

Select the Search type to 'Encrypted Body Contains', enter the search word and press enter to search indexed email messages.

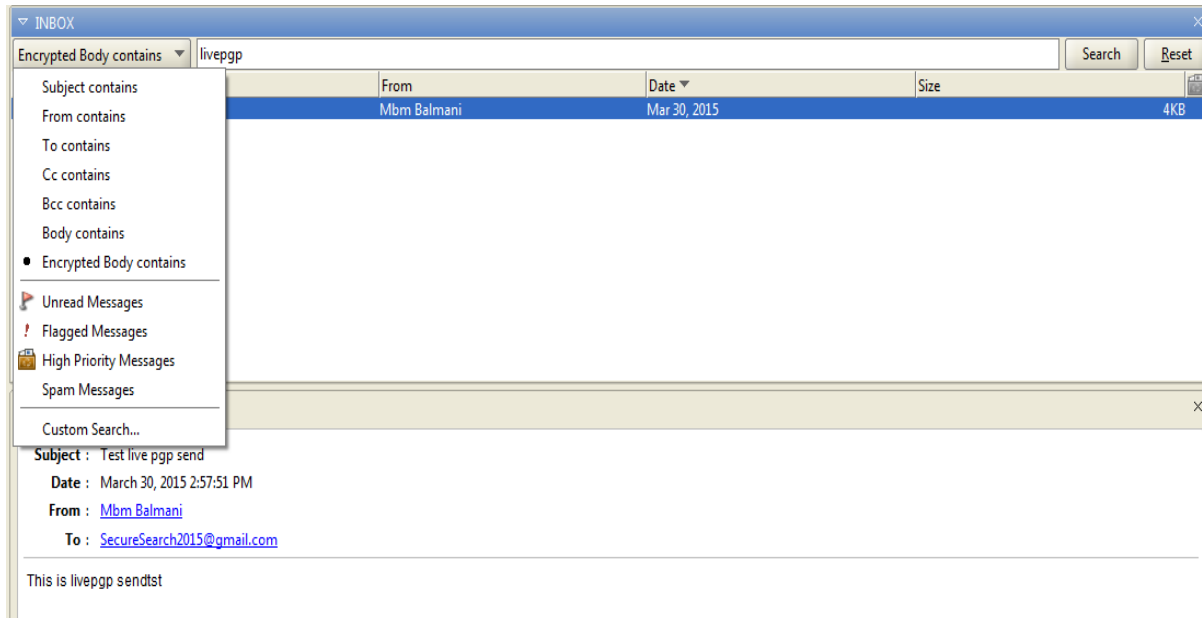


Figure D-10 Columba Encrypted Message Search

## D.3 Demo Account Configuration

To configure the demo account, the demo account home directory (columbademo1 or columbademo2) in the Deployment\DemoConfiguration\DemoAccounts directory needs to be extracted to .columba directory at User home 'C:\Users\\.columba'.

The KeyStoreFile and DatabaseURL location configuration settings in the indexer configuration file at C:\Users\\.columba\securesearcher\indexConfiguration.xml need to be corrected. Change the <User> ('Dell' in demo configuration) to the installed machine user directory.

- KeyStoreFile="C:\Users\\.columba\securesearcher\keystore\IndexerKeystore.jks"
- <DatabaseURL  
DatabaseURL="jdbc:hsqldb:file:C:\Users\\.columba\securesearcher\database\IndexStore;user=test;password=test">

Moreover, the PGP and S/MIME keys used for the demo account are available as part of the deployment directory. Use the default key store password 'Welcome' as the master secure search password.

### D.3.1 Demo Account Configuration Details

#### D.3.1.1 Demo1 Account Details

Username: [SecureSearch2015@gmail.com](mailto:SecureSearch2015@gmail.com)

Password: secure2015

Demo Key store password: Welcome

*D.3.1.1 Demo2 Account Details*

Username: [SecureSearch20151@gmail.com](mailto:SecureSearch20151@gmail.com)

