Techniques for Enhancing the Performance of Secure Sockets Layer-Based Systems

by

Norman Lim, B. Eng. (CSE)

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Master of Applied Science

in

Electrical and Computer Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada K1S 5B6

Your file Votre référence
ISBN: 978-0-494-83039-0
Our file Notre référence
ISBN: 978-0-494-83039-0

The undersigned recommend to
the Faculty of Graduate and Postdoctoral Affairs
acceptance of the thesis

Techniques for Enhancing the Performance of Secure Sockets Layer-Based Systems

submitted by

Norman Lim, B. Eng. (CSE)

in partial fulfillment of the requirements for

the degree of Master of Applied Science in Electrical and Computer Engineering

---

Chair, Howard Schwartz, Department of Systems and Computer Engineering

---

Thesis Supervisor, Shikharesh Majumdar

Carleton University
August 2011

Abstract

Providing secure communications in distributed systems often comes at the cost of a performance penalty. The Transport Layer Security (TLS) protocol, previously called the Secure Sockets Layer (SSL) protocol, is one of the common protocols used to enable secure communication over the Internet. Due to the CPU-intensive security algorithms used, transferring data using TLS is known to be significantly slow. This thesis proposes a technique, called *security sieve*, which enhances the performance of SSL/TLS-based data transmission. Specifically, security sieve reduces the transmission time of transferring classified documents, which may also contain non-sensitive data. The security sieve technique separates the classified data from the non-classified data, and transmits the classified data over a secure channel, and transfers the non-classified data over a (faster) non-secure channel. At the receiving end, the separated data is re-assembled to reconstruct the original document.

Based on prototyping and measurement, an investigation of the security sieve technique is carried out. Experimental results are presented to demonstrate the effectiveness of security sieve compared to the state of the art: sending the entire document over a single secure channel. In addition to the basic security sieve technique, the effectiveness of using: (1) multiple channels to achieve concurrency of operations, and (2) batching of multiple file transfer requests to amortize the channel establishment/tear down overhead over multiple file transfers, is also investigated.

Acknowledgements

First and foremost, I would like to express my sincere thanks to my thesis supervisor, Professor Shikharesh Majumdar, for all his guidance, and continuous support throughout the completion of this thesis. In addition, I am grateful to Carleton University, the Natural Science and Engineering Council (NSERC), and the Government of Ontario for providing financial support for this research. I would also like to acknowledge Cistech Limited for their support in this research. Last but not least, I want to thank my parents, my brother, and my sister for their unwavering support and encouragement. Without them, completing this thesis would not have been possible.

Table of Contents

List of Tables

List of Figures

List of Abbreviations and Symbols

λ	Arrival Rate (in requests per second)
<$S>	Secure Start Tag
<$E>	Secure End Tag
3DES	Triple Data Encryption Standard
AES	Advanced Encryption Standard
AP	Additional Processing
API	Application Programing Interface
BES	Batch Even Split
BFS	Batch File Split
BSS	Batch Segment Split
CPU	Central Processing Unit
DES	Data Encryption Standard
DKS	Dynamic Key Size
ECC	Elliptic Curve Cryptography
EFS	Even File Set
ES	Even Split
GB	Gigabyte
GHz	Gigahertz
HTTP	Hyper Text Transfer Protocol
HTTPS	Secure Hyper Text Transfer Protocol
ID	Integrated Data
IDE	Integrated Development Environment
I/O	Input/output
JCE	Java Cryptography Extension
JSSE	Java Secure Socket Extension
KB	Kilobyte
M	Number of non-secure channels
MAC	Message Authentication Code
Mbps	Megabits per second
MB	Megabytes
MC-SSL	Multiple Channel Secure Sockets Layer
ms	Milliseconds
N	Number of secure channels
P	Proportion of classified information in a document
PO1	Performance Optimization 1: Using multiple channels to achieve concurrency of operations.
PO2	Performance Optimization 2: Batching of multiple file transfer requests.
RAM	Random Access Memory

RSA	Rivest Shamir Adleman
s	seconds
SBR	Signed Binary Representation
SDLL	Sub-data Lists Lengths
SHA	Secure Hash Algorithm
SS	Segment Split
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UFS	Uneven File Set
x	File size (in megabytes)

Glossary of Terms

(Secure or Non-secure) batch data list:

The data list that the batch sieve algorithm uses to store the (classified or non-classified) information of all the files in the batch data list being sieved.

(Secure or Non-secure) data list:

The data list that the sieve algorithm uses to store the (classified or non-classified) information contained in the text document that is being sieved.

(Secure or Non-secure) sub-batch data lists array:

In the multiple channel-based security sieve system with batching, it is the array that contains the (secure or non-secure) sub-data lists for multiple files. This array is returned after the batch split algorithm partitions the (secure or non-secure) batch data list.

***(Secure or Non-secure) sub-data lists array*:**

In the multiple channel-based security sieve system, it is the array of `ArrayList<String>` references that contains the (secure or non-secure) sub-data lists. This array is returned after the split algorithm partitions the (secure or non-secure) data list.

Batch data list:

In the systems that use batching, it is the list that contains the data for multiple files.

Batch split/combine algorithms:

The algorithms used by the multiple channel-based systems for partitioning the batch data list among multiple channels. At the receiving end, the data received from the multiple channels is re-combined.

Integrated Data:

A variable that the integration algorithm uses to store the data contained in the sieved data lists back in proper order.

Secure tag(s):

The general term used to refer to either the `<$S>` tag or `<$E>` tag. The plural form collectively refers to both the `<$S>` tag and `<$E>` tag.

Sieved batch data list(s):

The general term used to refer to either the secure batch data list or non-secure batch data list. Note that the plural form collectively refers to both the secure and non-secure batch data lists.

Sieved data list(s):

The general term used to refer to either secure data list or non-secure data list. Note that the plural form collectively refers to both the secure and non-secure data lists.

Split/combine algorithms:

The algorithms used by the multiple channel-based systems for partitioning data among multiple channels. At the receiving end, the data received from the multiple channels is re-combined.

Sub-batch data list:

In the multiple channel-based security sieve system using batching, it is the term used to describe the portion of a sieved batch data list that a channel sends. The batch split algorithm partitions a sieved batch data list into multiple sub-batch data lists.

Sub-data list:

In the multiple channel-based security sieve system, it is the term used to describe the portion of a sieved data list that a channel sends. The split algorithm partitions a sieved data list into multiple sub-data lists.

Sub-data list arrays:

The term used to collectively refer to the *secure sub-data lists array* and *non-secure sub-data lists array*.

Sub-data list lengths (SDLL) array:

An integer array used by the SS algorithm that keeps track of the total number of characters contained in each sub-data list.

Sub-document:

In the multiple channel-based secure-only system, it is the term used to describe the portion of a document that each channels sends. The split algorithm partitions the document to be transferred into multiple sub-documents.

Sub-documents array:

In the multiple channel-based secure-only system, it is the `String` array that contains all the sub-documents that each channel sends. This array is returned after the split algorithm partitions the document to be transferred.

Text segment:

The term used to refer to a `String` object contained in the sieved data lists.

Chapter 1: Introduction

Secure communications in distributed systems is very important for applications that require the exchange of highly sensitive information. These applications can include e-commerce applications, online-banking systems, and other business, health, or government related applications. However, providing security often comes at the cost of a performance penalty due to the CPU-intensive encryption/decryption and hashing algorithms used by the security protocols. This performance penalty can reduce the performance of a system and lead to higher latencies (e.g. response times). One of the common protocols used to enable secure communications over public communication mediums, such as the Internet, is the Secure Sockets Layer (SSL) protocol [1], now known as the Transport Layer Security (TLS) protocol [2]. This research concerns engineering SSL/TLS-based systems for enhanced system performance. Specifically, this thesis focuses on improving the performance of the data transmission software that is responsible for the transfer of classified text documents between two sites.

1.1 Motivation of the Thesis

In a distributed environment, exchanging text documents containing sensitive information is common. The state of the art for transferring a text document containing confidential information is to transmit the entire document using a single secure channel. This can result in long document transmission times, especially if the documents are large, because of the CPU-intensive security algorithms used to encrypt, decrypt, and hash the data that is transferred. In some cases, the text documents that are transferred will not exclusively contain classified information, but it is likely that the document will contain non-sensitive information as well. Non-sensitive information can include a

chapter, paragraph, sentence, or even a single word that does not need to be protected. For certain documents it is possible to protect the confidentiality of the document by only protecting the most sensitive information. For example, when sending a lengthy document on a patient's medical history, only the name, the social insurance number and the address of a patient may need to be protected. Furthermore, when displaying a web page containing confidential banking information, there are parts of the web page, including the HTML tags, JavaScript code, and images, which are not sensitive and do not need to be protected. When sending structured data, such as a database, it may be possible to leave certain rows or columns that contain non-sensitive information unprotected. Determining which information should be protected is the responsibility of the user transmitting the document. For cases where the information is highly sensitive, the entire document may need to be protected, but in some cases, only a portion of the document needs to be protected.

1.2 Proposed Solution

This thesis proposes a performance enhancement technique called *security sieve*, which separates the classified and non-classified information in a document and sends the separated information over a secure channel and a (faster) non-secure channel, respectively (see Figure 1.1). At the receiving end, the separated data is re-assembled so that the original document can be reconstructed. Security sieve is based on the concept of selective security, which is the idea of applying security to the most sensitive information, and therefore limiting the use of the expensive encryption/decryption and hash operations. Sending only the classified information over the secure channel and sending the remaining information over a non-secure channel can potentially reduce the

overall document transmission time. To the best of our knowledge, no such a technique has been deployed by systems described in the existing literature. The sieve and integration algorithms are discussed in more detail in Section 3.1 and Section 3.2, respectively.

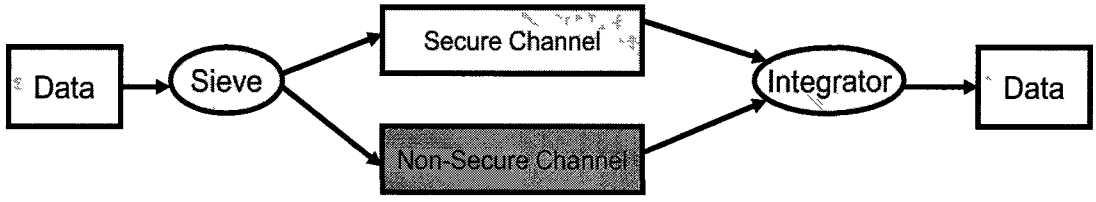


Figure 1.1: Overview of security sieve technique.

The classified information in a document can be "marked" by the author (see Figure 1.2), which is the approach that is used in this thesis. In this sample document, the classified information that needs to be protected are the phrases: "Norman Lim" and "Carleton University". Note that marking corresponds to delimiting each phrase with a pair of special strings: "<$S>" and "<$E>" that does not appear in the document being transmitted.

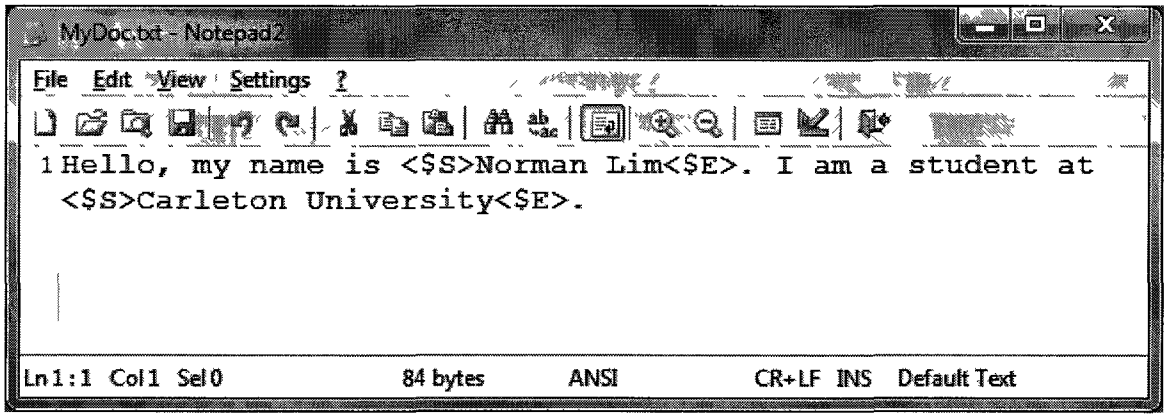


Figure 1.2: A sample marked text document.

This research focuses on the security sieve technique, and how much reduction in overall document transmission time security sieve can achieve when compared to using a single secure channel-based system. To evaluate the performance of the security sieve technique, a prototype security sieve client-server system (*security sieve system* for short) was developed, and its performance was compared to the performance of a single secure channel-based client-server system (*secure-only system* for short). The design and implementation of these systems is discussed in Section 4.1 and Section 4.2, respectively.

1.2.1 Additional Performance Optimization Techniques

In addition to the basic security sieve technique, this thesis also investigates using two other performance optimization techniques. *Performance Optimization 1* (PO1) involves adding multiple channels to achieve concurrency of operations. *Performance Optimization 2* (PO2) consists of batching multiple file transfer requests to amortize the channel establishment/tear down overhead over multiple file transfers. PO1 and PO2 are deployed in both the security sieve and secure-only systems to investigate how using these techniques affects system performance. PO1 and PO2 are discussed in more detail in Section 3.3 and Section 3.4, respectively.

1.3 Scope of the Thesis

This thesis focuses on the security sieve technique. As discussed in Section 1.2, in this thesis a "marked by author approach" is followed. The classified information in a document is "marked" by the author of the document. A disadvantage with this technique is that human error may be a problem: in a text document the author may forget to identify some information as confidential. However, for more structured data, such as databases, determining which data should be protected can be more straightforward. For

example, in a database it is easy to identify certain columns or rows of a table that should be protected. More advanced techniques for determining how to mark the classified information in a document is beyond the scope of this thesis, and can form a direction for future research. Further discussion is provided in Section 6.2.

1.4 Contributions of the Thesis

The main contributions of the thesis are presented next.

- *Security sieve*: a novel technique that separates the classified and non-classified information in a document is introduced. The classified information is transferred using a secure channel, whereas the non-classified information is transferred using a (faster) non-secure channel.
- Algorithms are devised for partitioning a single document's data for sending over multiple channels so that concurrency of operations can be achieved. Additional algorithms are also developed for partitioning a batch data list, which contains the data for multiple files.
- A prototype security sieve system is built and its performance is evaluated through measurements made on the system. Various insights into system behavior and performance are obtained from the experimental results that includes the following:
    - A demonstration of the effectiveness of security sieve when transferring documents with varying size and proportion of classified information is presented. The performance improvement of the security sieve system compared to the secure-only system is discussed.
    - The effect of different cipher suites on the performance improvement achieved by the security sieve system is presented.
- The performance of security sieve when used in a wireless environment is discussed.

- The effectiveness of the performance optimization techniques, PO1 and PO2, is demonstrated. The impact of various system and workload parameters on the performance improvement produced by these techniques is discussed.
- A simulation-based analysis of the security sieve system using PO2 and subjected to an open stream of request arrivals is conducted.

A paper [3] based on the initial research results has been published in a refereed international conference.

1.5 Thesis Outline

The rest of the thesis is organized as follows. Chapter 2 provides background information on the SSL/TLS protocol, and discusses related work. Chapter 3 describes the security sieve algorithms, and the performance optimization techniques in more detail. Chapter 4 explains the design and implementation of the prototype secure-only and security sieve systems, including how the performance optimizations were implemented. Chapter 5 presents and discusses the results of the performance evaluation conducted on the security sieve system, secure-only system, and the systems deploying the two performance optimization techniques. Lastly, Chapter 6 concludes the thesis and discusses possible directions for future research.

Chapter 2: Background and Related Work

This chapter presents background information on the SSL/TLS protocol and discusses related work.

2.1 Overview of Secure Communications

Secure communication is essential in computer systems that require the exchange of highly sensitive information. For example, computer and information systems in the commerce, government, health, or business fields need to provide the necessary mechanisms to ensure authentication, data integrity, and data confidentiality when transferring sensitive data. Authentication, integrity, and confidentiality are the three main aspects of security [1]. Authentication is used to confirm that the communicating parties are indeed the entity they claim they are. Confidentiality ensures a third party cannot view the information that is transferred. Lastly, integrity lets the communicating parties know if information was modified when it was being transferred. The accepted standard for Internet security is the TLS protocol [2] [4], a successor to the SSL protocol [5].

2.2 Overview of the TLS/SSL Protocol

SSL was originally developed by Netscape Communications in 1994 [5] to provide application-independent secure communications over the Internet. SSL was mainly used to securely transfer Web data between a Web browser and Web server. SSL operates on top of the Transmission Control Protocol (TCP), usually on port 443 [4]. In January 1999, the Internet Engineering Task Force modified the SSLv3 protocol (released in 1995), and renamed it TLSv1.0 [2]. The SSL/TLS protocol uses cryptography, digital

signatures, and certificates to provide data confidentiality, data integrity, and client /server authentication, respectively [5]. A brief description on cryptography, digital signatures, and digital certificates is presented next.

2.2.1 Cryptography

TLS uses cryptography algorithms to encrypt the original information (called the *plaintext*) to a form that is unreadable and has no semantic meaning (called the *ciphertext*). Once the ciphertext is decrypted with the proper key, the ciphertext is transformed back to plaintext. A typical encryption algorithm has two main inputs: (1) the plaintext and (2) the encryption key. There are two types of cryptography: *symmetric (private key) cryptography* and *asymmetric (public key) cryptography*. Each of them is briefly discussed next.

2.2.1.1 Symmetric Cryptography

Symmetric cryptography algorithms, such as the Data Encryption Standard (DES) [6] and Advanced Encryption Standard (AES) [7], use the same key for encryption and decryption. Symmetric cryptography is commonly used to encrypt data for bulk data transfer. The shared key is agreed upon by both sides, and remains the same for the entire data transfer. Symmetric key lengths usually range from 40 to 256 bits.

There are two types of symmetric cryptography algorithms: (1) *stream ciphers* and (2) *block ciphers*. Stream ciphers (e.g. RC4 [8]) encrypt/decrypt data one bit at a time, whereas block ciphers encrypt/decrypt data in blocks of multiple bits at a time. The block size determines the number of bits that a block cipher encrypts at a time. DES and AES are examples of block ciphers that use block sizes of 64 bits, and 128 bits, respectively.

2.2.1.2 Asymmetric Cryptography

Asymmetric cryptography algorithms, such as Rivest Shamir Adleman (RSA) [9], use separate keys for encryption and decryption, and are commonly used to exchange the shared key used in symmetric cryptography. In asymmetric cryptography, an entity has two keys: a *public key*, and a *private key*. The public key is public knowledge and is used by other entities to encrypt messages intended for the owner of the public-private key pair. Conversely, the private key is kept secret and is only known by the owner, since only the owner of the private key can decrypt the messages encrypted with the public key.

2.2.2 Digital Signatures

The TLS protocol signs each message with a digital signature for data integrity purposes. Specifically, the TLS protocol uses hash functions, such as SHA-1 [10], to produce a fixed-length message digest, called the message authentication code (MAC). The Secure Hash Algorithm (SHA) has a 160-bit message digest size. The inputs into the hash functions include the symmetric key and the message. Both the message and MAC are encrypted and sent to the receiver. The receiver decrypts the data, extracts the message, and hashes the message using the same hash algorithm to generate its own MAC. If the MAC that the sender transmitted is the same as the one the receiver generates, the receiver can safely assume that the contents of the message were not changed. If the MACs are different, the receiver knows that the message was changed, and may have been tampered with during transmission.

2.2.3 Digital Certificates

TLS uses digital certificates to authenticate servers, and optionally, clients can be authenticated as well. Digital certificates are digital documents used to verify that a specific public key is owned by the correct entity. X.509 [11] is a commonly used certificate standard that defines what information a digital certificate contains and what data format the information is stored in. Common information found on a digital certificate includes: name of the entity, organization, address, public key, signature algorithm, and more. Digital certificates help prevent a malicious entity from impersonating another entity with a false public key.

2.3 Description of SSL/TLS Protocol

The TLS protocol comprises of three protocols [5]: (1) Handshake Protocol, (2) Record Protocol, and (3) Alert Protocol. Each of these protocols is discussed next.

2.3.1 Handshake Protocol

The Handshake Protocol is used during the session negotiation phase (also called the handshake phase). The handshake phase consists of cipher suite negotiation, authentication, and symmetric key exchange. The cipher suite specifies the encryption algorithms (ciphers) to use for authentication, bulk data encryption/decryption, and symmetric key exchange. The symmetric key exchange, which is performed using an asymmetric cryptography algorithm (e.g. RSA [9]), is used to mutually establish the secret key that is going to be used for bulk data encryption/decryption.

2.3.2 Record Protocol

The Record Protocol phase (also called the bulk data transfer phase) is started after the session negotiation phase is completed. In this phase, a symmetric encryption

algorithm and the secret key established during the handshake phase are used to encrypt/decrypt the transmitted data. The Record Protocol specifies a common format in which to transfer data, in the form of a TLS Record. The TLS Record includes the following information: message type, protocol version, application data, data length, MAC, and padding/pad length. If the size of the plaintext is not an exact multiple of the block size, padding is used to increase the size of the plaintext to the required size. Recall that block size is the number of bits that a symmetric cipher encrypts at a time. Padding is also important because it makes the plaintext data less predictable and harder for attackers to discover the size of the plaintext data. The length of the application data (i.e. size of user's data) in a TLS Record cannot exceed 16KB.

2.3.3 Alert Protocol

The Alert Protocol is only used when something goes wrong during the Handshake Protocol or Record Protocol phases. Depending on the severity of the error, two types of alert messages can be sent: warning or fatal. If the alert level is fatal, the connection or security may be compromised, and the session is likely to be terminated.

2.4 Performance Cost of Secure Communications

It is well known that applying secure communications comes at the cost of a performance penalty. Sending data using a secure channel can result in long data transfer times because of the CPU-intensive operations that need to be applied to the data. These operations include: operations performed by cryptography algorithms (to encrypt/decrypt data), and hash algorithms (to sign the data). Additional overheads are also incurred when opening (e.g., during the TLS handshake phase), and closing of the secure channel. The performance impact of using SSL/TLS and the cryptography algorithms it uses was

extensively studied in [1], [12], [13], [14], and [15]. Each of them focuses on a particular aspect of secure communication and is discussed next.

2.4.1 Performance Impact of SSL/TLS

In [1], Zhao presents an in-depth analysis of a secure session. The paper focuses mainly on the SSL protocol, which has similar components to other security protocols. The commonly used SSL cryptographic algorithms, including RSA, DES, AES, and SHA-1, are also analyzed. Zhao broke down the data for a secure session in order to show how asymmetric cryptography, symmetric cryptography, and hashing contribute to the SSL performance cost. The results of experiments showed that cryptography consumed about 71% of the total processing time in a HTTPS transaction. When 1KB of data is transferred, asymmetric cryptography, which is used during the handshake (session negotiation) phase, contributes to almost 90% of the SSL processing. However, when the data exchanged is greater than 32KB, the bulk data transfer phase becomes more dominant, and the processing associated with symmetric cryptography and hashing, starts to increase.

Kant [12] analyzed the performance and architectural impact of SSL on servers in terms of throughput, utilization, number of processors, and other parameters. The experiments were conducted on an Intel Pentium III Xeon Server running Windows NT 4.0. The experiments were run where the client repeatedly retrieved webpages from a webserver. There were three webpage file sizes: small (30 bytes), average (36KB), and large (1MB). The experiments were run with and without SSL for comparison purposes. The ciphers used were: 512-bit RSA and 128-bit RC4. For one processor, the non-SSL throughput was found to be 5.6 to 7.1 times higher than SSL throughput. However, the

authors found that increasing the number of processors, did improve the SSL throughput. In a response time comparison, the authors found that the response time increased by up to a factor of 10 when using SSL compared to not using SSL. Specifically, when transferring a 1MB file, it took 1-2 seconds without SSL, versus 10-15 seconds when using SSL.

2.4.2 Effect of Using HTTPS

In [13], Goldberg et al. compares the performance of the hypertext transfer protocol (HTTP) and secure HTTP (HTTPs). HTTPS uses SSL for security. The authors created a small Intranet environment by connecting two PCs using 10 Mbps Ethernet. The PC that acted as the Webserver had the following specifications: a 200 MHz Intel Pentium CPU, 256MB of RAM, and ran Windows NT 4.0. The client PC had a 100Mhz Pentium CPU, 32MB of RAM, and also ran Windows NT 4.0. Experiments were conducted where webpages that had sizes ranging from 1000 bytes to 100 000 bytes were transferred. The experiments were run with HTTP and HTTPS. Results showed that transfer rates decreased by about 22% when HTTPS was used instead of HTTP.

2.4.3 Performance of Handheld Wireless Devices

In [14], Argyroudis et al. conducted a performance analysis of SSL on a HP iPAQ H3630 (206 MHz) wireless handheld device. Running cryptographic algorithms on a handheld device can be even slower and expensive compared to running the algorithms on a fixed workstation because of the limited processing speed, memory, and battery life of handheld devices. The authors conducted a 1MB file transfer between two handheld devices using 1024-bit and 2048-bit RSA public key encryption, and 256-bit AES private key encryption. The mean time for the entire transaction using a 1024-bit public key and

a 2048-bit key were: 7.76s and 8.73s, respectively. In comparison, the mean transaction time without using any security protocol was 4.25s.

In [15], Berbecaru presented a comprehensive analysis of the SSL protocol (and its cryptography algorithms) on various handheld devices. Berbecaru modeled a small wireless environment with various handheld devices to measure the overhead of using SSL. Establishing an SSL connection with full handshake (using a 1024-bit RSA public key encryption) from an iPAQ (400MHz) to a Jornada (206MHz) required approximately one second, which was about three times more expensive than establishing connection to a 1.7GHz laptop. The experiments also showed that transferring 1KB of data using SSL (using 1024-bit RSA, and 128-bit AES) took about ten times longer compared to sending the data without security protocols.

2.5 Techniques for Improving the Performance of Secure Communication

As discussed in Section 2.4, providing secure communications can degrade the performance of a system because of the CPU-intensive cryptography and hash algorithms used to protect the data that needs to be transferred. As a result, much effort has been invested into developing techniques to improve the performance of secure communications. One of the ways to improve the performance of secure communications is to reduce the processing time required to execute the cryptography algorithms. There are two common techniques to speedup cryptography algorithms [5]: (1) use specialized hardware accelerators with dedicated modular arithmetic units, and (2) use software optimized cryptography algorithms and techniques. The following sub-sections discuss

some of the techniques that are used to improve the performance of secure communication.

2.5.1 SSL Hardware Accelerators

SSL adapter cards, as discussed in [16], can be used to offload servers from the CPU-intensive parts of a SSL transaction, including symmetric and asymmetric cryptography. The idea behind using SSL adapter cards for performing the security-related algorithms is similar to the idea of using graphic cards for reducing the graphics-related processing that a CPU has to perform. The SSL adapters are used to improve the performance of SSL-based communication, and can negotiate handshakes, as well as perform the expensive encryption/decryption operations. This allows the server to focus on processing client requests, which increases the number of transactions the server can complete per second. Using an SSL adapter can also reduce the response time delays because the reduction in CPU load also reduces the delay for clients. SSL hardware accelerators usually have dedicated modular arithmetic processing units, which are designed to accelerate the RSA encryption/decryption operations. Recall, that RSA is an asymmetric cryptography algorithm that is commonly used during the SSL handshake phase. Thus, these accelerators can improve system performance by reducing time spent during the handshake phase.

2.5.2 Software Optimized Techniques

In [17], Potlapally et al. propose algorithmic optimizations and techniques to improve the computational efficiency of asymmetric encryption. Emphasis is put on modular exponentiation-based crypto algorithms, such as RSA. The authors analyze algorithmic optimizations like Chinese Remainder Theorem and Montgomery

Multiplication, and other advanced techniques including block size selection and algorithm-level caching.

2.5.2.1 Elliptic Curve Cryptography

Elliptic Curve Cryptography (ECC) [18] is an alternative approach to public-key cryptography that is able to provide a comparable level of security as other public-key encryption methods, but with a smaller key size and faster computations. ECC was originally introduced by Victor Miller and Neal Koblitz in 1985, but has recently attracted the attention of researchers. In [19] Gupta, et al., compare the performance of using ECC with using RSA for SSL transactions. Their results show that a RSA decryption speedup factor of 2.4 can be achieved when using a 160-bit ECC key instead of a commonly used 1024-bit RSA key. It has been shown that 160-bit ECC key provides an equivalent level of security as a 1024-bit RSA key. In [20], Wang et al. propose a new signed binary representation (SBR) technique to improve the performance of ECC. ECC performance is dependent on arithmetic operations such scalar multiplication. The authors claim that the proposed SBR technique, which is used to represent integers, requires less memory and is faster than the traditional methods.

2.5.3 Selective Security

In [21], [22], [23], and [24] selective security is used. The idea is to apply security (or different levels of security) to the most sensitive information. A key benefit of selective security is that the costly cryptographic operations used during the bulk data transfer phase can be limited by applying the operations only to the data that requires it most.

2.5.3.1 Selective Security for Content Adaptation Purposes

In [21], Portmann and Seneviratne propose using selective security in systems that use content adaptation proxies. The purpose of content adaptation is to tailor the content of data to meet a client's capabilities. For example, the quality of a media file may need to be lowered for a client that has restricted processing capabilities or network resources. The problem is that content adaptation proxies are incompatible with end-to-end security. The authors propose a simple extension to TLS protocol that allows an application to selectively protect elements within a data stream. The idea is to apply security to only the sensitive elements of the data stream, and expose the rest of the data to an intermediary system for potential content adaptation purposes. A new record layer type is added to the TLS protocol stack. The new record type, Cleartext Application Data, is used to transport the non-sensitive data.

2.5.3.2 The Dynamic Key Size Architecture

The authors of [22] propose a Dynamic Key Size (DKS) architecture that can be integrated into security protocols to provide more efficient secure mobile communication. The proposed approach uses information sensitivity level (specified by the user) and device capability to select a suitable algorithm key length. An application is provided an interface for selectively choosing information at different security levels depending on the sensitivity of the information. The goal is to maintain a balance between performance and security. The algorithm key length selection is based on: (1) the device's processing and network capability, and (2) the information security level. The device capability is measured using three categories: processing, power (battery life), and network capabilities. Each of these categories has three classification levels: 0 (low), 1 (medium),

and 2 (high). The mobile device's performance level is calculated by adding up these three category values. Furthermore, there are four information security levels: most critical, critical, least critical, and not critical. Each information sensitivity level has seven levels with each level corresponding to a device performance level. After the information sensitivity level is chosen, and the device performance level is calculated, a key size can be determined. In this paper, the authors have not discussed how key size is calculated, but have mentioned that the higher the performance level of the device and higher the information sensitivity level of the data, the larger the key size will be.

2.5.3.3 Multiple Channel SSL

An extension to the SSL protocol called multiple channel SSL (MC-SSL) was proposed in [23]. The idea is to have multiple channels between the client and server. Each channel has a different level of security, and is used to transfer data that with various levels of information sensitivity. MC-SSL allows the client to negotiate multiple SSL channels each with its own security characteristics (e.g. cipher suite).

2.5.3.4 Adaptive Software Architecture

In [24], adaptive secure software architecture is proposed to support e-commerce transactions. The aim is to allow dynamic decision making that deals with the trade-off between security and performance. The system uses four security classes defined based on the degree of network congestion and information sensitivity. As the security level increases, the number of keys used for encrypting messages increase. For example, in first level, the data is only encrypted with one key, whereas in the second level the data is encrypted with the first level's key along with another key.

2.5.4 Using Parallelism to Improve Performance

The work in [25] and [26] examine how applying parallelism can improve the performance of security protocols, and are discussed next.

In [25], Alaidaros et al. propose an algorithm that aims to speed up the bulk data transfer phase in SSL. The proposed algorithm performs the encryption of data, and the calculation of the MAC in parallel. The encryption of the MAC is done separately after it has been calculated. The original SSL protocol calculates the MAC first, and then encrypts the data and the MAC together. The parallel algorithm was simulated with two processors, and it was found that the new algorithm achieved a speedup of 1.74 over the sequential algorithm.

Nahum et al. [26] perform an experimental study to demonstrate how encryption protocol performance can be improved using parallelism. The results showed that both packet-level and connection-level parallelism can improve performance. In packet-level parallelism the unit of concurrency is a packet, whereas in connection-level parallelism each connection is assigned a processing element. Due to the CPU intensive nature of cryptographic protocols, parallelism is an effective means for improving performance.

2.5.5 Comparison with Related Work

The proposed security sieve technique is also based on the concept of selective security, as discussed in [21]; however the main difference between this research and the existing research is that this research focuses on improving the performance of transmitting documents containing both classified and non-classified information, using the standard SSL/TLS protocol. The techniques described in [22], [23], and [24] allow clients to specify an information sensitivity level for the document they want to transfer,

and then the entire document is transferred using a single secure channel that has a security level that meets the document's security requirements. With security sieve, the client can specify which information within a document is sensitive and needs to be protected, and which information does not need to be protected. The security sieve technique separates the classified and non-classified information in a document, and transfers the separated information over a secure and a non-secure channel, respectively. At the receiving end, the classified and non-classified information is recombined to reconstruct the original document. To the best of our knowledge, no such technique has been deployed by systems discussed in the existing literature.

Chapter 3: The Security Sieve Algorithms and the Performance Optimization Techniques

This chapter explains the security sieve algorithms, which are used to separate and re-combine the classified and non-classified information contained in a marked text document. In addition, the performance optimizations techniques: using multiple channels, and batching of multiple file transfer requests, are discussed in more detail.

3.1 Sieve Algorithm

The sieve algorithm separates the classified and non-classified information in a marked text document and stores the information in two data lists: the *secure data list* and *non-secure data list*, respectively. The secure data list contains the classified information and is transmitted using the secure channel. Similarly, the non-secure data list contains the non-classified information and is transferred over the non-secure channel. The secure data list and non-secure data list are collectively referred to as the *sieved data lists*. Figure 3.1 shows an illustration of the sieve algorithm. The sequence of arrows in this diagram shows the order in which the sieve algorithm separates the non-classified and classified information in the original marked text document. For example, arrow 1 illustrates the *First List Tag* being added to the non-secure data list to indicate that data should be retrieved from this list first when it is time to integrate the sieved data lists. Arrows 2 through 5 illustrate the classified and non-classified information being copied, and added to the appropriate sieved data lists.

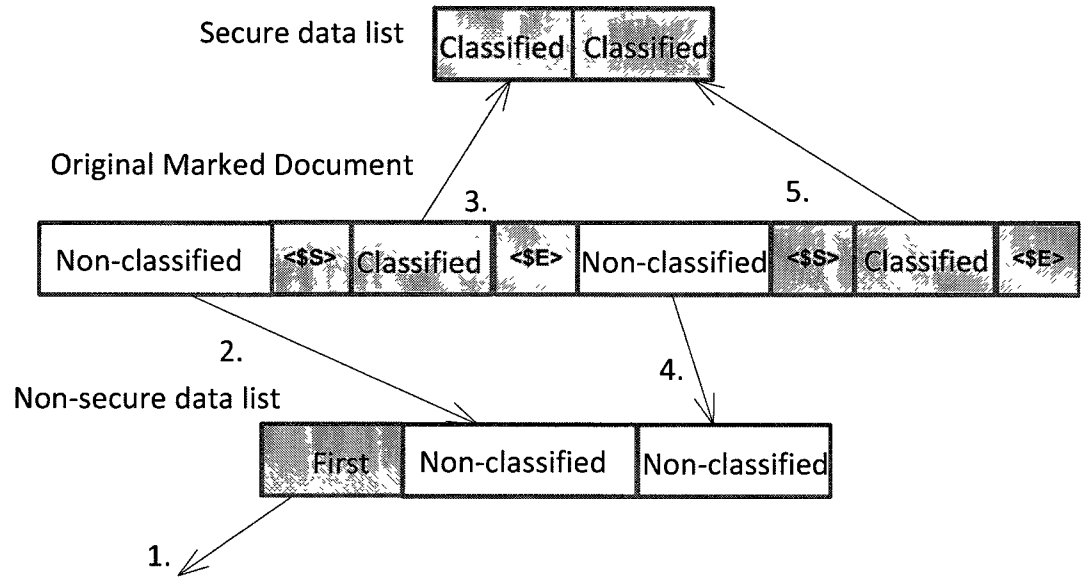


Figure 3.1: Illustration of the sieve algorithm.

To distinguish between classified and non-classified information in the document, the classified information is marked by the author with the following tags (collectively referred to as the *secure tags*):

1). *Secure Start Tag*, `<$S>`: indicates where the classified text begins, and

2). *Secure End Tag*, `<$E>`: indicates where classified text ends.

A sample marked document was shown in Figure 1.2. It is assumed that the special strings used to represent the secure tags are not part of the author's text in the document. Note that the secure tags can be represented using other strings as well. The sieve algorithm treats the text in the document to be sieved as a string, and searches this string for the secure tags. Two indices are used to keep track of the position that is being examined in the string:

1). *Current Index*: stores the starting position of the most recently found secure tag

2). *Previous Index*: stores end position of the previously found secure tag

As discussed earlier, the sieve and integration algorithms also use a special tag called the *First List Tag*, <!First!>, which indicates the sieved data list (i.e. secure or non-secure data list) that data should be taken from first when it is time to integrate the sieved data lists.

A flow chart of the sieve algorithm is shown in Figure 3.2. Each step in the flow chart is explained next. First, the current and previous indices are initialized to zero. Next, the sieve algorithm checks whether the document starts with a <$S> tag, or in other words if the document starts with classified information (see Step 2). If so, the first list tag is added to the secure data list (see Step 3a); otherwise the first list tag is added to the non-secure data list (see Step 3b). If the document starts with classified data, the algorithm continues with Step 4. Otherwise, the document did not begin with <$S> tag, and the algorithm skips Step 4 and goes directly to Step 5, which is the start of the second phase of the sieve algorithm.

In Step 4, the algorithm starts by finding the starting position of the <$E> tag (recall that a <$S> tag was already found). The starting position of the <$E> tag is stored in the current index. The data between the previous index and the current index (i.e. the text between the <$S> and <$E> tags) is copied, and added to a new entry in the secure data list. The previous index is then updated by assigning the current index value to it. Updating the previous index also involves adding the length of the secure tag (which is four) to the previous index value so that the previous index value points to the end position of the secure tag.

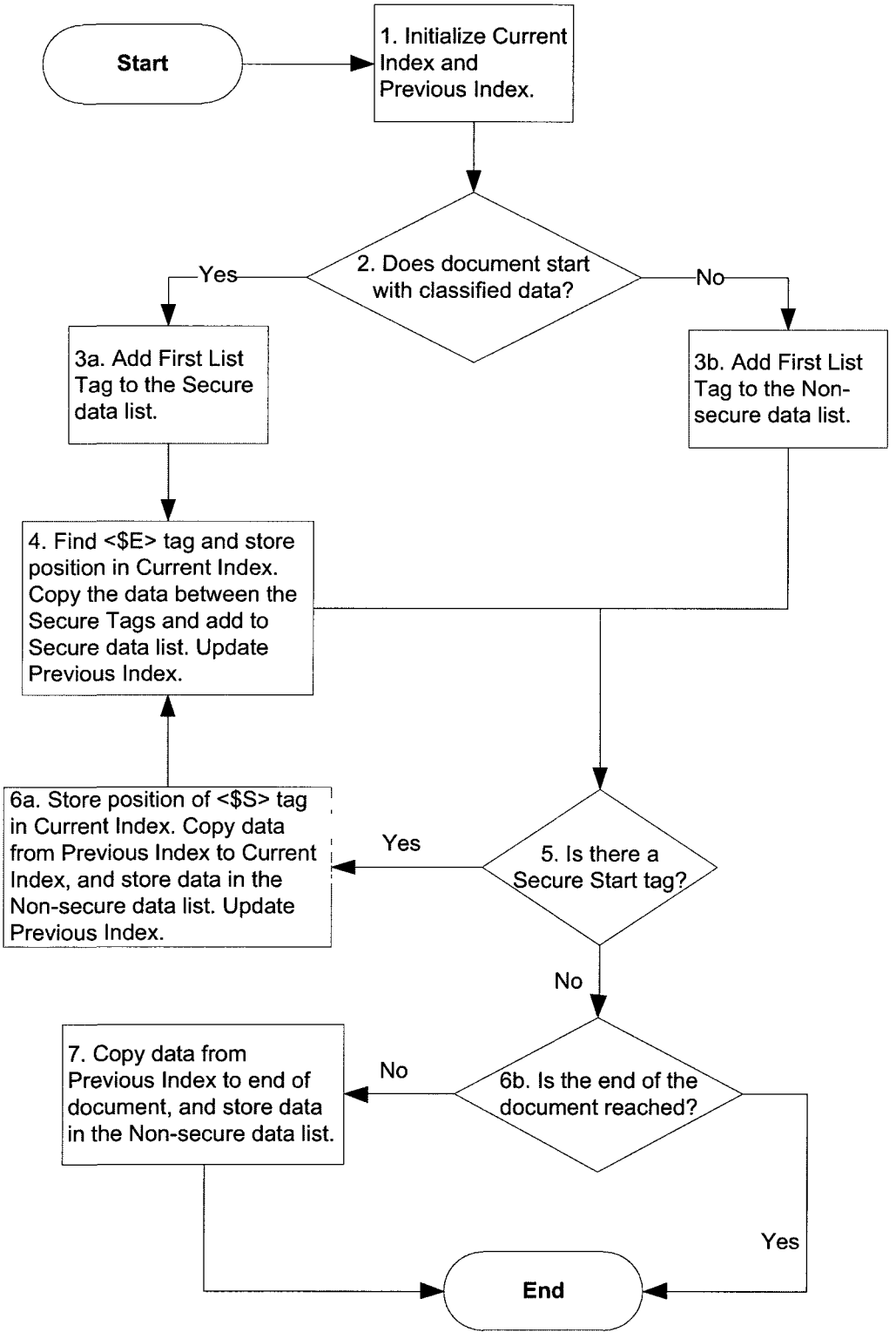


Figure 3.2: Flow chart for the sieve algorithm.

Step 5 is the start of the second phase of the sieve algorithm. In this phase, the algorithm continually searches the document for more <$S> tags, starting from the current index position, until no more <$S> tags are found. If a <$S> tag is found, the algorithm goes to Step 6a; otherwise, the algorithm goes to Step 6b. In Step 6a, the starting position of the <$S> tag is stored in the current index, and the text between the previous index and current index is copied and added to a new entry in the non-secure data list. The previous index is then updated again. The algorithm then goes back to Step 4, where the matching <$E> is found, and then the text between the <$S> and <$E> tags is copied and added to the secure data list. The algorithm then goes back to Step 5, and the operations continue in this fashion until no more <$S> tags are found.

Once no more <$S> tags are found, the algorithm then checks if the end of the document has been reached (Step 6b). If so, the algorithm ends; otherwise, Step 7 needs to be executed before the algorithm ends. In Step 7, the data between the previous index and the end of the document is copied, and added to the non-secure data list.

3.2 Integration algorithm

An illustration of the integration algorithm is shown in Figure 3.3. The integration algorithm puts the data contained in the secure and non-secure data lists back in sequence, in a variable named *Integrated Data* (ID). The sequence of arrows in this diagram shows the order that the integration algorithm recombines the data contained in the sieved data lists. For example, arrow 1 illustrates the *First List Tag* being removed from the non-secure data list. Arrows 2 through 5 illustrate the classified and non-classified information being removed from the sieved data lists, and being re-combined in the ID variable.

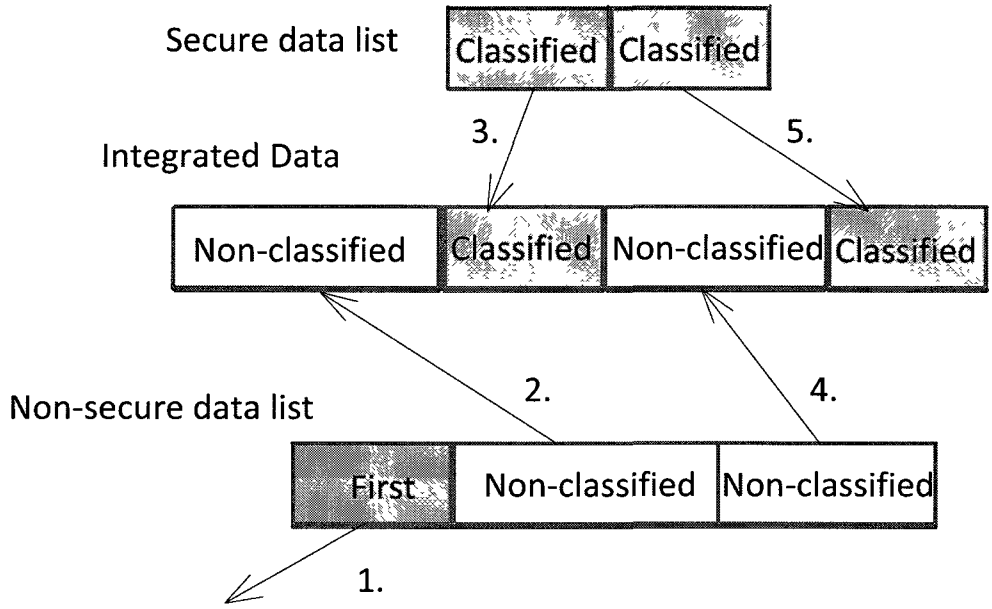


Figure 3.3: Illustration of the integration algorithm.

A flow chart of the integration algorithm is shown in Figure 3.4. Each step in the flow chart is explained next. First, the algorithm determines which data list (secure or non-secure); data should be retrieved from first. This is done by checking the first entries of each list for the `<!First!>` tag. If the first list tag is found in the secure data list (see Step 2a), the first list tag is removed. In addition, the first data element is removed from the secure data list and is stored in the ID variable. Otherwise, the non-secure data list is the first list (see Step 2b), and only the first list tag is removed since the second phase of the algorithm (beginning at Step 3) starts by removing data from the non-secure data list.

The second phase of the algorithm removes data from the non-secure data list and secure data list (alternating between these lists in this sequence) until both lists are empty. The removed data is stored in the ID variable. In Step 3, the algorithm checks if the non-secure data list is empty. If so, the algorithm proceeds to Step 5; otherwise, the first data

element is removed from the non-secure data list and is added to the ID variable (see Step 4). Steps 5 and Steps 6 are similar to Steps 3 and 4. The only difference is this time the operations are performed on the secure data list instead of the non-secure data list. In Step 7, the algorithm checks if both the non-secure and secure data lists are empty. If so, the algorithm exits; otherwise, the algorithm returns to Step 3. The integration algorithm is able to put the data contained in the sieved data lists back in sequence using this simple procedure because the data in the sieved data lists are also kept in order.