Password: secure2015

Demo Key store password: Welcome

## References

- Adobe Systems Incorporated., 2015. Adobe PDF. Available at: <http://www.adobe.com/products/acrobat/protect-pdf-security-encryption.html> [Accessed November 2, 2015].
- Apache Software Foundation, 2012. Apache Lucene. Available at: <https://lucene.apache.org/core/> [Accessed April 17, 2015].
- Archive, C.D., 2015. Bouncy Castle. Available at: <http://www.bouncycastle.org/devmailarchive/msg09187.html> [Accessed October 4, 2015].
- Atkins, D., Stallings, W. & Zimmermann, P., 1996. *PGP*, Available at: <http://tools.ietf.org/pdf/rfc1991.pdf>.
- Blackburn, S.M. et al., 2006. *The DaCapo Benchmarks : Java Benchmarking Development and Analysis*,
- Bloom, B.H., 1970. *Space/time trade-offs in hash coding with allowable errors*,
- Bouncy Castle, 2015. Smime Decryption Example. Available at: <http://www.docjar.org/docs/api/org/bouncycastle/mail/smime/examples/ReadEncryptedMail.html> [Accessed April 10, 2015].
- Bouncy Castle Inc, 2013. Bouncy Castle. Available at: <http://bouncycastle.org/java.html> [Accessed April 9, 2015].
- Bouncy Castle Inc, 2014. Jce PGP Converter. Available at: <http://grepcode.com/file/repo1.maven.org/maven2/org.bouncycastle/bcpg-jdk14/1.47/org/bouncycastle/openpgp/operator/jcajce/JcaPGPKeyConverter.java> [Accessed April 17, 2015].
- Brunschwig, P., 2015. Enigmail. Available at: <https://www.enigmail.net/home/index.php>.
- Callas, J. et al., 2007. *OpenPGP*, Available at: <http://tools.ietf.org/pdf/rfc4880.pdf> [Accessed February 17, 2015].
- Chang, W.-T., 2005. Thunderbird encrypted search enhancement. Available at: [https://bugzilla.mozilla.org/show\\_bug.cgi?id=280588](https://bugzilla.mozilla.org/show_bug.cgi?id=280588) [Accessed February 18, 2015].
- Chang, Y. & Mitzenmacher, M., 2005. Privacy Preserving Keyword Searches on Remote Encrypted Data. , pp.442–455.
- Comodo CA Limited, 2015. Comodo. Available at: <https://www.comodo.com/home/email-security/free-email-certificate.php> [Accessed February 18, 2015].

- Crispin, M., 2003. *IMAP RFC*,
- Crocker, D.H., 1982. *Standard for the format of ARPA Internet Text Messages*, Available at: <http://tools.ietf.org/pdf/rfc822.pdf>.
- Cryptix, 2005. Cryptix. Available at: <http://www.cryptix.org/> [Accessed April 10, 2015].
- daniele athome et al., 2015. guardianproject GPG. Available at: <https://github.com/guardianproject/gnupg-for-java> [Accessed April 10, 2015].
- Dierks, T. & Rescorla, E., 2008. *The Transport Layer Security (TLS) Protocol*,
- Dietz, F. et al., 2013. Columba. Available at: <http://sourceforge.net/projects/columba/> [Accessed April 6, 2015].
- Einarsson, B., McCarthy, S. & Novak, B., 2015. Mailpile. Available at: <https://www.mailpile.is/> [Accessed February 18, 2015].
- Elkins, M.R. & Blosser, J., 2014. Mutt. Available at: <http://www.mutt.org/> [Accessed February 18, 2015].
- Fan, L. et al., 2000. *Summary cache: A scalable wide-area Web cache sharing protocol*,
- Fastpicket, 2012. Bouncy Castle PGP Encryption 2. Available at: <http://fastpicket.com/blog/2012/05/14/easy-pgp-in-java-bouncy-castle/> [Accessed April 10, 2015].
- Fortuna, B., 2014. Mstor. Available at: <https://github.com/benfortuna/mstor> [Accessed April 10, 2015].
- Free Software Foundation, 2015. GNU. Available at: <https://www.gnu.org/> [Accessed June 8, 2015].
- Freed, N. & Borenstein, N.S., 1996. *MIME*,
- Gilles, 2012. Mutt Search. Available at: <http://unix.stackexchange.com/questions/46580/e-mail-client-in-linux-which-allows-to-search-encrypted-mail> [Accessed February 18, 2015].
- Goh, E., 2004. *Secure Indexes*,
- Google, 2015. Gmail API Java. Available at: <https://developers.google.com/gmail/api/quickstart/quickstart-java> [Accessed April 10, 2015].
- GPGTools, 2015. GPG Tools. Available at: <https://gpgtools.org/> [Accessed February 11, 2015].