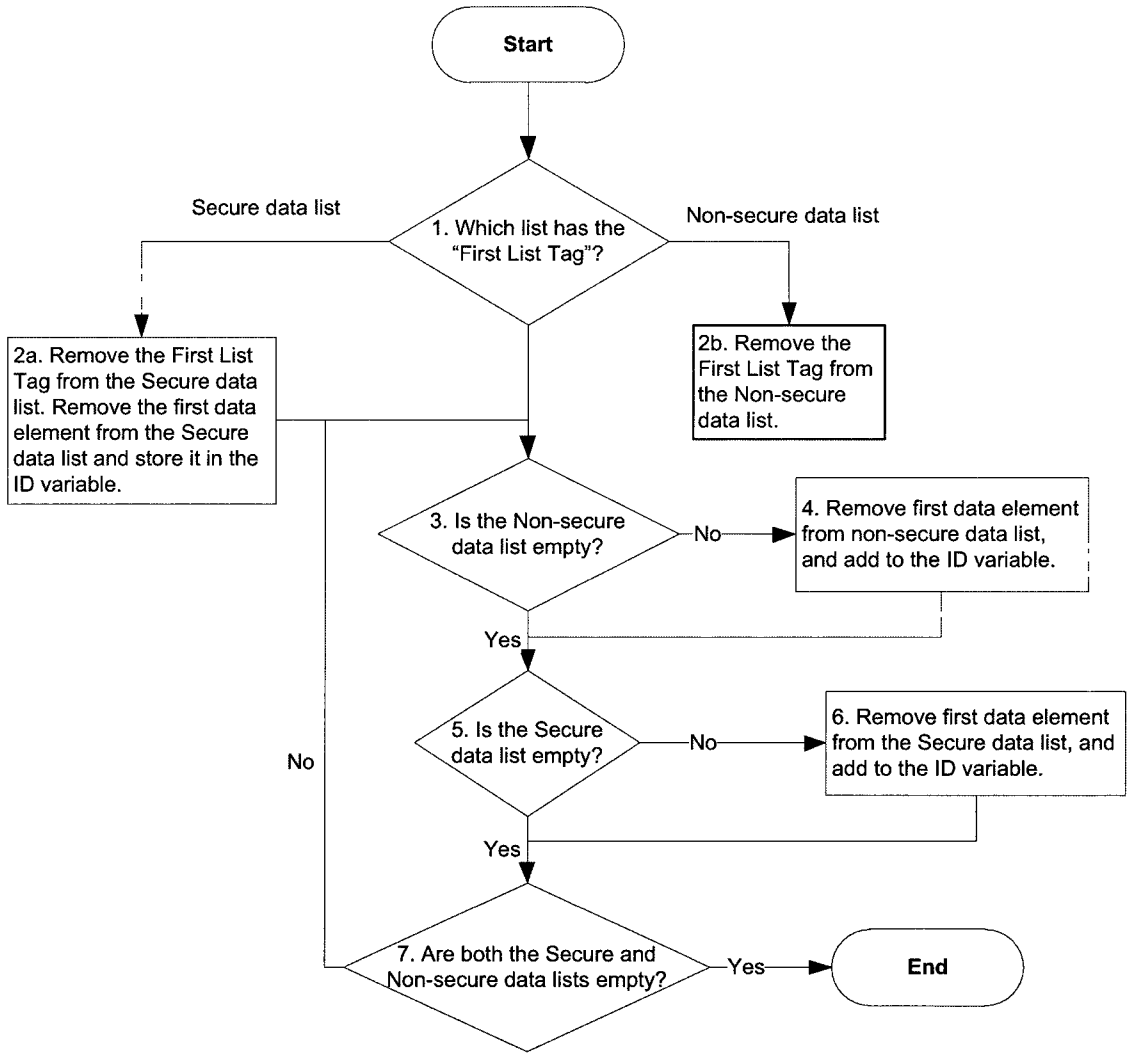


Figure 3.4: Flow chart for the integration algorithm.

The security sieve algorithms were explained in the two previous sections. In the next two sections, the two performance optimization techniques are discussed in more detail.

3.3 Performance Optimization 1: Multiple Channels (PO1)

PO1 involves using multiple channels to achieve concurrency of operations. Adding concurrency can potentially reduce the response time, but at the cost of a higher connection time since multiple channels need to be established and terminated. In the multiple channel-based secure-only system (see Figure 3.5a), a splitter partitions the document to be transmitted into N sub-documents, where N is the number of secure channels. At the receiving end, the binder re-combines the N sub-documents to reconstruct the original document.

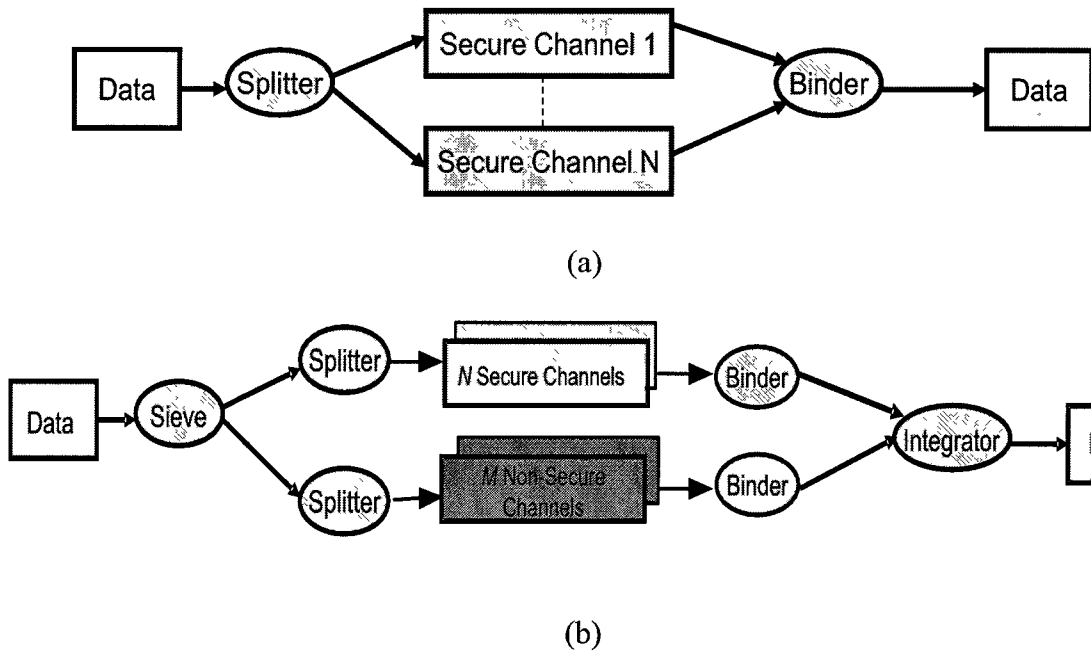


Figure 3.5: Overview of PO1 for (a) the secure-only system and (b) the security sieve system.

For the multiple channel-based security sieve system (see Figure 3.5b), the data is partitioned after sieving the document. The secure and non-secure data lists are each partitioned into N and M sub-data lists, respectively, where N and M is the number of secure and non-secure channels. At the receiving end, the original secure and non-secure data lists are reconstructed by re-combining the sub-data lists. After retrieving the secure and non-secure data lists, these lists are integrated to reconstruct the original document. Additional details on the design and implementation of PO1 is provided in Section 4.3.

3.4 Performance Optimization 2: Batching (PO2)

PO2 consists of batching multiple file transfer requests into a batch data list, and then transferring this batch data list; instead of transferring a single file at a time. Batching can enhance performance by amortizing the channel establishment/tear down over multiple file transfers, and can be used in situations where a client needs to send multiple files to the server. In addition, batching can potentially reduce costs in environments that levy a price per transaction.

Batching for the security sieve system (see Figure 3.6) works as follows. Multiple file transfer requests are batched by reading the data from multiple files, and storing the data in the batch data list. Once the batch data list is ready to be transmitted (i.e. when a given number of files are in the batch), the batch data list is sieved using the B-Sieve (or batch sieve) algorithm. The batch sieve algorithm stores the secure and non-secure batch data in the secure and non-secure batch data lists (collectively known as the *sieved batch data lists*), respectively. The secure and non-secure batch data lists are sent using a secure and non-secure channel, respectively. After receiving the sieved batch data lists, the data lists are integrated to retrieve the original batched data list. This is done using the B-

Integrate (batch integrate) algorithm. Note that batching for the secure-only system is the same as the security sieve system, except that the sieving and integrating steps are not required.

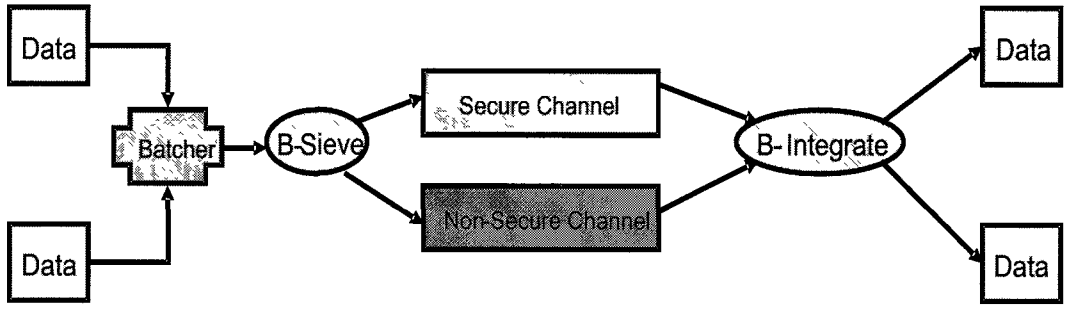


Figure 3.6: Overview of PO2 for the security sieve system.

Figure 3.7 illustrates how PO1 can be used with PO2. As discussed in Section 3.3, using multiple channels requires partitioning the data. In this case, the batch data list, which contains the data for multiple files, has to be divided among the multiple channels. Thus, the implementation of the splitter/binders will be different than those used in PO1. Details on the design and implementation of PO2 and PO2 with multiple channels are presented in Section 4.4 and Section 4.5, respectively.

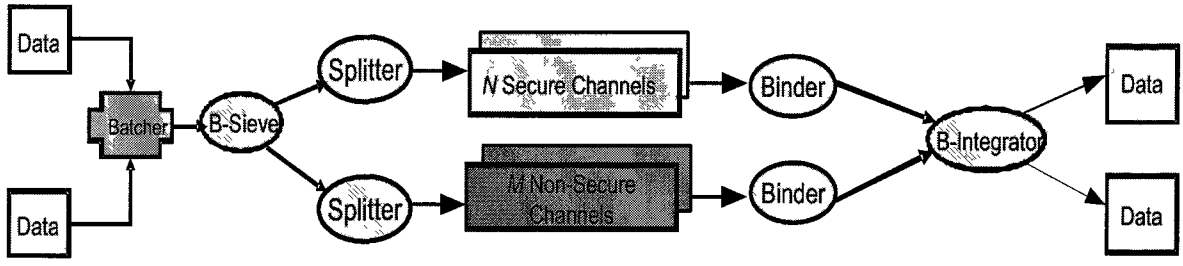


Figure 3.7: Combination of PO1 and PO2: a security sieve system using batching and multiple channels.

Chapter 4: Prototype Design and Implementation

This chapter describes the design and implementation of the prototype secure-only and security sieve systems. The design and implementation of the performance optimization techniques are also explained.

4.1 Secure-only System

In the secure-only system the client and server communicate using a single secure channel. A prototype secure-only system was implemented in Java using NetBeans IDE [27] version 6.9. An overview of the secure-only system is shown in Figure 4.1. This diagram shows the main classes used in the secure-only system. The secure-only client establishes a secure channel with the secure-only server using the `javax.net.ssl.SSLSocket` application programming interface (API) [28], provided by Java's Secure Socket Extension (JSSE) [29]. Both the secure-only client and server use `java.io.BufferedReader` [30] and `java.ioBufferedWriter` [31] objects to write/read data to/from the underlying sockets (as shown in Figure 4.1).

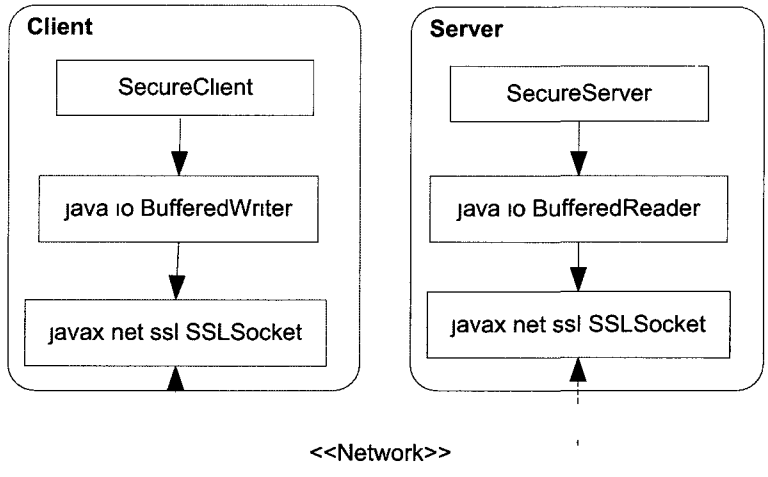


Figure 4.1: Secure-only system overview.

4.1.1 Secure-only Client

A class diagram showing the four versions of secure-only client is presented in Figure 4.2. The first version of the secure-only client is the basic single secure channel-based client. The other three versions of the secure-only client deploy the two performance optimization techniques. The suffix at the end of the class name (see Figure 4.2) identifies the version of client as follows:

1). Suffix *MC*: PO1: multiple channels, is deployed
2). Suffix *B*: PO2: batching of multiple files, is deployed.
3). Suffix *MCB*: Both PO1 and PO2 are used.

All four versions of the secure-only client use the `Utility` class. In terms of implementation, the difference between the different versions of the client lies in the implementation of the `start()` method and in the specific `Utility` methods invoked. This section focuses on the single channel-based client. The other versions of the client and the utility methods that they use are discussed in Section 4.3 and Section 4.4.

A secure-only client is created by invoking the `SecureClient()` constructor and specifying the following parameters: (1) server host name, (2) secure port number, and (3) the cipher suite to be used for the secure channel. Note that for the versions of the client where multiple channels are used, the number of secure channels needs to be specified as well. The `start()` method is used to run the client, and when invoked the client tries to connect to the server using the hostname and port number attributes. The `askForInput()` and `readInputFile()` utility methods are used to get the directory of the document from the user, and read the document that will be sent to the server, respectively. The `send()` method uses the specified `BufferedWriter` to write the supplied `String` to the underlying socket. The `Utility` class also specifies the key

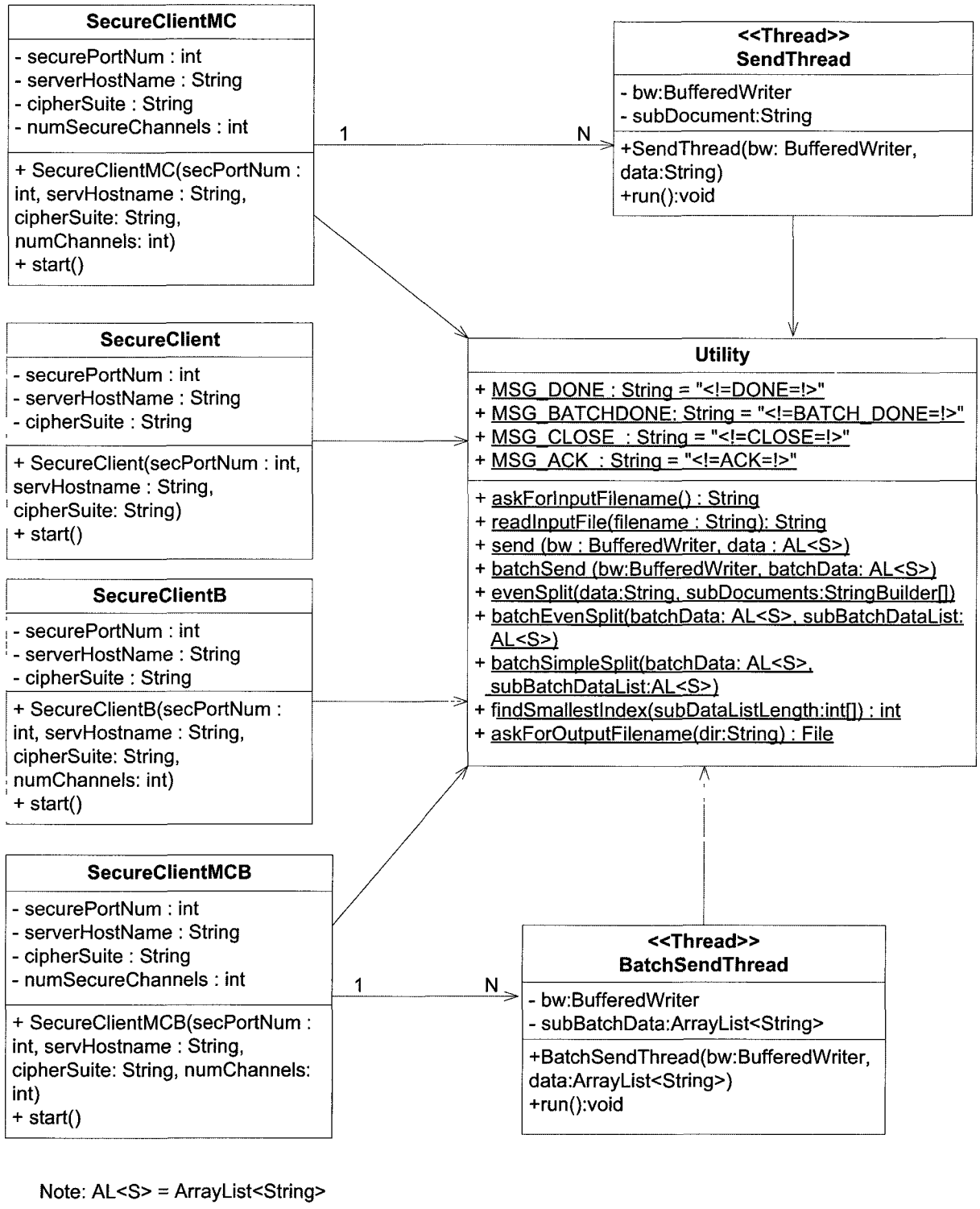


Figure 4.2: Class diagrams for all versions of the secure-only client.

messages and symbols exchanged between the client and server. It is assumed that these special messages are not present in the text of the document that is being transferred. The messages are used as follows:

- `MSG_DONE` informs the server that all the data for a file has been sent,
- `MSG_CLOSE` is sent when client is about to close its connection,
- `MSG_ACK` is the acknowledgement message.

Note that these same messages are also used in the security sieve system.

4.1.2 Secure-only Server

A class diagram of the secure-only server is shown in Figure 4.3. Like the client, there are four versions of the server, but they are not shown in this class diagram because they are very similar. This section discusses the single channel-based secure-only server and the utility methods it invokes. The utility methods invoked by the other versions of the server are discussed in Section 4.3 and Section 4.4.

The single channel-based secure-only server is created using the `SecureServer()` constructor and specifying port number of the secure server socket. Note that for the versions of the server where multiple channels are used, the number of secure channels needs to be specified as well. The server is started by invoking the `start()` method, which uses the JSSE `SSLServerSocketFactory` [32] and `SSLServerSocket` [33] APIs to create a SSL/TLS server socket (discussed in more detail in Section 4.2.3). After the server socket is created, the server waits for a connection request from the secure-only client. Once a connection is established, the server invokes the `receive()` method which uses the supplied `BufferedReader` to read data from the underlying socket. Data is read 8192 characters at a time, and is

appended to the supplied `dataRead` parameter. 8192 characters are read at a time because the client sends this number of characters at a time.

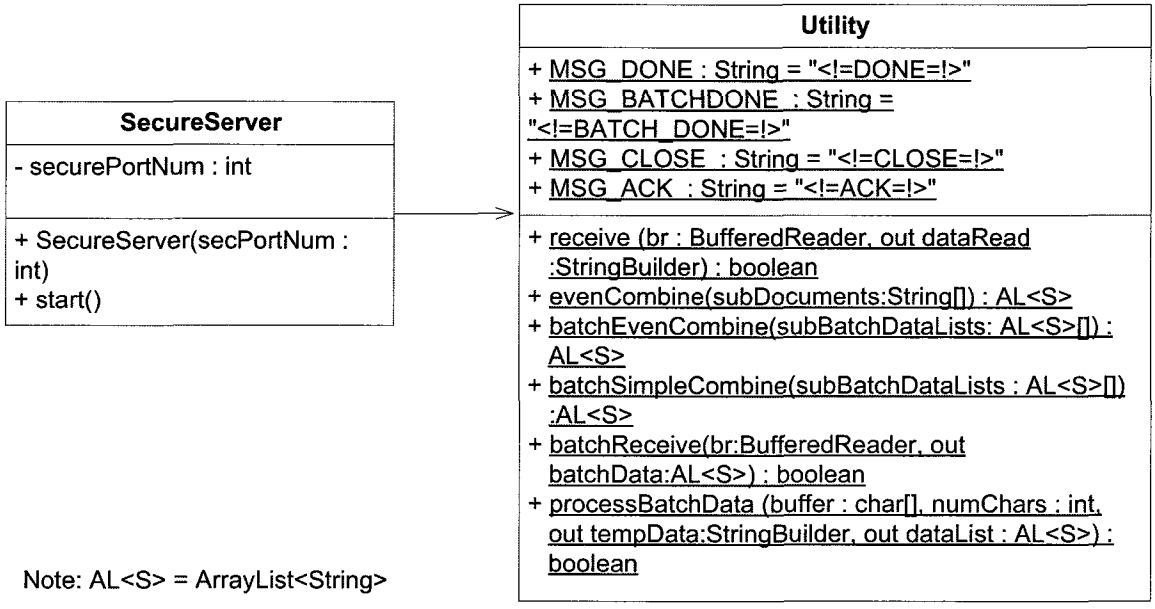


Figure 4.3: Class diagram of the basic version of the secure-only server.

4.1.3 Running the Secure-only System

Running a Java program that uses JSSE sockets is a little more complex compared to executing a normal Java program. First, a digital certificate needs to be created for the server, which can be done with the help of Java's `keytool` [34] command. `Keytool` is Java's key and certificate management utility that allows users to administer their own public/private key pairs and digital certificates. The created key pairs and certificates are stored in a `keystore` file. The following command: "`keytool -genkeypair -keystore ServerKeystore -keyalg RSA`" generates a RSA public-private key pair, and creates a X.509 digital certificate (discussed in Section 2.2.3) for the server. The keytool command also prompts the user for a keystore password and other certificate information. The certificate chain and private key are stored in a keystore file called

"ServerKeystore" on the user's home directory. To run the server, the following command is used:

- "`java -Djavax.net.ssl.keyStore= ServerKeystore -Djavax.net.ssl.keyStorePassword=123456 SecureServer`"

Note that the keystore file must be in the working directory of the `SecureServer` class file. The `SecureClient` is run the same way except with the class name "SecureServer" replaced with "SecureClient".

4.2 Security Sieve System

A prototype security sieve system was implemented in Java using NetBeans IDE [27] v6.9. A component diagram of the security sieve system is displayed in Figure 4.4. The non-secure and secure channels are established using a TCP socket, and a TLS/SSL socket, respectively. The TCP sockets are created using the `java.net.Socket` [35] API, whereas the SSL sockets are created with the `javax.net.ssl.SSLSocket` [36] API. As in the case with the secure-only system, the `java.io.BufferedReader` and `java.ioBufferedWriter` objects are used to read data from and write data to the underlying sockets.

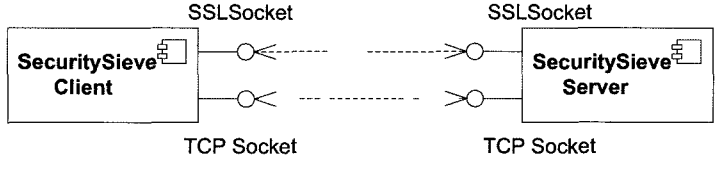


Figure 4.4: Security sieve system component diagram.

4.2.1 Security Sieve Client

A class diagram of the different versions of the security sieve client is shown in Figure 4.5. As in the case of the secure-only client, there are four versions of the security

sieve client. The different versions of the security sieve client are identified with the same suffixes, as explained in Section 4.1.1. The Utility class contains the common methods used by the different versions of the client, and specifies the key symbols and messages exchanged between the client and server. It is assumed that these messages and symbols are not present in the text of the document that is transferred. Note that the methods and attributes that have the same name, as described in Section 4.1.1, perform the same role. This section describes the single channel-based security sieve client and the utility methods it invokes. The other utility methods and symbols are discussed in Section 4.3 and Section 4.4. For the security sieve system, single channel means that one secure and one non-secure channel are used.

The single channel-based security sieve client is created by invoking the `SecuritySieveClient()` constructor and specifying the following parameters: (1) server host name, (2) secure port number, (3) non-secure port number, and (4) the cipher suite to be used for the secure channel. For the versions of the client where multiple channels are used, the number of secure and non-secure channels needs to be specified as well. The `sieve()` and `send()` methods (in the utility class) are used by the security sieve client to transfer data to the server. The sieve method implements the sieve algorithm (discussed in Section 3.1). The secure and non-secure data lists are implemented as `ArrayList` [37] objects (part of Java's Collections Framework). The send method writes the supplied sieved data list to the underlying socket using the socket's `BufferedWriter`. The client sends the secure and non-secure lists concurrently by using two `SendThread` objects. The send threads invoke the send method. More details on how the security sieve client sends data is discussed in Section 4.2.4.

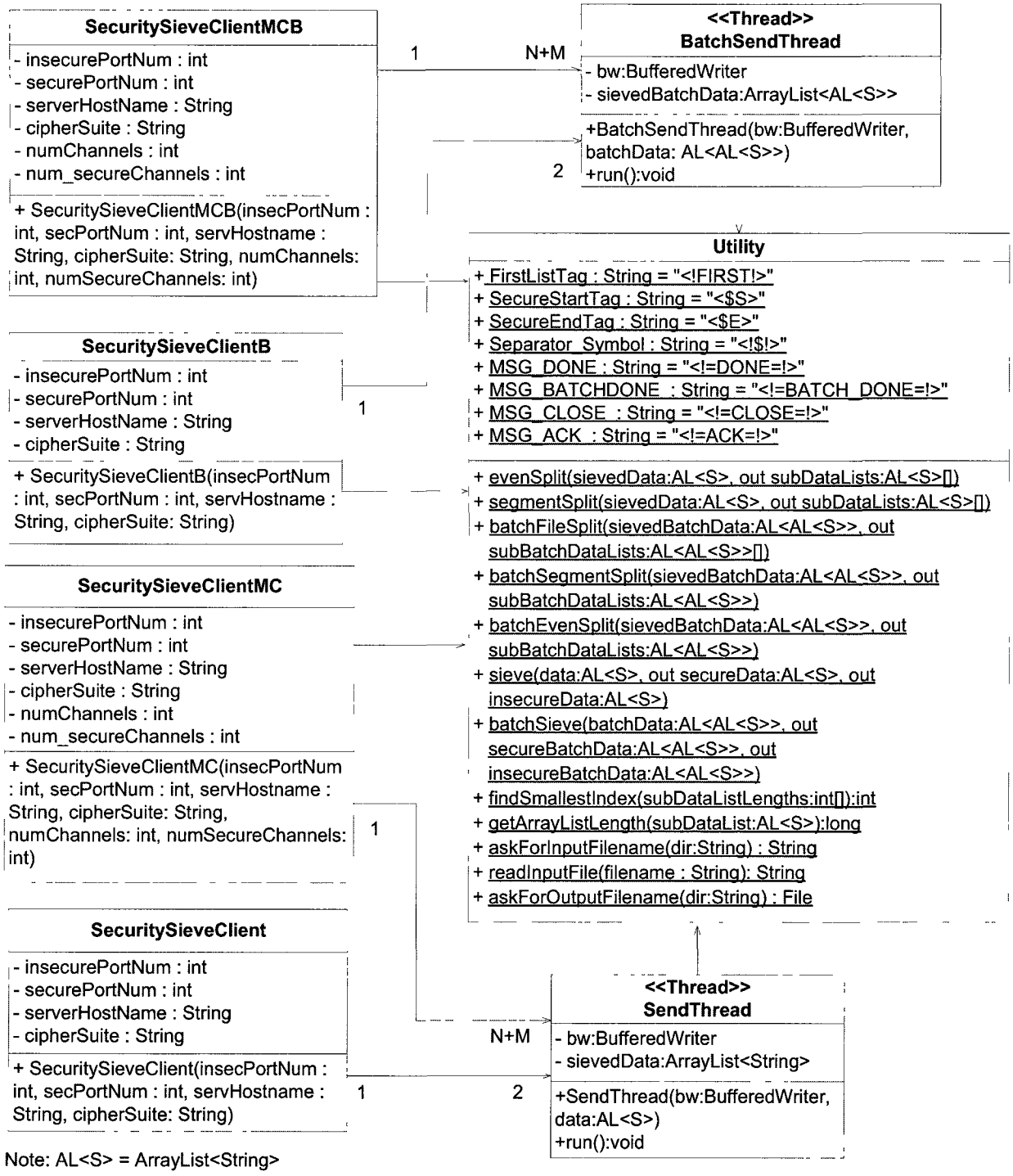


Figure 4.5: Class diagram for all versions of the security sieve client.

4.2.2 Security Sieve Server

Similar to the client, there are four different versions of the security sieve server, but they are not shown in the class diagram in Figure 4.6, since they are very similar. Each version of the security sieve server can communicate with the corresponding version of the security sieve client. Like the client, the server uses a `Utility` class that contains messages, symbols, and methods used by the different versions of the server. This section focuses on discussing the single channel-based security sieve server. The other utility methods and symbols are discussed in Section 4.3 and Section 4.4.

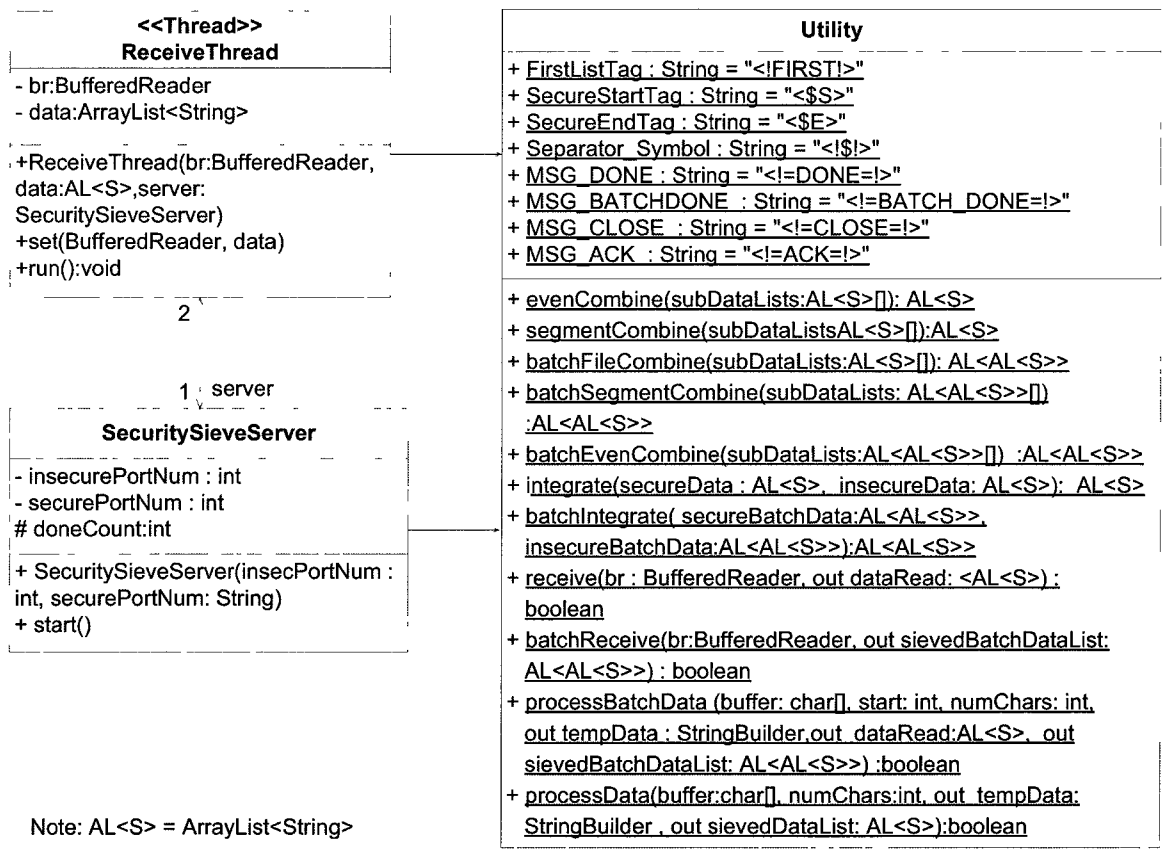


Figure 4.6: Class diagram of the basic security sieve server.

The security sieve server is created by invoking the constructor, which accepts two parameters: (1) the port number for the non-secure socket, and (2) the port number for the secure socket. Note that for the versions of the server where multiple channels are used, the number of secure and non-secure channels needs to be specified as well. The `start()` method is used to start the server, and when invoked the server creates the secure and non-secure sockets, and listens for client connection requests.

After a connection with a client is established, the server creates two `ReceiveThread` objects: one thread to read data from the non-secure channel, and one thread to read data from the secure channel. The thread objects invoke the `receive()` method, which reads data from a socket using the supplied `BufferedReader` object. All the data that is read is temporarily stored in a buffer until it is processed by the `processData()` method. After receiving `MSG_DONE`, which indicates that all the data has been sent, the thread increments the server's `doneCount` attribute. When the `doneCount` attribute equals two, the server knows that both threads have finished receiving data. The receive and process data methods are discussed in more detail in Section 4.2.4.

After receiving all the data, the `integrate()` method, which implements the integration algorithm (discussed in Section 3.2), is invoked. The integrate method returns an `ArrayList<String>` object instead of a `String` object because copying the data from the secure and non-secure data lists, and appending it to a `String` object was found to be quite slow, as discussed next.

Initial testing of the `integrate()` method revealed that when returning the integrated data as a `String` object, the time required to integrate the sieved data lists of a

10MB file (containing 5MB of secure data and 5MB of non-secure data) was approximately 87 ms. In comparison, when the integrate method returns the integrated data in an `ArrayList<String>` object, it only takes 0.03 ms to integrate the sieved data lists. Returning an `ArrayList` is faster because the `String` objects stored in the sieved data lists do not need to be copied; instead, the object's references are manipulated. The integrate method in the security sieve prototype copies the references of the `String` object contained in the sieved data lists and arranges them in proper order in a new `ArrayList<String>` object.

4.2.3 Connection Establishment

A sequence diagram of the security sieve server's socket creation is shown in Figure 4.7. First, the server retrieves an `SSLServerSocketFactory` [32] object by invoking the static method `SSLServerSocketFactory.getDefault()`. The server then uses this object to create a `SSLServerSocket` [33] object by invoking the `createServerSocket()` method with the `securePortNum` private attribute. Next, the non-secure (TCP) socket is created by invoking the `ServerSocket()` constructor (provided by Java's `ServerSocket` API [38]) with the `insecurePortNum` attribute. The server then waits for incoming client connection requests by invoking the socket objects' `accept()` method, which blocks the server until a client establishes a connection with the server. The secure and non-secure channels between the client and the server are now established. The server then retrieves the I/O streams from the sockets using the `getInputStream()` and `getOutputStream()` methods, and creates `BufferedReader` and `BufferedWriter` objects (part of Java's I/O package).

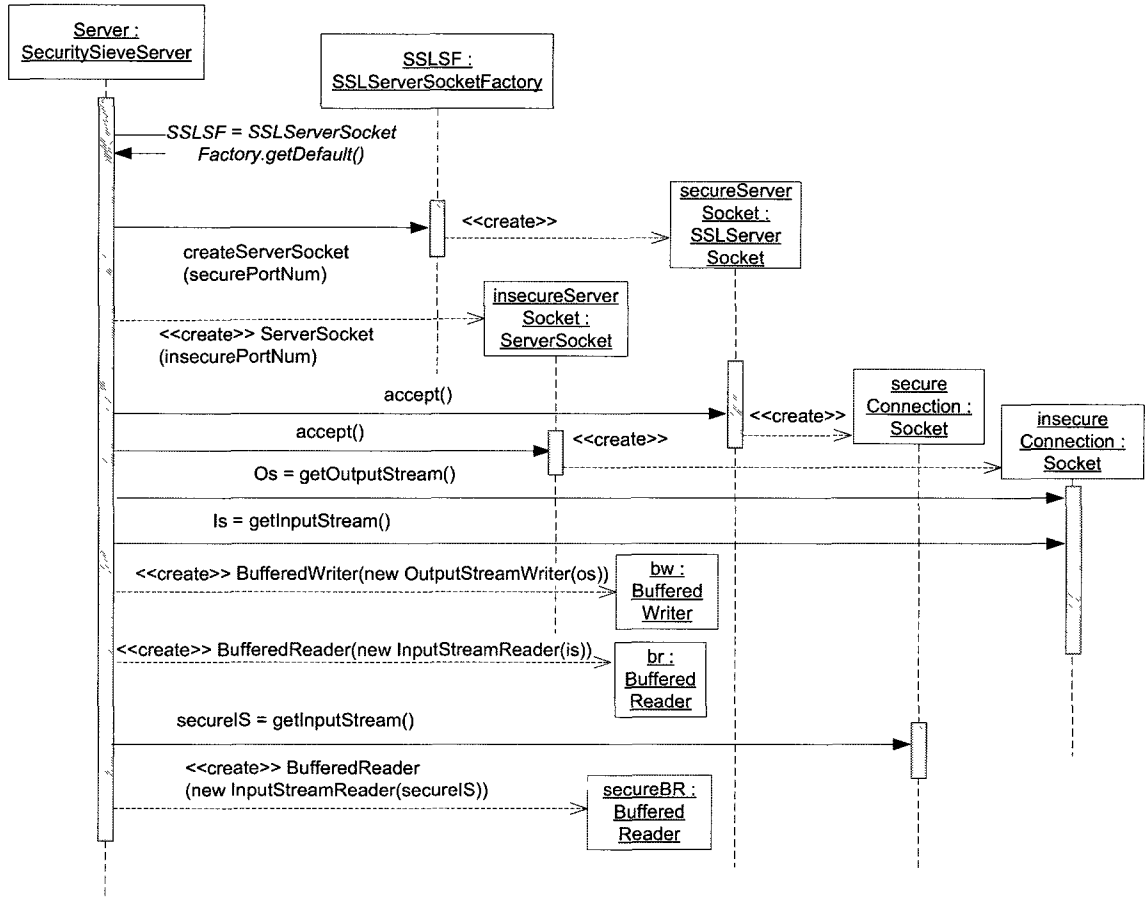


Figure 4.7: Sequence diagram of the security sieve server's socket creation.

The sequence diagram of the client's connection establishment is presented in Figure 4.8. First, the client attempts to establish a non-secure channel with the server by creating a `Socket` [35] object using the server host name and `insecurePortNum` attribute. The non-secure channel is established after the server accepts the connection. The client then creates a `SSLSocketFactory` object using Java's `SSLSocketFactory` [39] API. Using this object the client tries to establish a secure channel with the server by invoking the `createSocket()` method with the hostname and secure port number attributes as the parameters. Once the server accepts the connection, the client sets the cipher suite to be used, and starts the SSL/TLS handshake. After both channels are

established, the client retrieves the I/O streams (using the `getInputStream()` and `getOutputStream()` methods) from the secure and non-secure sockets, and creates the `BufferedReader` and `BufferedWriter` objects.

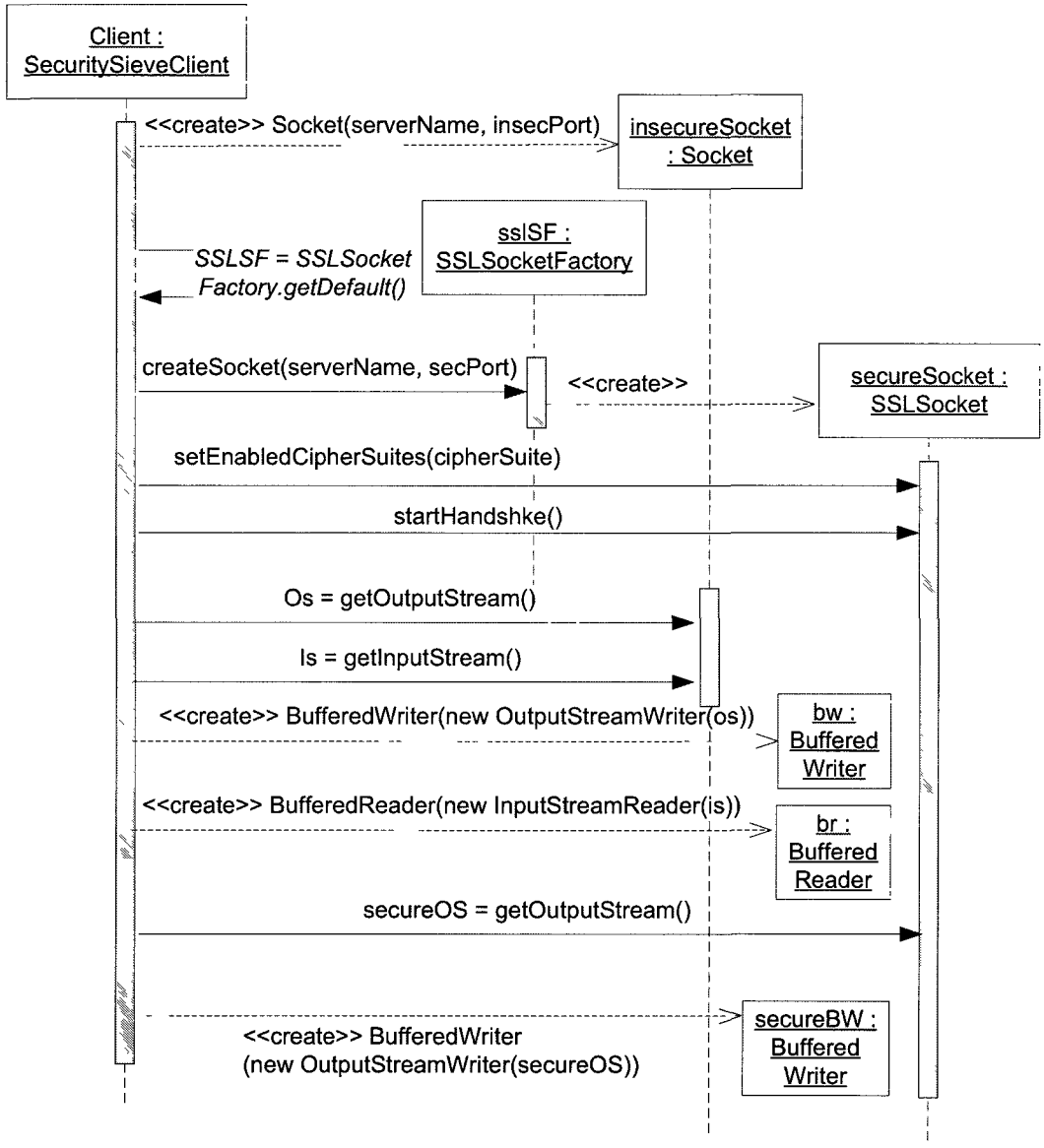


Figure 4.8: Sequence diagram of the security sieve client connection establishment.

4.2.4 Sending and Receiving Data

In the security sieve system, sending/receiving data involves using the socket's buffered writers/ buffered reader objects to write/read the secure and non-secure data lists (implemented as `ArrayList<String>` objects). As discussed in Section 4.2.1 and Section 4.2.2, sending and receiving data are completed by the client's send threads and server's receive threads. The send and receive threads invoke the `send()` and `receive()` methods, respectively, to accomplish their tasks. The client and server create the send thread and receive thread objects using their respective constructors (refer to Figure 4.5 and Figure 4.6) The `SendThread` class has two attributes: `bw` (a `BufferedWriter` object), and `data` (an `ArrayList<String>` object). The send thread writes the data stored in the `data` attribute to the underlying socket using the `bw` attribute. Similarly, the `ReceiveThread` class has two attributes: `br` (a `BufferedReader` object), and `data` (an `ArrayList<String>` object). The receive thread uses the `br` attribute to read data from the underlying socket, and stores the received data in the `data` attribute.

A sequence diagram of the send and receive methods is shown in Figure 4.9. In the send method, the supplied sieved data list is sent by writing each `String` object in the list to the server's socket, followed by sending a *Separator Symbol*, `<!$!>`. Note that each `String` object in the sieved data list is referred to a *text segment*. The purpose of the separator symbol is to segregate the individual text segments so that the receiver can distinguish them. The `MSG_DONE` symbol is sent to inform the receiver that all the data has been written. The individual text segments are sent, instead of sending the entire

`ArrayList<String>` object because the `BufferedWriter` and `BufferedReader` objects cannot write and read `ArrayList` objects.

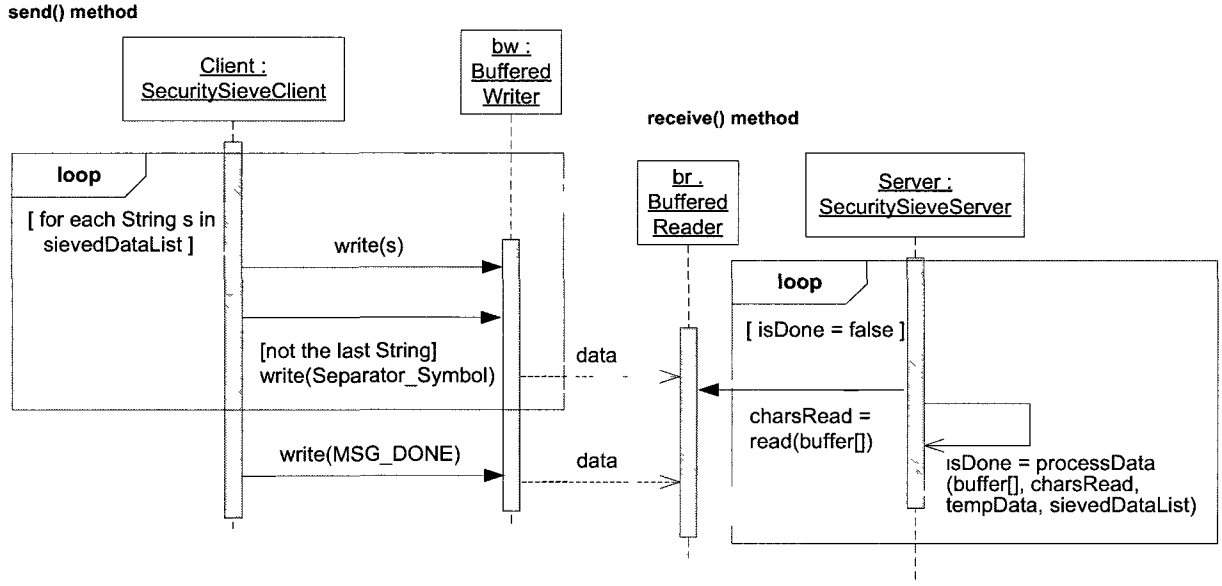


Figure 4.9: Sequence diagram of the send and receive methods.