Guardian News and Media Limited, 2015. NSA Surveillance. Available at: <http://www.theguardian.com/world/2013/jun/07/nsa-prism-records-surveillance-questions> [Accessed November 2, 2015].

Hall, E., 2005. *Mbox*,

Hoffmann, M.R., Janiczak, B. & Mandrikov, E., 2015. EclEmma. Available at: <http://www.eclEmma.org/> [Accessed April 17, 2015].

Hush Communications Canada Inc., 2015. Hushmail. Available at: <https://www.hushmail.com/> [Accessed February 11, 2015].

HyperSQL, 2014. HSQL DB. Available at: <http://www.hsqldb.org/> [Accessed April 10, 2015].

Internet Mail Consortium, 2015. Internet Mail Consortium. Available at: <http://www.imc.org/smime-pgpmime.html> [Accessed February 17, 2015].

Java-Source, 2015. Java Source. Available at: <http://java-source.net/open-source/mail-clients> [Accessed April 6, 2015].

Kaliski, B., 1998. *PKCS #7: Cryptographic Message Syntax*, Available at: <http://tools.ietf.org/pdf/rfc2315.pdf>.

Koch, W. et al., 2015. GPG. Available at: <https://www.gnupg.org/> [Accessed February 17, 2015].

Krawczyk, H., Bellare, M. & Canetti, R., 1997. *RFC HMAC*, Available at: <http://tools.ietf.org/pdf/rfc2104.pdf>.

MagnusS, 2011. Bloom filter Create hashes. Available at: <https://github.com/MagnusS/Java-BloomFilter/blob/master/src/com/skjegstad/Utils/BloomFilter.java> [Accessed April 9, 2015].

Microsoft, 2015. Outlook. Available at: <http://products.office.com/en-us/outlook/email-and-calendar-software-microsoft-outlook> [Accessed February 11, 2015].

Mozilla, 2015. Thunderbird. Available at: <https://www.mozilla.org/en-US/thunderbird/> [Accessed February 11, 2015].

Mozilla Developer Network, 2015a. JavaXPCOM. Available at: [https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XPCOM/Language\\_bindings/JavaXPCOM](https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XPCOM/Language_bindings/JavaXPCOM) [Accessed April 6, 2015].

Mozilla Developer Network, 2015b. Thunderbird extension Environment. Available at: [https://developer.mozilla.org/en-US/Add-ons/Setting\\_up\\_extension\\_development\\_environment](https://developer.mozilla.org/en-US/Add-ons/Setting_up_extension_development_environment) [Accessed April 6, 2015].

- MozillaZine, 2013. Mozillazine Extension Development. Available at:  
[http://kb.mozillazine.org/Getting\\_started\\_with\\_extension\\_development](http://kb.mozillazine.org/Getting_started_with_extension_development) [Accessed April 6, 2015].
- Nanavati, M. et al., 2014. Cloud security: A Gathering Storm. *Communications of the ACM*, 57, pp.70–79. Available at: [http://dl.acm.org/ft\\_gateway.cfm?id=2593686&type=html](http://dl.acm.org/ft_gateway.cfm?id=2593686&type=html).
- NIPS, 2015. NIPS Encryption Standards. Available at:  
[http://csrc.nist.gov/groups/ST/toolkit/block\\_ciphers.html](http://csrc.nist.gov/groups/ST/toolkit/block_ciphers.html) [Accessed April 10, 2015].
- NIST, 2014. NIST Hash Policy. Available at: <http://csrc.nist.gov/groups/ST/hash/policy.html> [Accessed April 10, 2015].
- Notmuch, 2014. NotMuchMail. Available at: <http://notmuchmail.org/> [Accessed February 18, 2015].
- Oberndörfer, T., 2014. Mailvelope. Available at: <https://www.mailvelope.com/> [Accessed February 11, 2015].
- Oracle, 2015a. Iterator. Available at:  
<https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html> [Accessed April 17, 2015].
- Oracle, 2014. Java Bitset. Available at:  
<https://docs.oracle.com/javase/7/docs/api/java/util/BitSet.html> [Accessed April 17, 2015].
- Oracle, 2015b. Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files 7. Available at: <http://www.oracle.com/technetwork/java/javase/downloads/jce-7-download-432124.html> [Accessed June 4, 2015].
- Oracle, 2015c. Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files 8. Available at: <http://www.oracle.com/technetwork/java/javase/downloads/jce8-download-2133166.html> [Accessed June 4, 2015].
- Oracle, 2015d. Java DB. Available at:  
<http://www.oracle.com/technetwork/java/javadb/overview/index.html> [Accessed April 10, 2015].
- Oracle, 2013a. Java Mail. Available at:  
<https://javamail.java.net/nonav/docs/api/javax/mail/Message.html> [Accessed April 10, 2015].
- Oracle, 2013b. JavaMail. Available at:  
<https://javamail.java.net/nonav/docs/api/javax/mail/Message.html> [Accessed April 17, 2015].