The receive method reads 8192 characters from the channel at a time using the socket's `BufferedReader` object. Note that 8192 characters are read at a time because that is the number of characters that the client sends at a time. The text segment that is read is temporarily stored in a buffer before being passed to the `processData()` method (refer to security sieve server's utility class in Figure 4.6). This method continually stores the text segments in a temporary `String` variable, called `tempData`, until a separator symbol is received. At this point, the data stored in the `tempData` variable, excluding the separator symbol and the data after the separator symbol, is added to the supplied `sievedDataList` parameter. Data is only added to the supplied sieved data list when an entire text segment has been read. The `processData()` method returns true if `MSG_DONE` is received; otherwise, false is returned and the receive method

continues reading more text segments. Together, the `receive()` and `processData()` methods construct a local `ArrayList<String>` object that is identical to the original `ArrayList<String>` object that the client sends.

4.2.5 Example of a File Transfer

A sequence diagram showing a sample file transfer between a security sieve client and security sieve server is shown in Figure 4.10 . Note that the details of the `send()` and `receive()` methods were discussed in Section 4.2.4. Once the secure and non-secure channels between the server and client have been established, the client invokes the `askForInputFilename()` method to get input from the user. The user then enters the name of the file they want to send, and the client invokes the `readInputFile()` method to read the user's file. The `sieve()` method is then invoked to separate the classified and non-classified information from the user's marked file. Next, the client creates two `SendThread` objects to send the secure and non-secure data lists. Lastly, the client invokes the non-secure channel's `BufferedReader read()` method to wait for the server's acknowledgment message. The read method blocks until the data is received.

On the server side the following occurs. After establishing the secure and non-secure channels with the client, the server creates two `ReceiveThread` objects: one thread to read data from the non-secure channel, and the other thread to read data from the secure channel. These threads invoke the `receive()` method, which reads the data from the channels and constructs local versions of the non-secure and secure data lists. After all the data has been received, the server sends an acknowledgment message to the client via the non-secure channel. Lastly, the server invokes the `integrate()` method, which recombines the sieved data lists, and returns the integrated data.

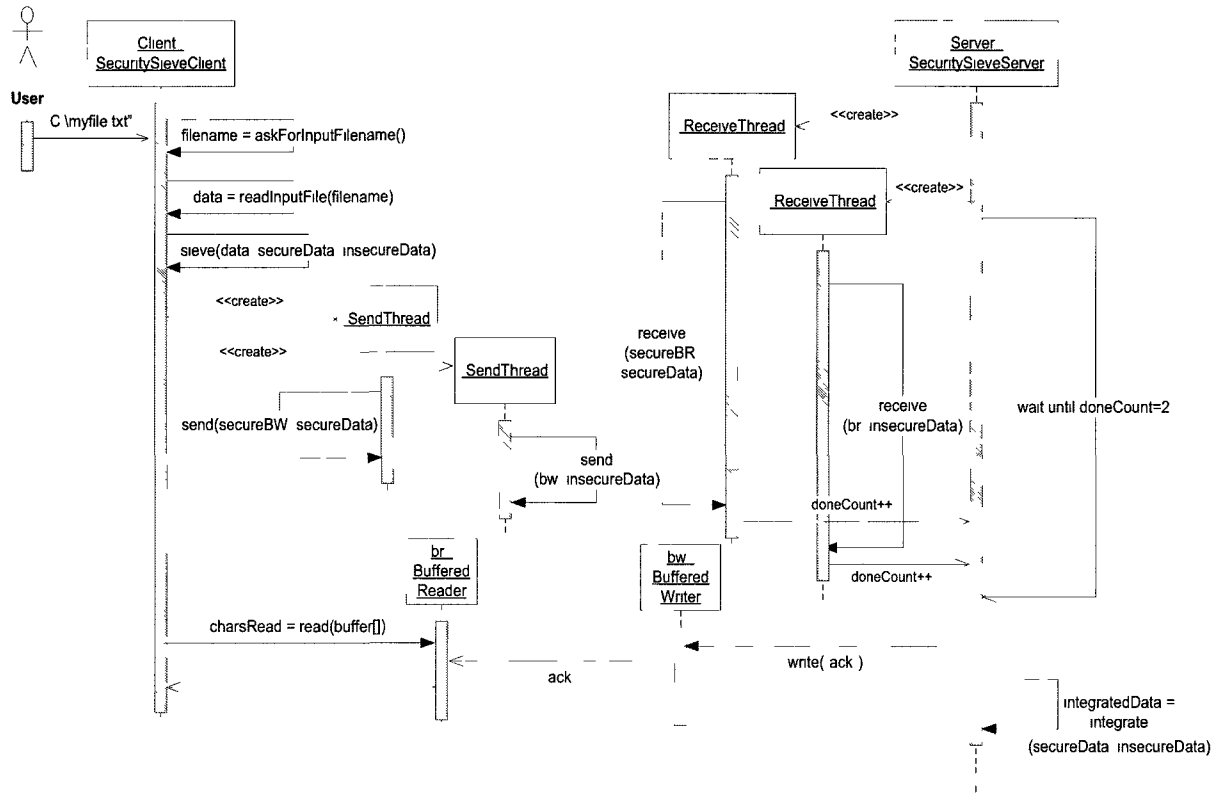


Figure 4.10: Sequence diagram showing an example file transfer in the security sieve system.

4.3 Performance Optimization 1: Multiple Channels

As shown in Figure 3.5a, adding multiple channels requires a splitter and binder. The functionality of the splitter and the binder was realized by developing a pair of *split/combine algorithms*, and implementing these algorithms as methods in the `Utility` class. The secure and non-secure channel connection establishment is completed in a manner similar to what was discussed in Section 4.2.4, but just repeated for each additional channel. In both the secure-only and security sieve systems, threads are used to send data to, and receive data from the channels. Each channel is responsible for handling a portion of the data in the document to be transferred. Both the server and client assign a

particular thread to service a channel. Thus, if there are three channels, the client has three `SendThread` objects and similarly the server has three `ReceiveThread` objects.

Two pairs of split/combine algorithms were developed: (1) *even* split/combine (ES) and (2) *segment* split/combine (SS). The SS algorithm, which is exclusive to the security sieve system, divides the sieved data lists by assigning each text segment (which is a `String` object in the sieve data list) to a separate channel. Conversely, the ES algorithm divides each text segment among the channels. Note that the ES algorithm used by the secure-only client divides all the data in the document at one time. These algorithms are explained in more detail in Section 4.3.3 and Section 4.3.4. But first, an overview of changes made to the secure-only and security sieve systems are provided.

4.3.1 Secure-only System deploying PO1

The split algorithm partitions the original document to be transferred into N sub-documents, and stores each of the N sub-documents in a `String` array, called the `subDocumentsArray`. Recall that N is the number of secure channels that the system uses to send data. After dividing the data, the split algorithm returns the sub-documents array to the client. When the client creates its `SendThread` objects, it passes a `String` object (taken from the sub-documents array) to the send thread's constructor. The secure-only system's `SendThread` constructor (see Figure 4.2) accepts two parameters: `bw` (a `BufferedWriter` object) and `subDocument` (a `String` object). The send thread uses the supplied buffered writer to write the `String` to the underlying socket.

At the receiving end, the server also has a local `subDocumentsArray`. Each `String` reference in the server's sub-documents array is supplied to one of the server's `ReceiveThreads` via the constructor. Once receiving is complete the server's sub-

document array, which now contains the data of the N sub-documents, is passed to the combine algorithm. The combine algorithm restores the original document by recovering the sub-documents from the local sub-documents array in the same order in which the split algorithm divided the original document.

4.3.2 Secure Sieve System deploying PO1

In the case of the security sieve system, the secure and non-secure data lists have to be divided separately, as shown in Figure 3.5b. Recall that the sieved data lists are `ArrayList<String>` objects. As such, the splitting/combining algorithms partition data stored in `ArrayList<String>` objects, and not in a single `String` as in the secure-only system. The `ArrayList<String>` objects are divided into *sub-data lists*, which contain a portion of the data contained in the original sieved data list. Note that sub-data lists and are also implemented as `ArrayList<String>` objects as well. The secure data list is divided into N sub-data lists. Likewise, the non-secure list is divided into M sub-data lists. Recall that N is the number of secure channels, and M is the number of non-secure channels. The N secure sub-data lists and M non-secure sub-data lists are stored in two variables: `secureSubDataListsArray` and `nonSecureSubDataListsArray`, respectively. These variables, which are collectively referred to as the *sub-data list arrays*, are arrays that have `ArrayList<String>` references. The split algorithm returns the sub-data list arrays to the client after partitioning the data. The client then assigns a sub-data list to each of its `SendThreads` to transfer. This is done by passing an `ArrayList<String>` reference, which is taken from the sub-data list arrays, to the send thread's constructor when it is being created.

On the receiving side, the server also has its own local sub-data list arrays. The `ArrayList<String>` references in these local sub-data list arrays are supplied to each of the server's `ReceiveThreads` via the `data` parameter in the receive thread's constructor. Each receive thread is responsible for reading from the channel the server assigns to it and storing the received data in the `data` attribute it is assigned. Using this approach, the server is able to know which one of its receive threads is reading from which channel, and where the received data is stored. Once receiving is complete, the secure sub-data lists array is supplied to the combine algorithm, which recombines the data and restores the original secure data list. Similarly, the non-secure sub-data lists array is supplied to the combine algorithm, which reconstructs the non-secure data list. The combine algorithm recovers the data contained in the supplied sub-data lists array in the same order in which the data was divided by the split algorithm.

4.3.3 Even Split (ES) Algorithm

A description of the ES algorithm that the security sieve client uses is presented next. Note that the `String` objects in the sieved data list are processed in sequence.

1). Remove a `String` object from the sieved data list to be divided, and get the length of the `String`. $Set\ \ c = \left\lfloor \frac{length\ of\ String}{length\ of\ \ sub-data\ lists\ array} \right\rfloor$.

2). Add c characters from that `String` to all the sub-data lists except the last one.

3). Add the remaining characters from the `String` to the last sub-data list.

4). If there are still `String` objects in the data list, go back to Step 1; otherwise, exit.

If a `String` cannot be evenly divided among the sub-data list then the last sub-data list will get the extra characters.

An example of the security sieve client using the ES algorithm to divide the non-secure data list among three channels is shown in Figure 4.11. Note that dividing the secure data list would be done in a similar manner. Recall that both the non-secure data list and sub-data lists are implemented as `ArrayList<String>` objects. Each number shown in the boxes of the diagram represents the number of characters in the text segment. The sequence of arrows in the diagram identifies the order in which the ES algorithm partitions the non-secure data list. A walkthrough of the example shown Figure 4.11 is presented next. First, the first `String` in the non-secure data list, which is 50 characters long, is divided into three parts (see arrow 1). Since the length of the non-secure sub-data lists array is 3, c equals 16. The first 16 characters of the first `String` are added to the first sub-data list. The next 16 characters of the first `String` are added to the second sub-data list, and the remaining 18 characters are added to the last sub-data list. This sequence of operations is repeated for each `String` in the non-secure data list (see arrows 2 through 7).

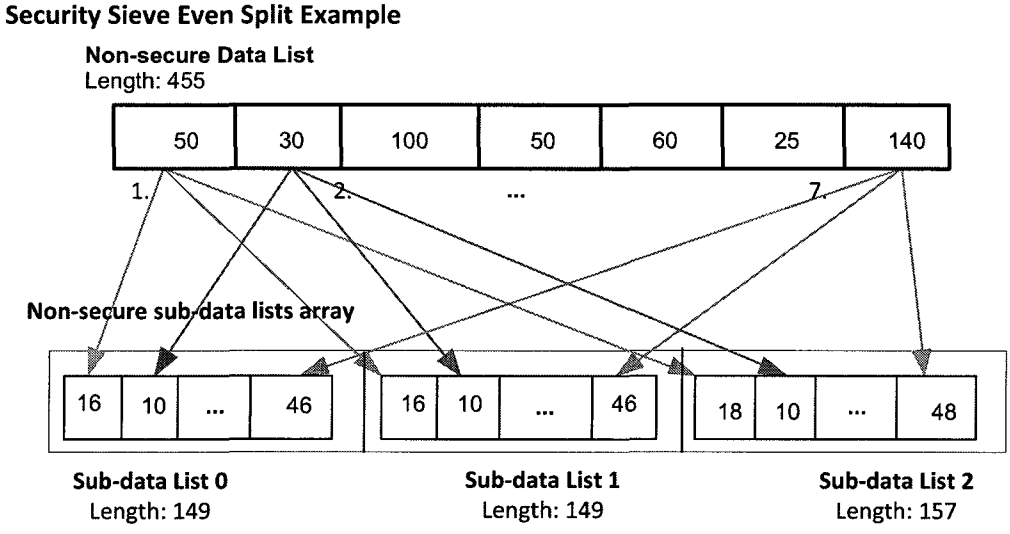


Figure 4.11: Illustration of the security sieve client's ES algorithm when three channels are used.

The ES algorithm used by the secure-only client is similar to the security sieve client's ES algorithm. The main difference is that the data to be divided is stored in a single `String` variable, which contains the text for an entire file. In addition, the secure-only client only needs to execute the ES algorithm once whereas the security sieve client executes the algorithm twice: once for partitioning the secure data list and once for partitioning the non-secure data list. Recall that the secure-only client uses a `String` array called the `subDocumentsArray` to store the divided data (refer to Section 4.3.1). The ES algorithm used by the secure-only client is presented next.

1). Get the number of characters stored in the `String` variable.
Set $c = \left\lfloor \frac{length\ of\ String\ variable}{length\ of\ the\ subDocuments\ array} \right\rfloor$.

2). Add c characters from the `String` variable to all the entries in the `subDocumentsArray` except the last one.

3). Add the remaining characters from the `String` variable to the last entry of the `subDocumentsArray`.

For both the secure-only and security sieve systems, the utility methods: `evenSplit()` and `evenCombine()` implement the ES algorithm and the corresponding combine algorithm.

4.3.4 Segment Split (SS) Algorithm

Recall that the SS algorithm, which is exclusive to the security sieve system, divides the sieved data lists by assigning each text segment (which is a `String` object in the sieve data list) to a separate channel. The utility methods, `segmentSplit()` and `segmentCombine()`, implement the segment split and combine algorithms, respectively (see Figure 4.5 and Figure 4.6).

The SS algorithm keeps track of the total number of characters contained in each sub-data list, in an integer array called `subDataListLengths` (SDLL). The length of the first sub-data list is stored in the first entry of the SDLL. Similarly, the length of the second sub-data list is stored in the second entry of the SDLL, and so on. Each time a text segment is added to the sub-data list, the length of the text segment is also added to the sub-data list's entry in the SDLL array. This operation is referred to as "updating the entry in the SDLL array".

The SS algorithm needs to keep track of the size of each sub-data list, so that it can determine the sub-data list that has the least number of characters. To find the index of the sub-data list with the least number of characters, the `findSmallestIndex()` utility method is invoked. This method simply finds the smallest number in the SDLL array, and returns the position of the smallest number. For example, if the first sub-data list is the smallest list, zero would be returned, and the SS algorithm would know that the next `String` should be added to the first sub-data list. The SS algorithm also uses a `String` variable, called `order`, to keep track of which sub-data list each of the `String` objects is added to. This `order` variable is used by the combine algorithm to reconstruct the original data list. The steps for the SS algorithm are presented next.

1). If the sieved data list is empty, go to Step 5. Otherwise, remove a `String` object from the sieved data list and add it to the first empty sub-data list. Update the entry in the SDLL array.

2). If all sub-data lists have one `String` object, continue to Step 3. Otherwise, go back to Step 1.

3). If there are still `String` objects in the sieved data list, continue to Step 4; otherwise, go to Step 5.

4). Remove the next `String`, and add it to the smallest sub-data list. Append the sub-data list number to the `order` variable. Update the chosen sub-data list's entry in the SDLL array. Go back to Step 3.

5). Add the order variable to the first sub-data list. Exit.

An example of the security sieve client using the SS algorithm to divide the secure data list among three channels is shown in Figure 4.12. The numbers shown in the boxes of the diagram represent the length (or number of characters) of each `String`. The sequence of arrows in the diagram identifies the order in which the SS algorithm partitions the secure data list. Arrows 1 to 3 in the diagram correspond to Steps 1 and 2 of the SS algorithm. Arrows 4 to 7 are showing Steps 3 and 4 of the SS algorithm. The last step of the SS algorithm is shown by arrow 8.

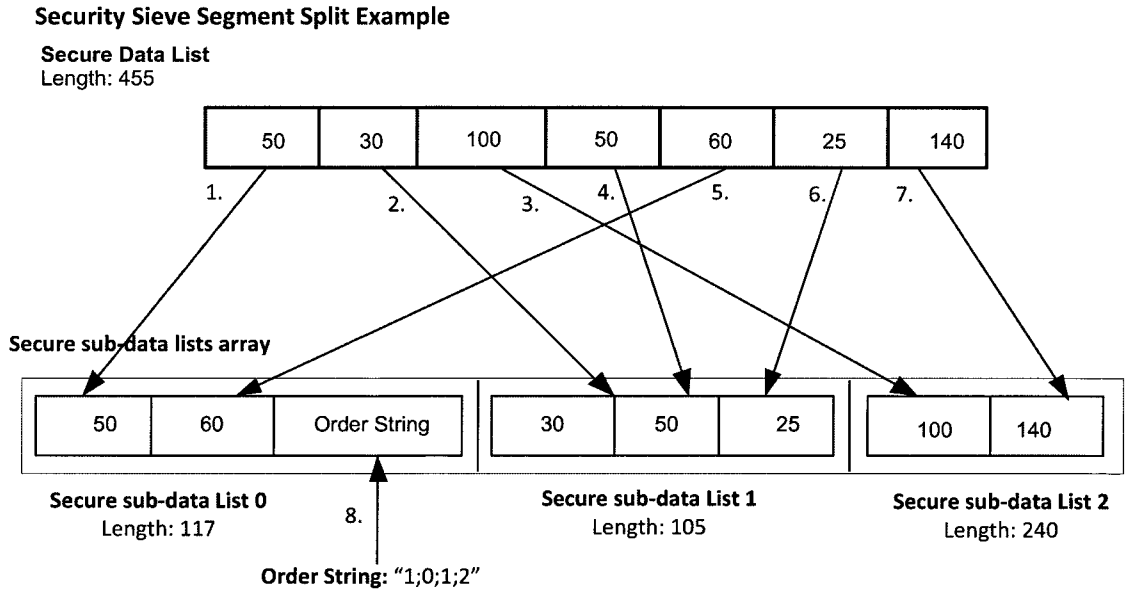


Figure 4.12: Illustration of the SS algorithm when three channels are used.

Compared to the ES algorithm, the SS algorithm is less complex since not each text segment in the data list has to be divided. However, depending on the size of the text

segments, the SS algorithm may not always partition the data list evenly among the channels (as shown in Figure 4.12). The SS algorithm also requires keeping track of the size of each sub-data list, and which the sub-data list each text segment is added too, but this does not incur too much overhead.

4.4 Performance Optimization 2: Batching of Multiple Files

The new additions required for adding PO2 to the security sieve and secure-only systems are explained next (refer to the client class diagrams in Figure 4.2 and Figure 4.5). An `ArrayList<String>` object named `batchDataList` is used to store data for multiple documents. Each entry in this batch data list contains data for an individual document. To send and receive the batch data, the `batchSend()` and `batchReceive()` methods are used. Once all the files in the batch data lists have been sent, the client sends a `MSG_BATCHDONE` symbol to the server. The details of how the secure-only and security sieve systems achieve batching are discussed next.

4.4.1 Secure-only System deploying PO2

The secure-only system's `batchSend()` and `batchReceive()` methods are very similar to the `send()` and `receive()` methods used by the single channel-based security sieve system (as discussed in Section 4.2.4). Both sets of methods are sending and receiving data contained in `ArrayList<String>` objects. The batch receive method invokes a `batchProcessData()` method which is very similar to the security sieve client's `processData()` method. The only difference is that the symbol used to segregate the individual files is a `MSG_DONE` symbol instead of the `Separator_Symbol`.

4.4.2 Security Sieve System deploying PO2

To sieve the batch data list, the `batchSieve()` method is used. The batch sieve method goes through each file in the batch data list and calls the `sieve()` method to individually sieve each document in the list. The secure and non-secure data lists obtained after sieving a document are added to two master lists called the `secureBatchDataList` and `nonsecureBatchDataList`, respectively. These master lists (collectively referred to as the *sieved batch data lists*) are implemented as `ArrayList` objects with `ArrayList<String>` references. The client sends the secure and non-secure batch data lists to the server by using two `BatchSendThread` objects (refer to the client class diagram in Figure 4.5). These thread objects invoke the `batchSend()` method, which sends each `ArrayList<String>` object in the sieved batch data lists by invoking the `send()` method (which was discussed in Section 4.2.4). After sending all the data, the `MSG_BATCHDONE` symbol is sent.

On the server-side, two `BatchReceiveThread` objects are used to read the sieved batch data lists sent by the client. These thread objects invoke two methods: `batchReceive()` and `processBatchData()`. Note that the server has its own sieved batch data lists, which are implemented as `ArrayList<ArrayList<String>>` objects, just like the corresponding client variables. The server supplies these sieved batch data lists to the batch receive method. The main difference between the new methods and the `receive()` and `processData()` methods (discussed in Section 4.2.4), is that the new methods have to handle reading multiple `MSG_DONE` symbols, along with multiple separator symbols (`<!$!>`), to re-create the sieved batch data lists. Recall that the `send()` method sends `MSG_DONE` to indicate that all the data in an

`ArrayList<String>` object (i.e. data for a single file) has been sent. Therefore, for each entry in the sieved batch data lists that is sent, a `MSG_DONE` symbol is also sent. All the received data for a single file is stored in an `ArrayList<String>` object named `dataRead` until `MSG_DONE` is received, at which point, the `dataRead` variable is added to the supplied `sievedBatchDataList` parameter. A new `ArrayList<String>` object is created to store the data for the next file in the batch. The batch receive method continues reading data until the `MSG_BATCHDONE` symbol is received.

To integrate the sieved batch data lists, the server invokes the `batchIntegrate()` method. The batch integrate method invokes the `integrate()` method (discussed in Section 4.2.2) to integrate each pair of entries in the sieved batch data lists. For example, the first `ArrayList<String>` objects in both the `secureBatchDataList` and `nonsecureBatchDataList` variables, which are the secure and non-secure data lists for the first document in the batch, are re-assembled using the `integrate()` method. The integrated data returned by the integrate method is stored in a master list called the *integrated batch data list* (implemented as an `ArrayList<ArrayList<String>>` object). After all the sieved data lists have been integrated, the batch integrate method returns the integrated batch data list to the server.

4.5 PO1 and PO2: Batching with Multiple Channels

Deploying PO1 and PO2 together requires the changes discussed in both Section 4.3 and Section 4.4. Deploying PO2 with multiple channels requires dividing the batch data list into multiple sub-batch data lists. Recall that the security sieve system partitions the sieved batch data lists separately (as shown in Figure 3.7), whereas, the secure-only client partitions the batch data list directly. As such, the splitting/combining algorithms

need to be able to divide data stored in `ArrayList<String>` objects for the secure-only system, and `ArrayList <ArrayList<String>>` objects for the security sieve system. In both the secure-only and security sieve systems, `BatchSendThread` and `BatchReceiveThread` objects are used to send/receive sub-batch data lists to/from each channel.

The functionality of the splitter and binder for this system was realized by deriving three sets of batch split/combine algorithms: (1) *Batch File Split/Combine* (BFS), (2) *Batch Even Split/Combine* (BES), and (3) *Batch Segment Split/Combine* (BSS). The BES and BSS algorithms are extensions of the ES and SS algorithms discussed in Section 4.3.3 and Section 4.3.4. The BFS algorithm used by the secure-only client, partitions the batch data list by assigning each channel an entire file to send. Similarly, the BFS algorithm used by the security sieve client partitions a sieved batch data list by assigning each channel a sieved data list to send. More details on these algorithms for both the secure-only and security sieve systems are described next.

4.5.1 Batch File Split (BFS) Algorithm

The BFS algorithm used by the secure-only client goes through each `String` object in the `batchDataList` and assigns the `String` to the sub-data list with the least number of characters. Recall that the `batchDataList` variable is an `ArrayList<String>` object, and each `String` in this variable contains the data for a single document. In terms of implementation, the secure-only system's BFS algorithm is identical to the SS algorithm (discussed in Section 4.3.4).

The BFS algorithm used by the security sieve client is similar to the SS algorithm (discussed in Section 4.3.4). The only difference is that the unit of data that is divided

among the channels is an entire sieved data list instead of a single text segment. Recall that the sieved data lists are implemented as `ArrayList<String>` objects. To get the length of an `ArrayList<String>` object, the length of each `String` in the array list needs to be added together. This calculation is performed by the `getArrayListLength()` method, which is implemented in the security sieve client's utility class (see Figure 4.5).

An illustration of the security sieve client using the BFS algorithm to divide the secure batch data list between two channels is shown in Figure 4.13. The (secure or non-secure) *sub-batch data lists array* and *sub-batch data lists* shown in the illustration perform the same role as the (secure or non-secure) sub-data lists array and sub-data lists, respectively (as discussed in Section 4.3 for PO1). The only difference is that the sub-batch data lists contain the data for multiple files, not just as single file. As such the *sub-batch data lists array* is an array containing `ArrayList<ArrayList<String>` references.

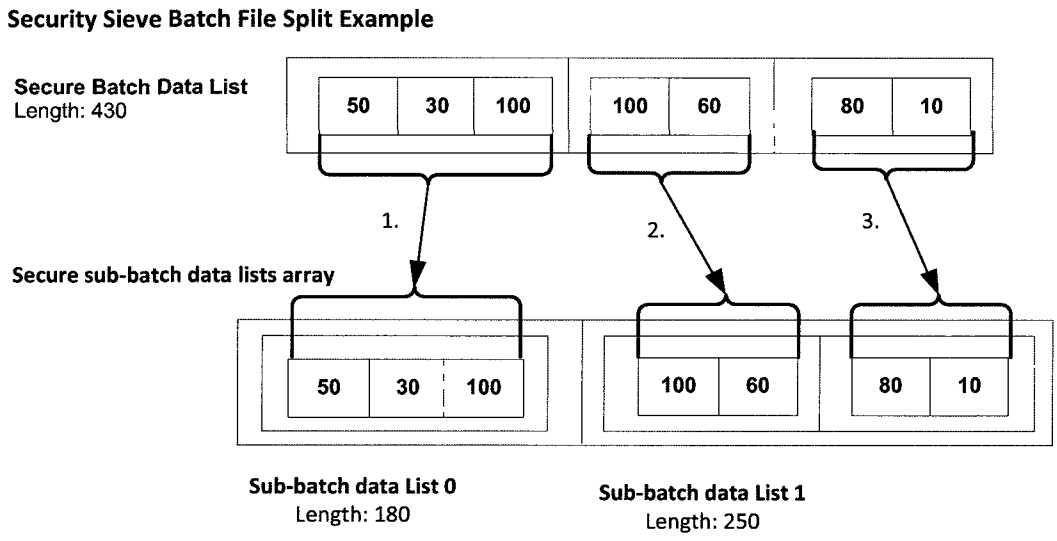


Figure 4.13: Illustration of the security sieve client's BFS algorithm when two channels are used.

A walk-through of Figure 4.13 is presented next. The numbers shown in the boxes of the diagram represent the length (or number of characters) of each `String`. The sequence of arrows in the diagram identifies the order in which the BFS algorithm partitions the secure batch data list. First, each sub-batch data list is assigned a secure data list (see arrows 1 and 2). The last secure data list (see arrow 3), is assigned to the sub-batch data list with the least number of characters, which happens to be the second sub-batch data list. The BFS algorithm and corresponding combine algorithm are implemented in the `batchFileSplit()` and `batchFileCombine()` methods in the utility classes of the security sieve and secure-only systems.

4.5.2 Batch Even Split (BES) Algorithm

The BES algorithm used by the security sieve client goes through the sieved batch data list, and uses the ES algorithm (discussed in Section 4.3.3) to divide each of the sieved data lists contained in the sieved batch data list. The steps for the BES algorithm used by the security sieve system are presented next.

1). Remove an `ArrayList<String>` object from the sieved batch data list and store it in a variable called *s*.
2). Use the (security sieve) ES algorithm (discussed in Section 4.3.3) to divide *s*.
3). If the sieved batch data list is not empty, go back to Step 1; otherwise, exit.

An illustration of the security sieve client dividing the non-secure batch data list between two channels using the BES algorithm is shown in Figure 4.14. Note that the numbers shown in the boxes of the diagram represents the length of each `String`. The sequence of arrows in the diagram identifies the order in which the BES algorithm partitions the non-secure batch data list. The algorithm starts off by dividing the first non-secure data list (see arrows 1 to 3), which is the list that contains the three `String`

objects with lengths equal to 50, 30, and 100 characters. This non-secure data list and the first `ArrayList<String>` objects from the two sub-batch data lists are supplied to the `evenSplit()` method (refer to the Utility class in Figure 4.5). In this example, the even split method divides each `String` in the non-secure data list in half, and stores the divided data in the two supplied sub-data lists. For example, the first string, which is 50 characters long, is divided into two strings, each with 25 characters (see arrow 1). Similarly, arrows 2 and 3 show the remaining text segments in the non-secure data list being divided in half. The details on how the ES algorithm partitions a sieved data list were described in Section 4.3.3. The two remaining non-secure data lists are also partitioned by the ES algorithm in a similar fashion as the first non-secure data list.

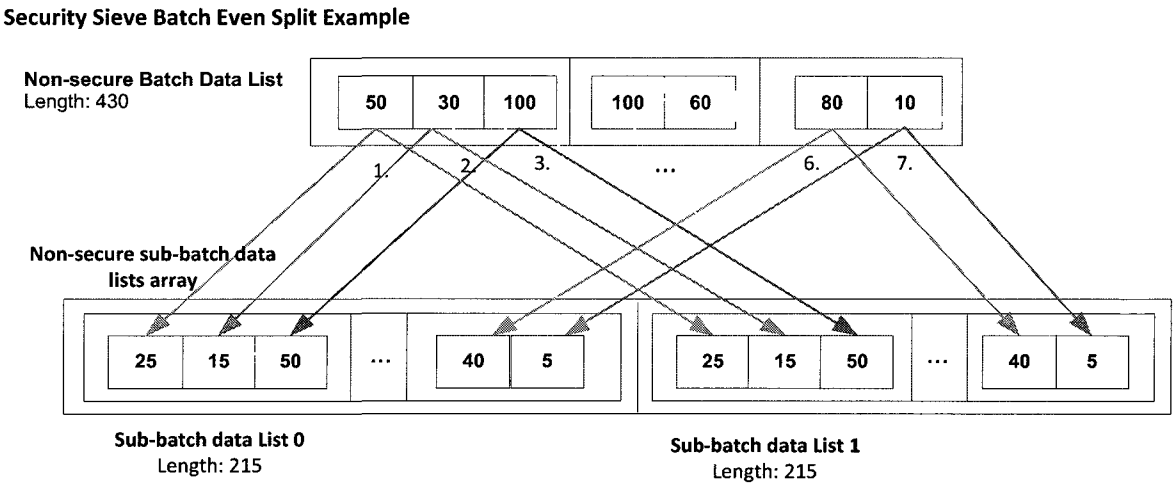


Figure 4.14: Illustration of the security sieve client's BES algorithm when two channels are used.

The BES algorithm used by secure-only client is implemented in a similar manner as the security sieve client's BES algorithm, and is described next.

1). Remove a `String` object from the `batchDataList` variable and store it in a variable called *s*. (Note each `String` object in the batch data list corresponds to data for an entire document.)
2). Use the (secure-only) ES algorithm (discussed in Section 4.3.3) to partition the data stored in *s*.
3). If there are more `String` objects in the `batchDataList`, go back to Step 1; otherwise, exit.

For both the secure-only and security sieve systems, the utility methods: `batchEvenSplit()` and `batchEvenCombine()` implement the BES algorithm, and the corresponding combine algorithm, respectively.

4.5.3 Batch Segment Split (BSS) Algorithm

The BSS algorithm, which is used only by the security sieve client, divides each `ArrayList<String>` object in the sieved batch data list using the SS algorithm (discussed in Section 4.3.4). Recall, that a sieved batch data list is either the secure batch data list or the non-secure batch data list obtained after sieving the batch data. The steps of the BSS algorithm are similar to the steps of security sieve client's BES algorithm discussed in Section 4.5.2. The only difference is that in Step 2, the SS algorithm is invoked instead of the ES algorithm.

An example of the security sieve client dividing the secure batch data list among two channels is shown in Figure 4.15. The numbers shown in the boxes of the diagram represent the length (or number of characters) of each `String`. The sequence of arrows in the diagram identifies the order in which the BSS algorithm partitions the secure batch data list. The algorithm starts by dividing the first secure data list, which contains three text segments with lengths equal to: 50, 30, and 100 characters, respectively. This secure data list is divided using the SS algorithm, and the results are shown in arrows 1 to 4.

Recall that the SS algorithm adds text segments to the sub-data list with the least number of characters. The first two text segments (see arrows 1 and 2) are added to the first and second sub-data lists, respectively. The last text segment is added to the second sub-data list (see arrow 3). Arrow 4 shows the `order` variable being added to the first sub-data list. The details on how the SS algorithm partitions a sieved data list were discussed in Section 4.3.4. The remaining secure data lists are partitioned using the SS algorithm as well, which is shown in arrows 5 through 10.

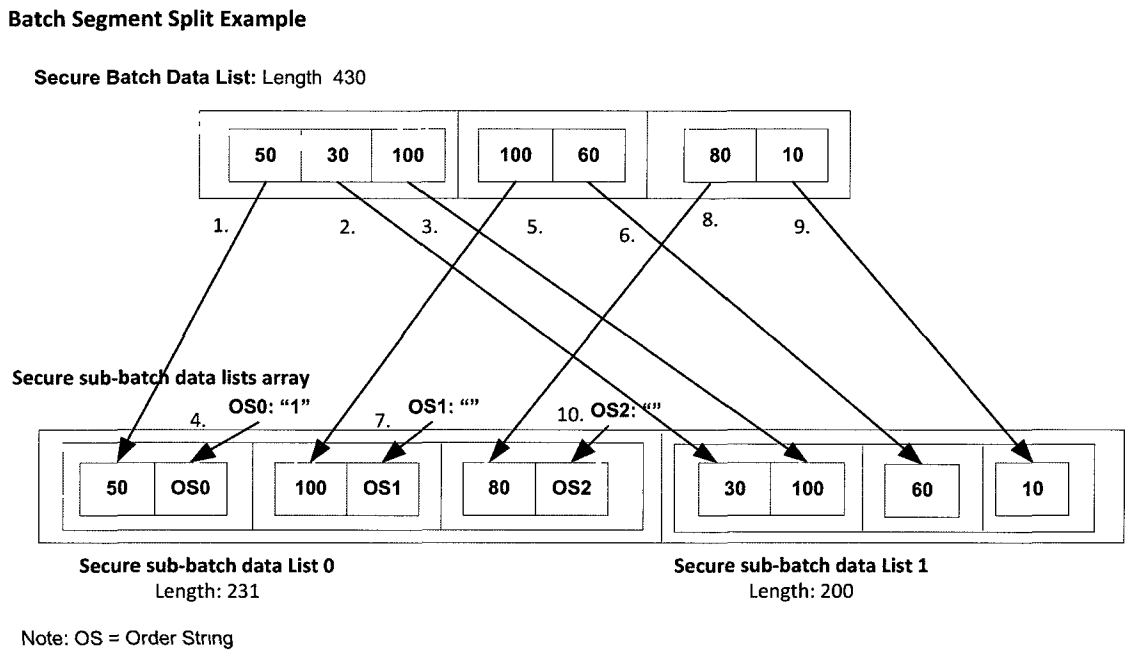


Figure 4.15: Illustration of the BSS algorithm when two channels are used.

The BSS algorithm and corresponding combine algorithm are implemented as methods named `batchSegmentSplit()` and `batchSegmentCombine()` in the security sieve client's and server's utility classes, respectively (see Figure 4.5 and Figure 4.6).

Chapter 5: Performance Evaluation

This chapter presents and discusses the results of the performance evaluation conducted on the prototype security sieve and secure-only systems. The evaluation of the two performance optimization techniques, PO1: using multiple channels, and PO2: batching of multiple files is also discussed.

5.1 Experimental Setup

The performance evaluation of the security sieve and secure-only systems was accomplished by conducting various file transfer experiments between two machines: one machine running as the client, and the other machine running as the server. Each experiment, unless specified otherwise, consists of performing 500 file transfers, and was conducted on both the secure-only and security sieve systems. All the file transfers come from the same client machine and the files transferred are identical to one another. For each file transfer the client establishes a new connection with the server. The client machine does the following for each file transfer: (1) connects to the server, (2) transfers the file, (3) records the performance metrics (discussed in Section 5.1.1), (4) closes the connection with the server, and terminates. After all 500 file transfers, the results of the individual file transfers are averaged and the confidence intervals at 95% confidence level are calculated. The confidence intervals, which are all less than $\pm$ 1%, are shown in the graphs. Note that because such small intervals are achieved, the marking for the intervals are often superimposed by the icons used in the graphs.

The next sections, Section 5.1.1 to Section 5.1.4, describe the following: the performance metrics, the files transferred in the experiments, the specifications of the systems, and the parameters used in the experiments.

5.1.1 Performance Metrics

Two metrics are used to compare the performance of the security sieve and secure-only systems: (1) mean *response time* and (2) mean *total time*. To measure the response time, the data transfer time is measured first. The client takes two timestamps (using the `nanoTime()` method provided by Java's System class [40]): once before the data is sent, and another timestamp when the client receives an acknowledgment from the server that all the data has been transferred. The data transfer time is the difference between these two timestamps. For the single channel-based secure-only system, the data transfer time is equal to the response time. In contrast, for the single channel-based security sieve system, in addition to the data transfer time, the response time includes the time it takes to sieve the file, and integrate the sieved data lists. The time to sieve the file is measured on the client machine by taking a timestamp before and after invoking the `sieve()` method, and then taking the difference between the timestamps. Similarly, the time to integrate the sieved data lists is measured on the server machine by taking a timestamp before and after invoking the `integrate()` method, and then taking the difference between the timestamps.

Additional or different measurements are made on the systems that deploy the performance optimization techniques, and are discussed next. In the experiments that use batching, the `batchSieve()` and `batchIntegrate()` execution times are measured instead of the `sieve()` and `integrate()` execution times. Furthermore, in the experiments that use multiple channels, the split and combine execution times are also included in the response times. The split and combine execution times are measured by

measuring the execution time of the split and combine methods (e.g. `evenSplit()` and `evenCombine()`).

The total time is the sum of the *connection establishment time* and response time. For the secure-only system, the connection establishment time includes the secure channel setup time and the TLS handshake time. For the security sieve system, the connection establishment time includes the secure channel connection establishment time, as well as the non-secure channel connection establishment time.

The mean response time and mean total time are computed by taking the mean of the response time and total time of all the file transfers performed during an experiment, respectively.

5.1.2 Description of the Files used in the Experiments

The files transferred in the experiments are categorized using two parameters:

1) File size, x (in megabytes (MB)),
2) Proportion of classified information, P

The naming convention for each file is x-P. The second parameter, P, is only important for the security sieve experiments since the secure-only system transfers all data over a secure channel(s). The files used in the experiments are Windows ".txt" files—each character in the file is one byte in size. The text files are synthetic and contain the appropriate number of characters, proportion of classified information, and tags, but the text does not have any semantic value. The text files were generated as follows. First, a 1MB-P file that contains five equal-sized secure data components was created. The length of each secure data component, l, is computed from x and P using the following formula: $l = \frac{x \times P}{5}$. The other files that have the same P value are created from the 1MB-P

file by replicating it an x number of times. For example, a 10MB-P file is created by copying the data generated for a 1MB-P file ten times.

5.1.3 System Specifications

The client and server machines both have Intel Core 2 Duo CPUs, 2GB of RAM, and run the Windows 7 operating system. The client's and server's CPU clock rates are 2.0 GHz, and 3.2 GHz, respectively. It is expected that similar relative performance results can be achieved when machines with different specs are used. Unless specified otherwise, the client and server are connected to a private local area network using a 100 Mbps Ethernet connection.

The secure channels use the TLSv1.0 (SSLv3.1) protocol [5]. Unless specified otherwise, the cipher suite used for the secure channels is: `RSA_WITH_3DES_EDE_CBC_SHA`. The RSA [9] algorithm with a 1024-bit key size is used for asymmetric cryptography and secret key exchange. The Triple DES (3DES) [6] algorithm is used for symmetric cryptography. Triple DES is an extension of the Data Encryption Standard (DES) that applies the DES algorithm three times to each data block using different keys each time. DES uses 64-bit keys but the actual key length used by the algorithm is only 56-bits long because the least significant bit in each byte is used as a parity bit, and is set so that each byte has an odd number of ones. Thus, the 3DES algorithm has an effective key length of 168-bits. The Secure Hash Algorithm (SHA) [10] is used for signing the data to ensure message integrity.

5.1.4 Summary of Experiment Parameters

A summary of the parameters used in the experiments is shown in Table 5.1. The parameters are divided into three categories: workload, system, and other. Workload

parameters specify the type of files transferred in the experiments, whereas system parameters define how the client and server machines were configured. When multiple communication channels are used, the type of split/combine algorithm used for partitioning the data among the multiple channels also needs to be specified.

Table 5.1: Summary of the parameters used in the experiments.

Type of Parameter	Name	Values	Default Value
Workload	**File size, *x* (in MB)** (see Section 5.2.2)	1, 3, 5, 7, 10	1
	Proportion of classified information in a file, *P* (see Section 5.2.1)	0.1, 0.25, 0.5, 0.75, 0.9, 1	0.5
	Batch File Set* (see Section 5.4.1-5.4.3)	Uneven, Even	N/A
	Arrival rate, λ (requests/s)* (see Section 5.4.4)	0.1, 0.5, 2, 10,	N/A
System	**Connection type** (see Section 5.2.4)	Wired- Ethernet, Wireless- Wi-Fi	Wired - Ethernet
	Symmetric Cipher (see Section 5.2.3)	3DES, AES-256	3DES
	Power Options (see Section 5.2.4)	High Performance, Power Saving	High Performance
	Number of Communication Channels, *N=M* (see Section 5.3)	1, 2, 3, 5	1
Other	**Split/Combine Algorithm*** (see Section 5.3)	Even, Segment	N/A
	Batch Split/Combine Algorithm* (see Section 5.4.1 and Section 5.4.2)	Even, Segment, File	N/A
	Additional Processing Time (in ms)* (see Section 5.2.4.1)	500	N/A

* This parameter is only used in the experiments discussed in the specified section.

The name of the parameter and the section where the effect of the parameter on system performance is discussed are shown in the second column. In addition, the values and default values that the parameters are given are displayed in the third and fourth columns, respectively. A factor at a time experiment is performed: one parameter is varied in an experiment and the other parameters are held at their default values (see Table 5.1). Note that there are some parameters, identified by asterisks, which are used only in the experiments discussed in the section specified in the second column.

The choice of default values for the parameters is discussed next. The file size is varied between 1MB and 10MB. A 1MB file size was chosen because it was a file size that was used in similar research in the literature (see [12], [13], and [14] for examples). To see the effect of transferring more data, file sizes up to 10MB were used in the experiments. The default value of P was chosen to be 0.5 since it is in the middle of the range of P values experimented with. For P=0.5, half the data is classified and half the data is non-classified. 3DES and AES are chosen to be the symmetric cipher suites used in the experiments because they are some of the more commonly used symmetric encryption algorithms used in TLS/SSL. They have also been used in previous research discussed in the literature (see [1], [14], and [15]). The default value for N is set to one because in the state of the art: data is typically transferred using a single secure channel. The default values for connection type and power options were chosen because those are the values used in most desktop machines.

5.2 Evaluation of the Security Sieve Technique

This section evaluates the security sieve technique by comparing the performance of the single channel-based security sieve system with the single channel-based secure-

only system. Recall that the single channel-based security sieve system uses one secure channel, and one non-secure channel (i.e., N=M). The response times and total times of transferring files with varying x and P values were measured. To evaluate the effectiveness of the security sieve technique, the improvement in mean response time and improvement in mean total time achieved using the security sieve system compared to the secure-only system was calculated using the following formula:

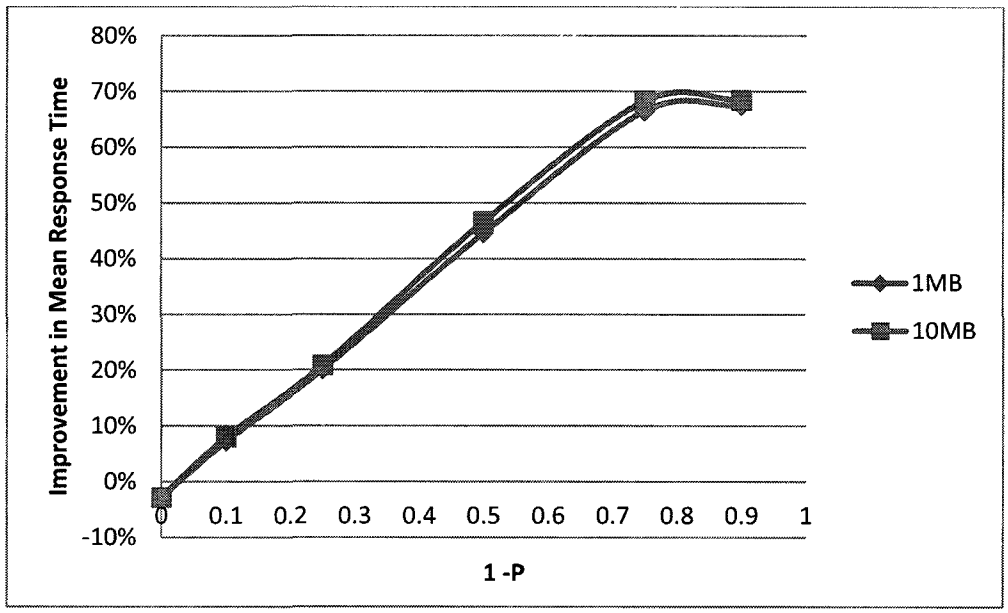
$$\%\ \text{improvement in mean time} = \frac{mean\ secure\ only\ \ time - mean\ \ security\ sieve\ time}{mean\ \ secure\ only\ time} \times 100 \quad (1)$$

where "time" can be response time or total time.