- Pilone, D. et al., 2001. Pooka. Available at: <http://sourceforge.net/p/pooka/wiki/Home/> [Accessed April 6, 2015].
- Qmail, 2015. Maildir. Available at: <http://www.qmail.org/man/man5/maildir.html> [Accessed April 10, 2015].
- Ramsdell, B. & Turner, S., 2010. *Secure/Multipurpose Internet Mail Extensions*, Available at: <http://tools.ietf.org/pdf/rfc5751.pdf>.
- Shmueli, E. et al., *Designing Secure Indexes for Encrypted Databases*, Available at: <http://www.cs.berkeley.edu/~dawnsong/papers/se.pdf>.
- Sloanseaman, 2012. Bouncy Castle PGP Encryption 3. Available at: <http://sloanseaman.com/wordpress/2012/05/13/revisited-pgp-encryptiondecryption-in-java/> [Accessed April 10, 2015].
- StackOverflow, 2013. MimeMessage MessageId. Available at: <http://stackoverflow.com/questions/17818501/set-messageid-in-header-before-sending-mail> [Accessed April 17, 2015].
- StarCom Ltd., 2011. StartSSL. Available at: <https://www.startssl.com/> [Accessed February 18, 2015].
- Thian, L. et al., 2005. *Efficient Search on Encrypted Data*, Available at: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1635501>.
- Torto, D. Del et al., 2015. *MIME Security with OpenPGP*, Available at: <https://tools.ietf.org/html/rfc3156>.
- Trancecrypt Inc., 2014. Neomail. Available at: <https://www.neomailbox.com/services/secure-email> [Accessed November 2, 2015].
- Wiki ServiceNow, 2012a. Bouncy Castle PGP Encryption 1. Available at: [http://wiki.servicenow.com/index.php?title=Sample\\_Java\\_BouncyCastle\\_Algorithm\\_for\\_Encryption](http://wiki.servicenow.com/index.php?title=Sample_Java_BouncyCastle_Algorithm_for_Encryption) [Accessed April 10, 2015].
- Wiki ServiceNow, 2012b. OpenPGP Decryption Example. Available at: [http://wiki.servicenow.com/index.php?title=Sample\\_Java\\_BouncyCastle\\_Algorithm\\_for\\_Encryption](http://wiki.servicenow.com/index.php?title=Sample_Java_BouncyCastle_Algorithm_for_Encryption) [Accessed April 10, 2015].
- WinZip Computing, 2013. Winzip. Available at: [http://kb.winzip.com/help/help\\_actions\\_encrypt.htm](http://kb.winzip.com/help/help_actions_encrypt.htm) [Accessed February 11, 2015].
- Xiaodong, D., David, S. & Adrian, W., 2000. *Practical Techniques for Searches on Encrypted Data*, Available at: [http://ieeexplore.ieee.org/xpls/abs\\_all.jsp?arnumber=848445](http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=848445).

Yemini, Y., 2010. GPG Wrapper. Available at:

<http://www.macnews.co.il/mageworks/java/gnupg/> [Accessed April 10, 2015].

Yen, A., Stockman, J. & Sun, W., 2015. ProtonMail. Available at: <https://protonmail.ch/>

[Accessed February 11, 2015].

Zhukov, A., 2002. JavaMaildir. Available at: <http://javamaildir.sourceforge.net/> [Accessed April 10, 2015].