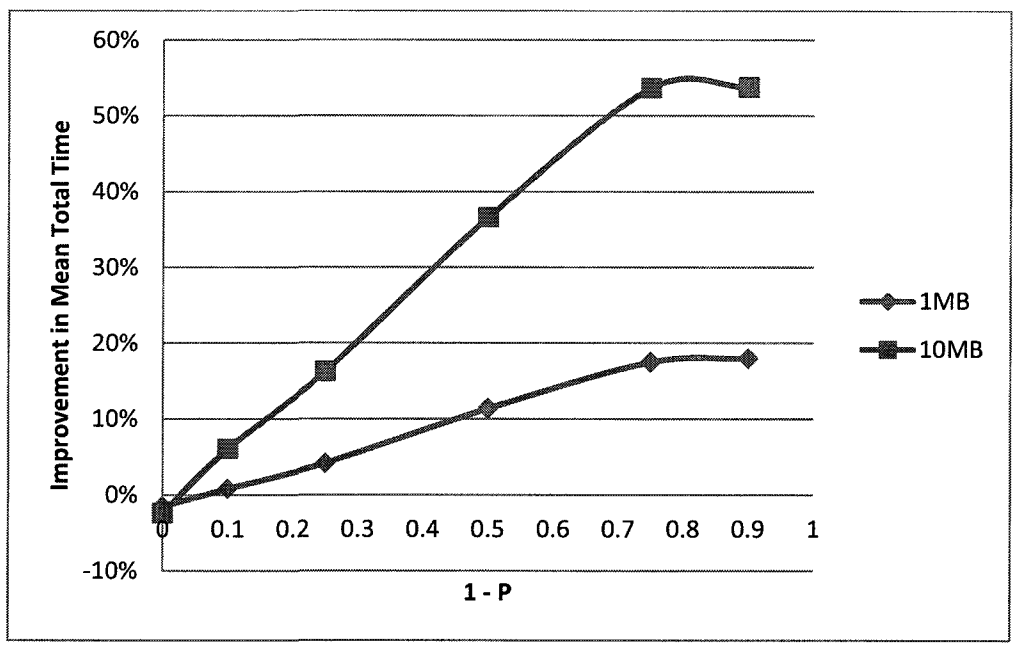
5.2.1 Effect of Varying the Proportion of Classified Information

The graphs in Figure 5.1 show the improvement in (a) mean response time and (b) mean total time achieved by the security sieve system when P is varied, and x equals 1MB and 10MB. As expected, for a given x, the improvement in mean total time increases as P decreases. The smaller the value of P (higher the value of 1-P), the higher the performance improvement achieved with the security sieve system. Specifically, when P is less than approximately 95% (or 1-P > 5%), the security sieve system starts outperforming the secure-only system. The reason for the performance improvement is because in the security sieve system, fewer security-related operations need to be applied to the data, which in turn reduces the response times. This improvement in response time is high enough to overcome the additional security sieve system overheads, which include: (1) the non-secure channel connection time, and (2) the sieve and integration times. The mean sieve time and mean integration time for the 1MB files were 1.72 milliseconds (ms) and 0.01 ms, respectively; and for the 10MB files, they were 16.9 ms

and 0.03 ms, respectively. The mean non-secure channel connection establishment time is approximately 25 ms.



(a)



(b)

Figure 5.1: The effect of P on (a) the improvement in mean response time and (b) the improvement in mean total time achieved by the security sieve system.

As expected, the largest improvement in performance is gained when transferring a file with a small *P*. For the 1MB-10 and 10MB-10 files, the mean response times for the security sieve system are 95 ms and 910 ms, respectively, and for the secure-only system they are 293 ms and 2884 ms, respectively. The values for improvement in mean response time are calculated using Eq. 1, and the values are 67% and 68% for the 1MB-10 and 10MB-10 files, respectively. For the 1MB-90 and 10MB-90 files, for which most of the data is classified, the improvement in mean response time is smaller: 7% and 8% respectively. The only case where the secure-only system outperforms the security sieve system is when *P* is one, that is, the entire document contains classified information. This is expected because of the additional overheads associated with using the security sieve system, and the absence of the performance saving non-secure data in the files.

As shown in Figure 5.1b, a significantly higher improvement in mean total time is achieved as *P* decreases. For example, the improvements in mean total time are 18% and 54% for the 1MB-10 and 10MB-10 files, respectively, compared to 0.8% and 6% for the 1MB-90 and 10MB-90 files. The improvement in mean total time is lower compared to the improvement in mean response time (as shown in Figure 5.1a) because of the following: (1) the mean secure channel connection time and handshake time (approximately 755 ms) makes up a large proportion of the total time for both the secure-only and security sieve systems, and (2) the additional non-secure channel connection time overhead incurred by the security sieve system. The reason for the larger decrease in improvement in mean total time for the 1MB-10 file compared to the 10MB-10 file is because the smaller the file size, the more the connection establishment time dominates the total time. With a large file, the connection time does not influence the total time as

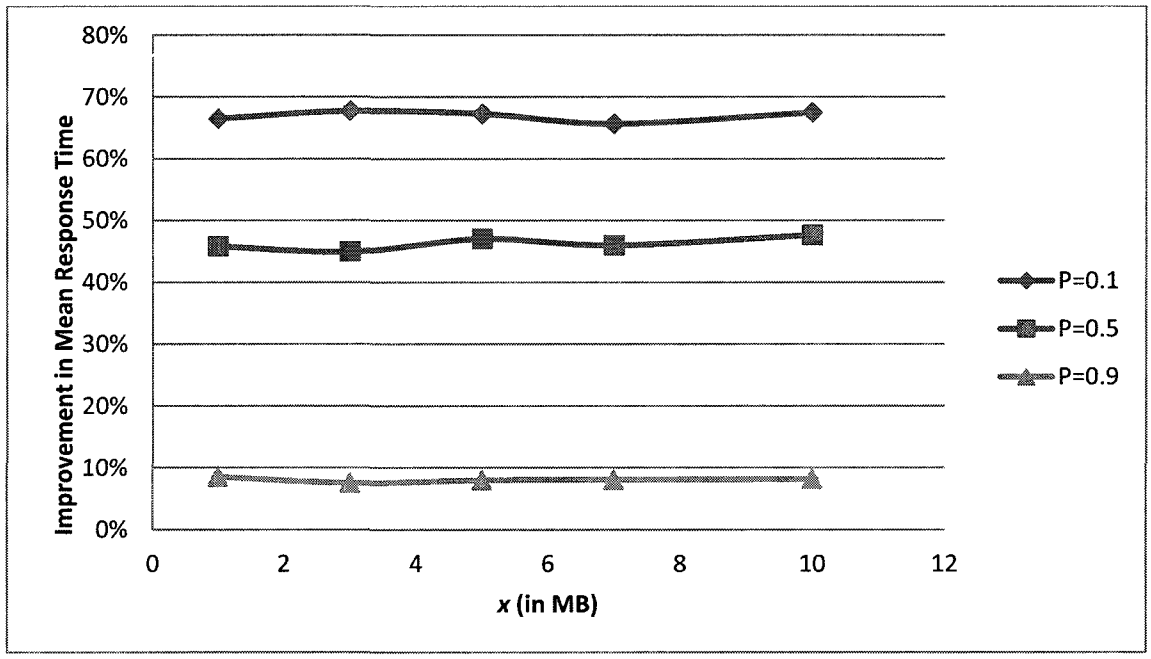
much because of the longer response times. The security sieve technique only reduces the response times and slightly increases connection establishment time due to the addition of the non-secure channel. Therefore, the improvement in mean total time is always a little lower compared to improvement in mean response time.

5.2.2 Effect of Varying the File Size

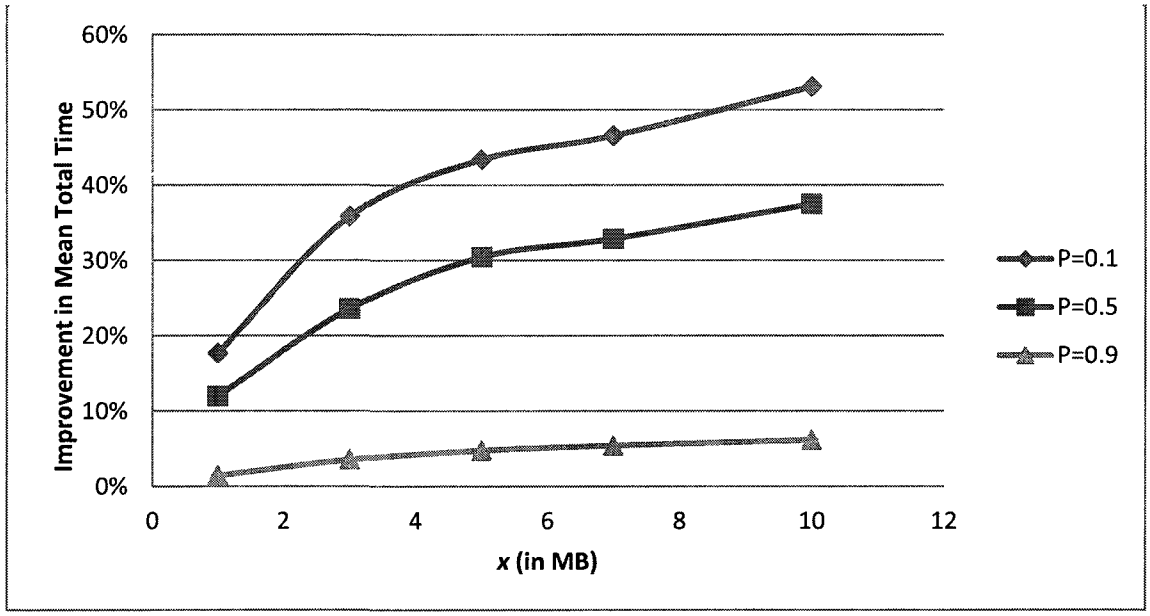
The improvement in (a) mean response time and (b) mean total time achieved by the security sieve system with various values of x are shown in Figure 5.2. P is held at a specific level of 0.1, 0.5, and 0.9, in each experiment. In all cases, the security sieve system outperforms the secure-only system. As shown in Figure 5.2a, for a given P, as x increases, improvement in mean response time remains relatively the same. The reason for this is because increasing the file size appropriately increases the response time for both the secure-only and security sieve systems, maintaining the same performance improvement. Conversely, as shown in Figure 5.2b, for a given P, as x increases, improvement in mean total time increases because of the reduced dominance of the connection establishment time on the total time (as discussed in Section 5.2.1 for Figure 5.1b).

5.2.3 Effect of Changing the Symmetric Cipher Suite

In these experiments the SSL/TLS cipher suite used by the secure channel is changed to `RSA_WITH_AES_256_CBC_SHA`. Specifically, the symmetric key encryption algorithm used for bulk data transfer is changed to AES [7] with a key length equal to 256-bits. AES is the successor to DES/3DES, and is more secure and more efficient than 3DES [42] [43]. In 3DES, data has to be encrypted three times which makes it a more CPU-intensive algorithm compared to AES.



(a)



(b)

Figure 5.2: The effect of x on (a) the improvement in mean response time and (b) the improvement in mean total time achieved by the security sieve system.

The improvement in mean response time achieved when using 3DES and AES-256 is shown in Figure 5.3. Note that for the 3DES case, the values for the improvement in mean response time are the same as those shown in Figure 5.1, and are shown here to compare with AES-256 values. As expected, as P decreases, the improvement in mean response time increases regardless of whether AES-256 or 3DES is used. However, the improvement in mean response time achieved when using AES-256 is lower compared to when 3DES is used. For example, when P is 0.1, the improvement in mean response time peaks at approximately 30% when AES is used, compared to approximately 67% when 3DES is used.

The reason for the lower performance improvement is because AES is naturally a faster and more efficient encryption algorithm as discussed above. Therefore, when using AES the performance penalty of sending data over the secure channel is smaller. This reduces the response time for the secure-only system, and to a less extent the security sieve system. For example, the mean response time for transferring 1MB of secure data using AES-256 was approximately 139 ms, compared to 293 ms when 3DES is used. For the security sieve system, the mean response time for transferring a 1MB-50 file using AES-256 was approximately 102 ms, compared to 162 ms when 3DES is used. The more significant decrease in response times achieved by the secure-only system reduces the performance improvement achieved by security sieve system.

Figure 5.3 also shows that when using AES-256, there is little change in performance improvement when P is between 0.1 and 0.5. This shows that for relatively low values of P, the data transmission time seems to influence the response time more

than the security-related processing time (i.e. the processing time required to encrypt/decrypt and hash the data).

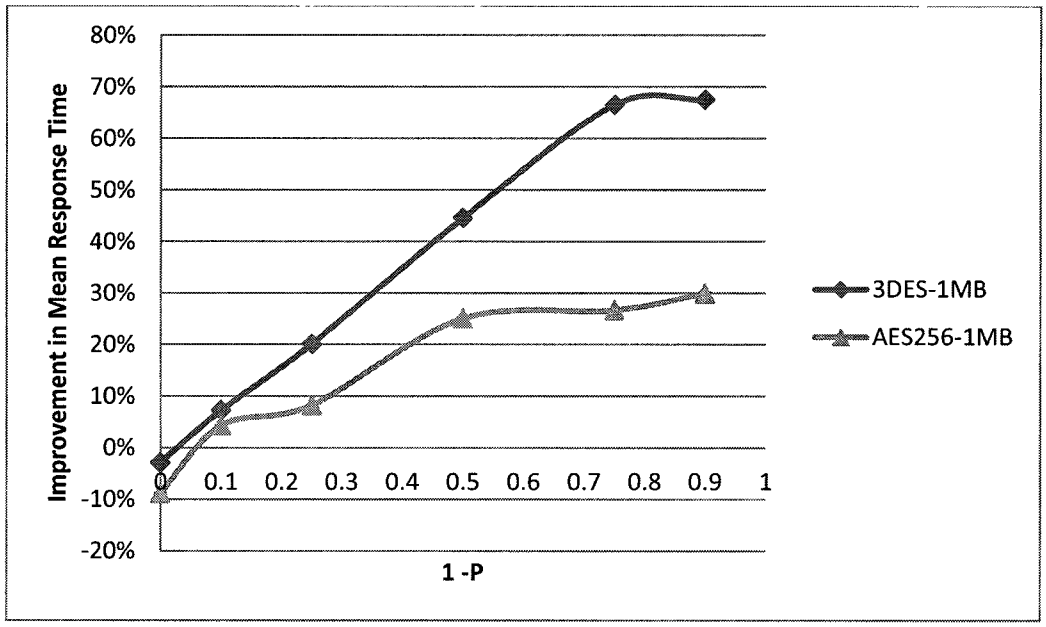


Figure 5.3: 3DES vs. AES-256: improvement in mean response time achieved by the security sieve system when *P* is varied.

The improvement in mean total time achieved when using 3DES and AES-256 with various values of *x* is shown in Figure 5.4. Note that the results shown for the 3DES-based system are the same as the results shown in Figure 5.2. For the 3DES and AES-256 based systems, the improvement in mean total time increases with file size; however the increase in performance improvement for the AES-based system is lower compared to the 3DES-based system. As discussed earlier, when using AES, the performance penalty for sending data over the secure channel is smaller. This is more favourable to the secure-only system since it sends more secure data, and thus the performance improvement achieved by security sieve system decreases.

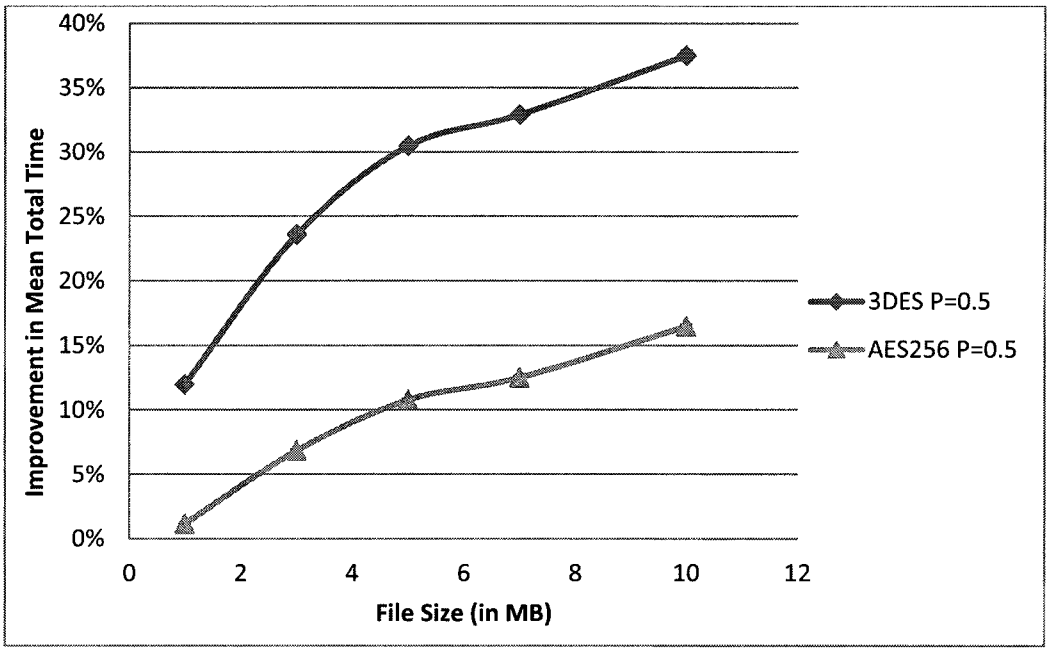


Figure 5.4: 3DES vs. AES-256: improvement in mean total time achieved by the security sieve system when x is varied.

5.2.4 Effect of Moving the Client to a Wireless Environment

These experiments investigate how performance improvement is affected when the client communicates with the server wirelessly using a Wi-Fi connection, instead of an Ethernet (wired) connection. In the wireless environment, the maximum data transmission speed is approximately 54 Mbps. To observe the effect of having a client with reduced processing speed sending data wirelessly (mimicking the lower processing capabilities of handheld devices), the Windows power option on the client machine was set to "power saving". The power saving option reduces the processing speed of the CPU by approximately 20% in effort to save power. This observation was made on the operating system's resource monitor. The default power option that the machines were set to is "high performance". When this option is used the operating system focuses on performance and makes no effort to save power.

The improvement in mean response time achieved in a wireless environment compared to a wired environment is shown in Figure 5.5. Note that for the wired case, the values for the improvement in mean response time are the same as those shown in Figure 5.1, and are shown here to compare with the wireless cases. As expected, when P decreases, the improvement in mean response time increases in all cases; however, the performance improvement in the wireless environment is much lower compared to the wired environment. For example, when P is 0.1, the improvement in mean response time achieved in the wireless environment is approximately 8%, but in the wired environment the performance improvement is approximately 67%.

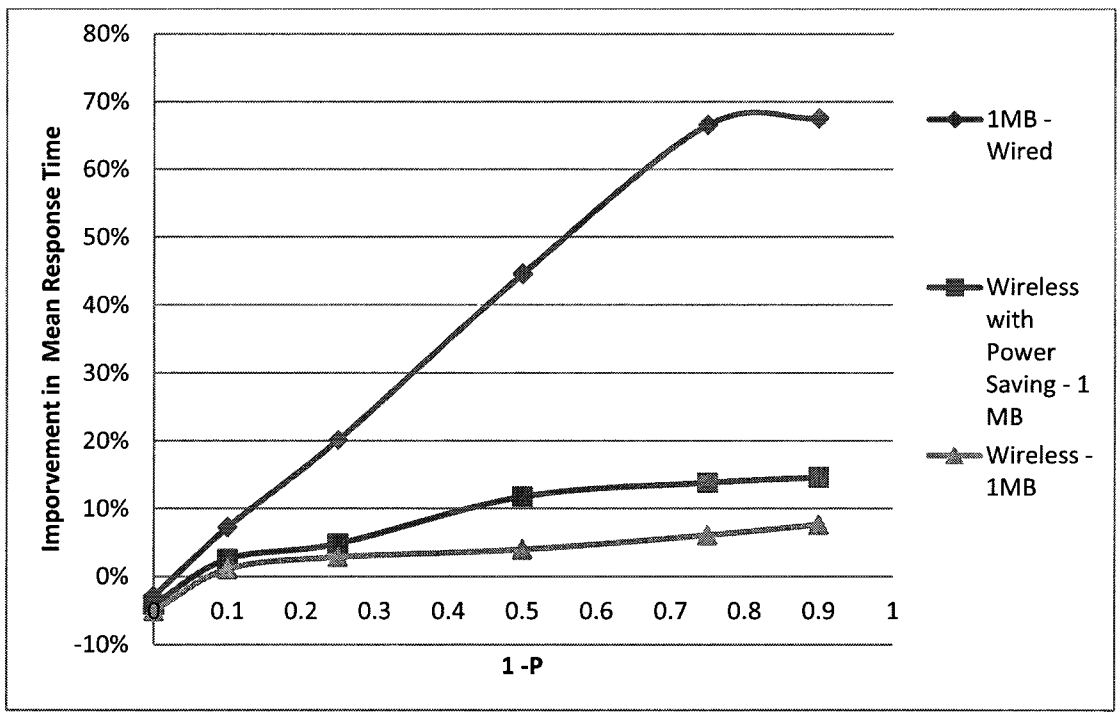


Figure 5.5: Wired vs. wireless environment: improvement in mean response time achieved by the security sieve system when P is varied.

The lower performance improvement achieved by the security system in a wireless environment can be attributed to the long data transmission times dominating the

response times. In other words, the response time now depends more on the data transmission time and to a less extent on the time required to perform the security-related operations (i.e. encryption/decryption and hashing). As a result, the security sieve technique, which reduces the security-related operations that need to be performed, is not as effective.

Although, when the processing capability of the client is reduced (i.e. the power saving option is used), the security sieve technique becomes more effective and performance improvement increases. For example, when P is 0.1 the improvement in mean response time increases from 8% to approximately 15%. The lower processing capability of the client increases the processing time required to perform the security-related operations, which lowers the dominance of data transmission time on the response time. The mean response time measured for transferring 1MB of secure data with the power saving option was 557 ms, compared to 501 ms when the power saving option was not used. This corresponds to an increase of approximately 11%. As more data is transferred, the percentage increase in mean response time will also increase since more security-related operations need to be performed.

The improvement in mean total time achieved by the security sieve system in the wired and wireless environments is presented in Figure 5.6. Note that for the wired environment, the values for the improvement in mean total time are the same as the values presented in Figure 5.2, and are shown here to compare with the values achieved in the wireless environment. As expected, for all cases, the improvement in mean total time increases with file size due to the reduced influence of the connection establishment time on the total time (as discussed in Section 5.2.2). However, notice that the increase

in performance improvement achieved by the wireless system not using the power saving option is lower compared to the wireless system using the power saving option. This is because in the power saving option case, as file size increases, the performance penalty for sending data securely becomes more significant due to the lower processing capability of the client. The security sieve technique, which minimizes the security-related operations that need to be performed, is most effective when the processing time required to perform the security-related operations is high.

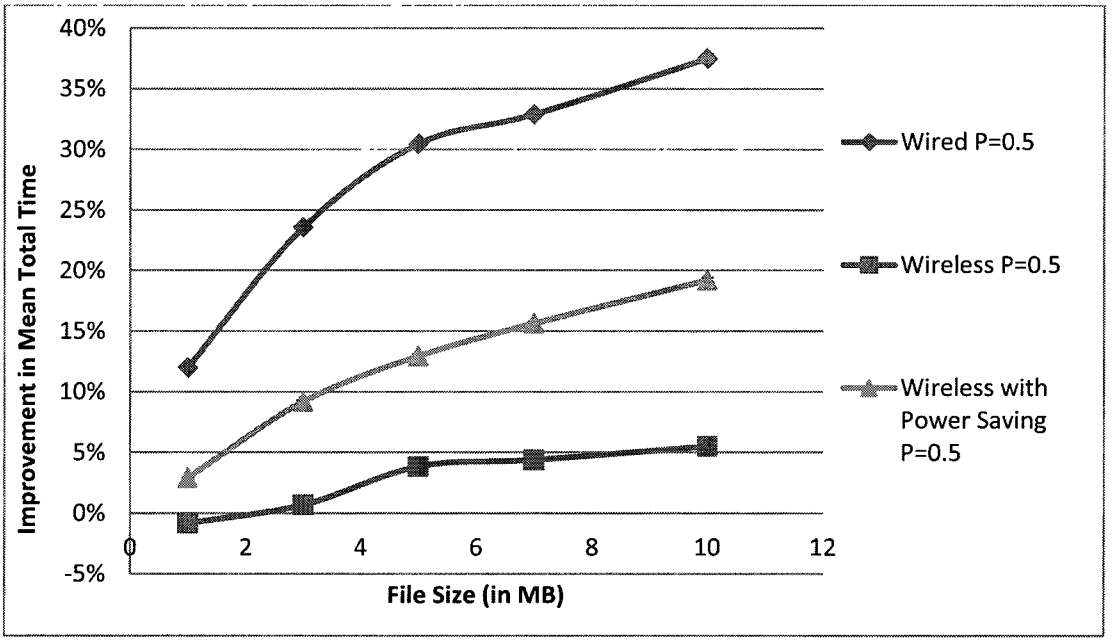


Figure 5.6: Wired vs. wireless environment: improvement in mean total time achieved by the security sieve system when x is varied.

5.2.4.1 Additional Processing Time

To further investigate how a client characterized by lower processing capability affects the performance improvement of the security sieve system in a wireless environment, experiments were conducted where the processing time required for a client to transfer secure data is artificially increased. A new parameter called Additional

Processing (AP) is used in these experiments. AP is defined as the additional processing time required for transferring 1MB of data using the secure channel. This additional processing time was simulated by making the client sleep for t milliseconds after a file transfer is completed. To make the client sleep, Java's `Thread.sleep()` [44] method was invoked. The amount of time a client sleeps, t, is calculated as follows. For the secure-only system, $t = x \times AP$, and for the security sieve system, $t = x \times P \times AP$. For example, in the case of the security sieve system, if AP is 500 ms, x is 5MB, and P is 0.5, t will be 250 ms.

The improvement in mean total time achieved by the security sieve system in the wireless environment, when AP is 500 ms compared to when no additional processing time is used, is shown in Figure 5.7. An AP of 500 ms was used because it emulates a processor with a clock rate of approximately 1.0 GHz. Many of the top end mobile devices today have clock rates of at least 1.0 GHz (see [45] and [46] for examples). Recall that the clock rate of the client machine was 2.0 GHz. Since the response time of sending 1MB of data wirelessly using the secure channel is approximately 500 ms, setting AP to 500 ms is slowing down the processing time by a factor of two. Conducting experiments using other AP values forms an important direct for future research.

The results in Figure 5.7 confirm that increasing the processing time required to perform the security-related operations does reduce the dominance of the data transmission time on the response time, leading to an increase in performance improvement achieved with the security sieve system. For example, when x is 1MB and P is 0.1, the improvement in mean response time achieved by the security sieve system

with AP=0 is approximately 8%; however, when AP is 500 ms, the improvement in mean response time increases to approximately 49%.

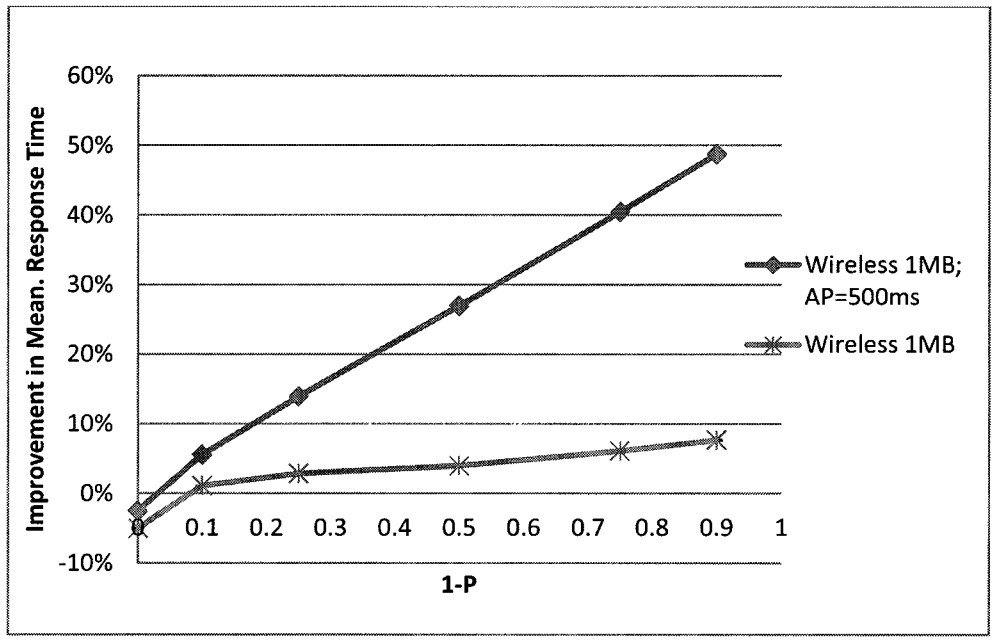


Figure 5.7: Wireless (AP=0) vs. Wireless (AP=500ms): improvement in mean response time achieved by the security sieve system when *P* is varied.

5.3 Evaluation of Performance Optimization 1 - Multiple Channels

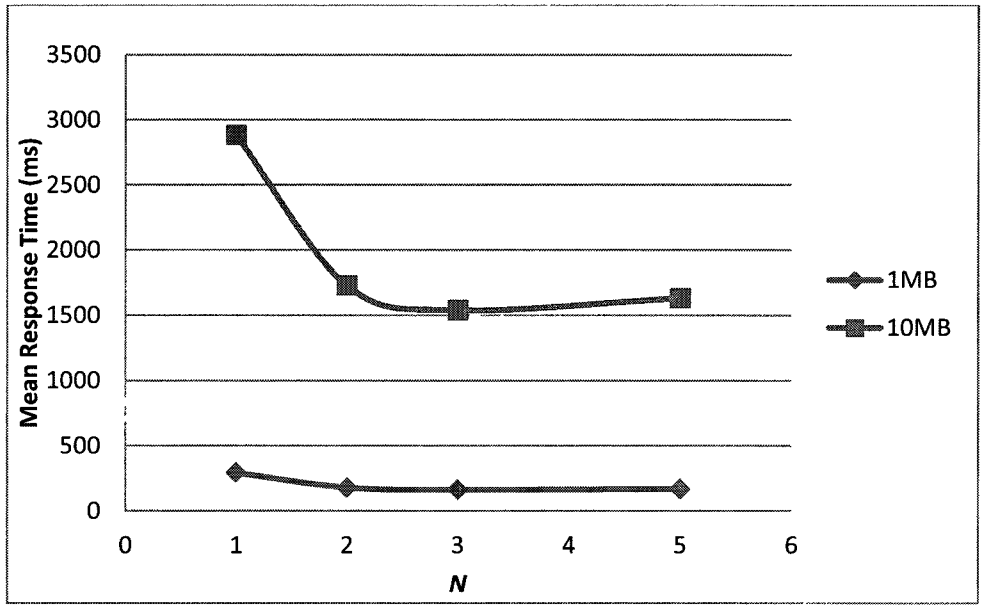
In this section, PO1, using multiple channels to achieve concurrency of operations, is evaluated. First, the effectiveness of adding multiple channels is discussed for the secure-only and security sieve systems separately. The results achieved by the secure-only and security sieve systems are then compared.

5.3.1 Effect of using Multiple Channels in the Secure-only System

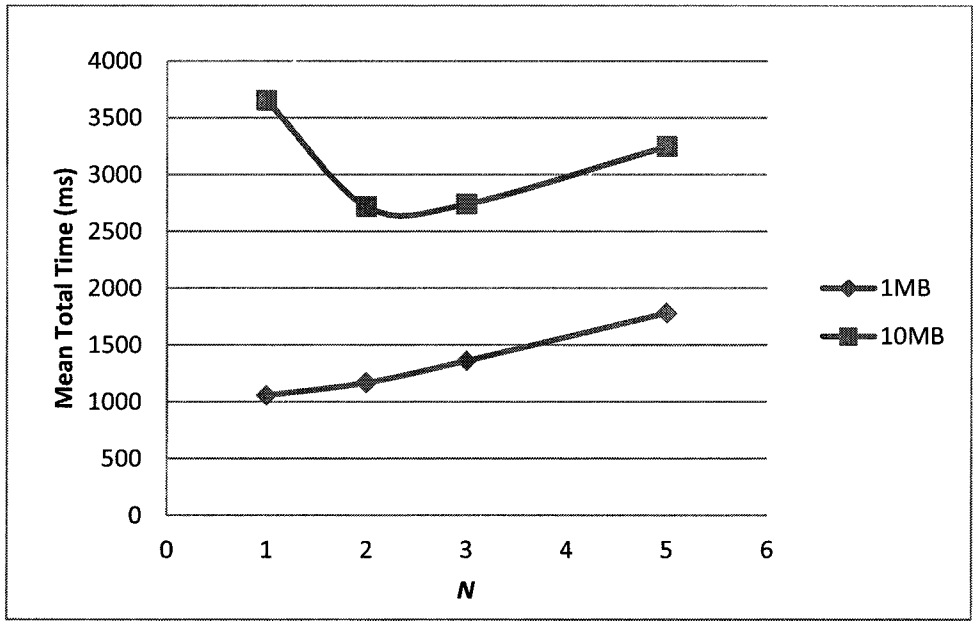
Figure 5.8 shows (a) the mean response time and (b) the mean total time achieved by the secure-only system when *N* is varied. For the 1MB file, using multiple channels marginally improves response time when compared to using a single channel. There is a

small improvement at first, and then a knee is reached. For the 10MB file, the improvement in mean response time is more substantial. The lowest mean response time is achieved when using three channels. The response time slightly increases when five channels are used. A possible reason for this can be the additional thread context switching overhead incurred when using multiple threads. Using multiple channels helps improve response time because the security-related operations (such as encryption/decryption) can be performed by one thread while another thread is waiting to send or receive data. For example, the send threads have to wait for the receive threads to finish reading data from the buffer before more data is transmitted.

As shown in Figure 5.8b, for the 1MB file, using multiple channels does not improve total time when compared to using a single channel. The marginal improvement in response time is not enough to overcome the additional connection establishment time. The mean connection establishment time for two channels is 988 ms, whereas for one channel it is only 755 ms. Each additional secure channel, adds approximately 230 ms to the overall connection establishment time. The connection establishment time for each additional secure channel is smaller than the connection establishment time for the first channel because the SSL/TLS protocol caches communication sessions. For the 10MB file, using multiple channels does improve total time. The lowest total time is achieved when two channels are used. When three and five channels are used, the total time is higher than the single channel case because the additional connection establishment time offsets the improvement in response time gained from using multiple channels.



(a)



(b)

Figure 5.8: The effect of using multiple channels on (a) the mean response time and (b) the mean total time achieved by the secure-only system.

5.3.1.1 Secure-only System: Effect of File Size when using Multiple Channels

Figure 5.9 presents the mean total time achieved by the secure-only system (as shown in Figure 5.8b), but this time with file size on the x-axis. This graph shows that there is a minimum amount of data that needs to be transferred for the multiple channel-based secure-only system to achieve a lower total time than the single channel-based secure-only system. There is a presence of a *file size threshold.* This threshold is the *x* value for the point at which the line for a single channel intersects the line for the multiple channels. The approximate file size thresholds for two, three, and five channels are approximately: 2MB, 3MB, and 7MB, respectively. As the number of channels used increases, the file size threshold increases because more data needs to be transferred so that the improvement in response time achieved when using multiple channels can overcome the connection establishment/tear down overhead.

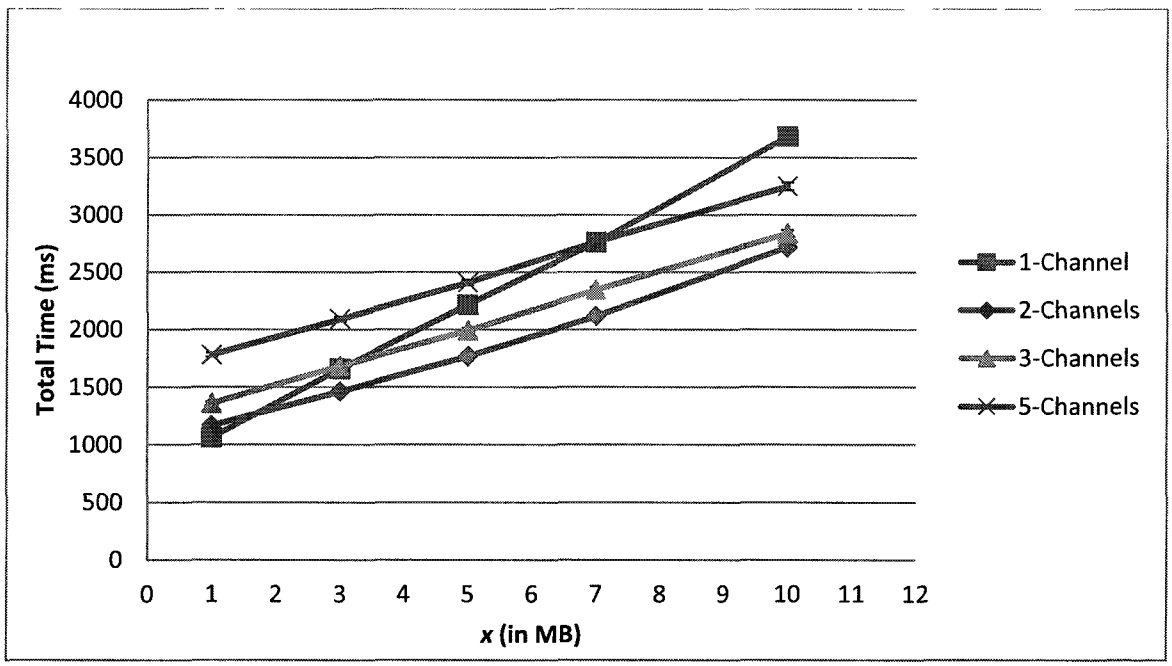
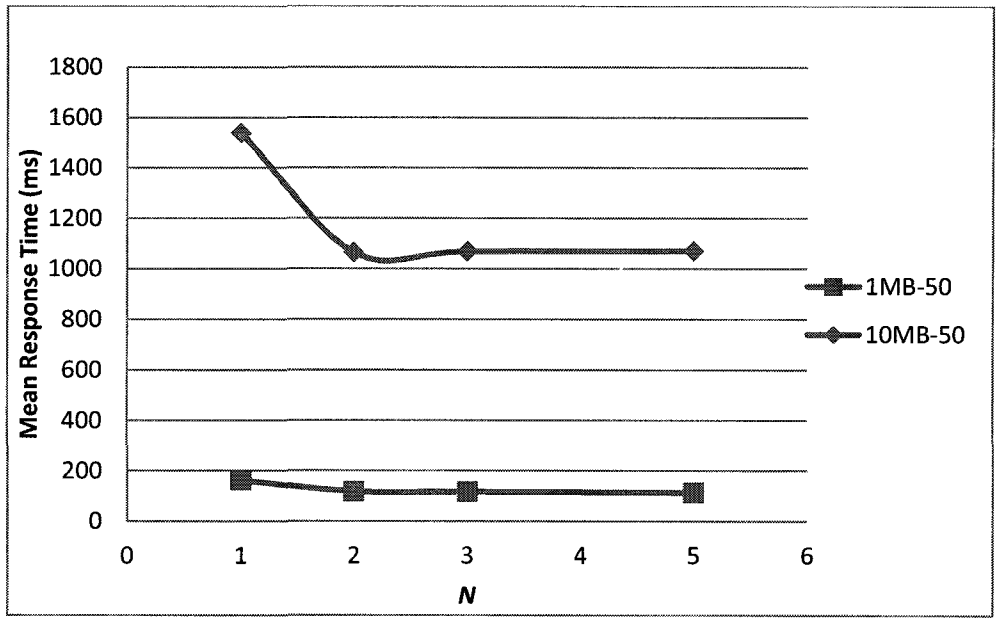


Figure 5.9: Mean total time achieved by the secure-only system when file size is varied, and multiple channels are used.

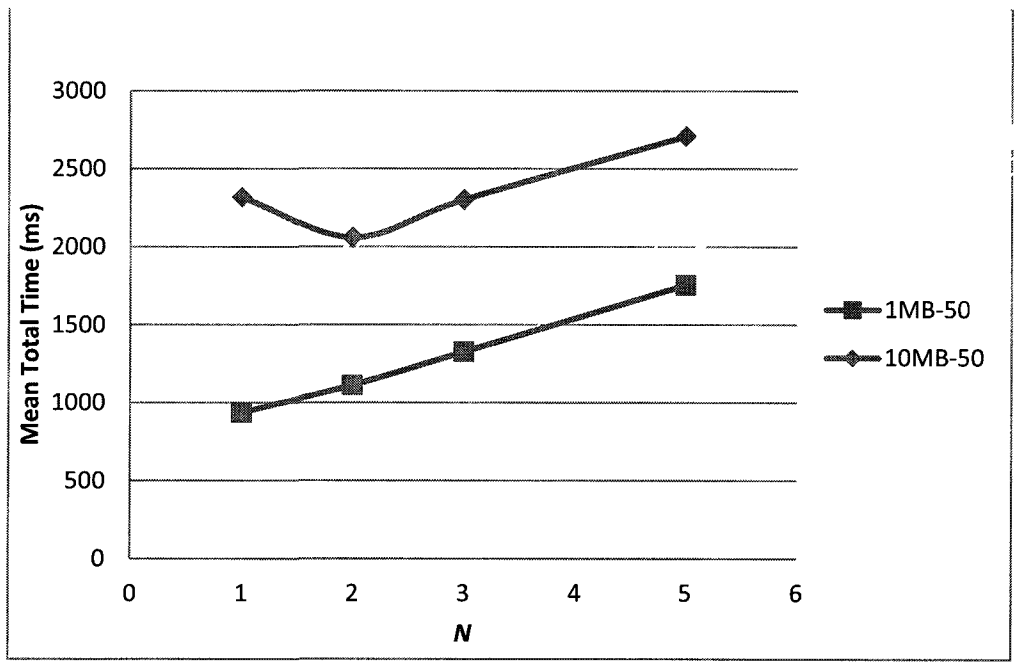
5.3.2 Effect of using Multiple Channels in the Security Sieve System

Figure 5.10 shows (a) the mean response time and (b) the mean total time achieved by the security sieve system when multiple channels are used. Note that the experiments discussed in this section use the default split/combine algorithm, ES (discussed in Section 4.3.3) to divide the data among the multiple channels. In addition, the number of secure channels used in these experiments is the same as the number of non-secure channels used (i.e. $N=M$). The effect of using multiple channels on the performance of the security sieve system (displayed in Figure 5.10) is similar to the results of the secure-only system, shown in Figure 5.8. For the 1MB file, as N increases, the response time decreases marginally until the knee is reached. For the 10MB file, the lowest response time is achieved when two channels are used. The response time slightly increases when using more than two channels. As discussed earlier, this may be caused by the thread context switching overhead incurred when using multiple threads. Recall that in the security sieve system, when $N=M=3$, there are six threads created for both the client and server: one thread to service each of the three non-secure channels, and one thread to service each of the three secure channels.

The mean total time achieved by the security sieve system when multiple channels are used, is shown in Figure 5.10b. For the 1MB file, the mean total time increases, as the number of channels increases. This is because the slight improvement in response time is not enough to overcome the additional connection establishment time incurred when using multiple channels. Each additional secure and non-secure channel adds about 230 ms, and 2 ms to the connection establishment time, respectively. For the 10MB file, the lowest total time is achieved when using two channels, since the two channel-based



(a)



(b)

Figure 5.10: The effect of using multiple channels on (a) the mean response time and (b) the mean total time achieved by the security sieve system.

system also achieved the lowest response time (as shown in in Figure 5.10a). In addition, the connection establishment time for the system using two channels is the lowest of all the multiple channel-based systems.

5.3.2.1 Security Sieve System: Effect of File Size when using Multiple Channels

Figure 5.11 shows the mean total time achieved by the security sieve system when x is varied and multiple channels are used. As expected, for a given N, as x increases, mean total time increases. Figure 5.11 reveals the existence of a file size threshold. Recall (from Section 5.3.1.1) that the file size threshold is the minimum amount of data that is required to be transferred for a system using multiple channels to achieve a lower mean total time than a system using a single channel. The approximate file size thresholds, when transferring files with P=0.5, for two channels and three channels are approximately 4MB, and 8MB, respectively. The file size threshold for five channels cannot be seen in this graph.

These results suggest that using two channels is most effective when there is at least 2MB of secure data being transferred (refer to the discussion in Section 5.3.1.1). Recall that a 4MB-50 file contains 2MB of secure data and 2MB of non-secure data. A minimum amount of secure data needs to be transferred so that the improvement in response time gained by using multiple channels can offset the additional channel establishment/tear down overhead. Using multiple channels is effective in reducing response times because of concurrency. When one thread is waiting to send or receive data, another thread can be performing the security-related operations (such as encryption/decryption).

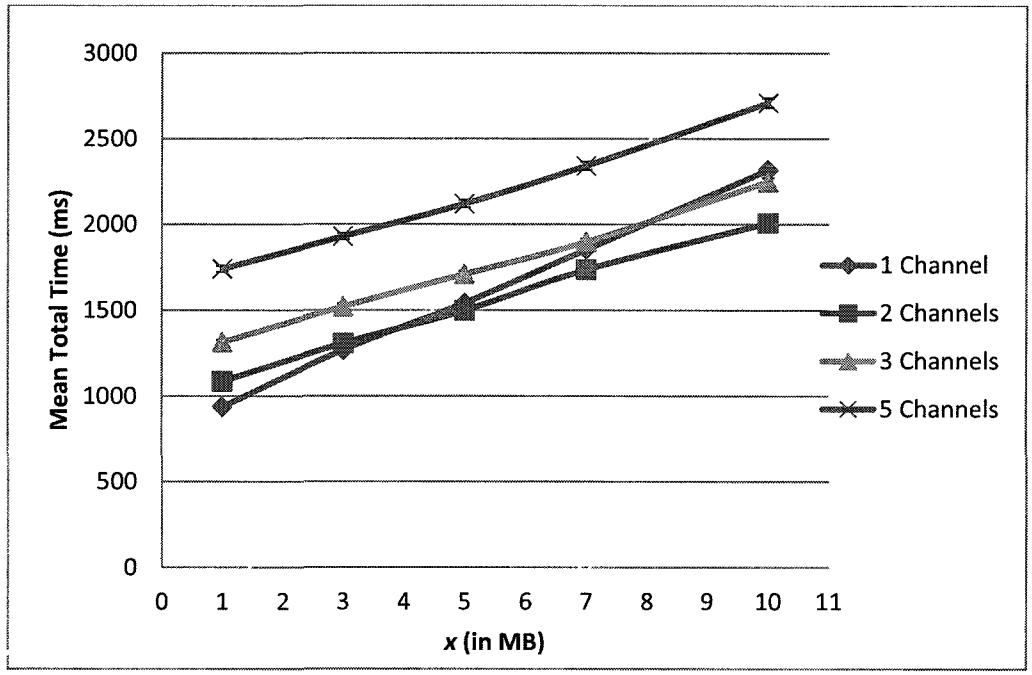


Figure 5.11: Mean total time achieved by the security sieve system when x is varied and multiple channels are used (P=0.5).

5.3.2.2 Comparison of the Split/Combine Algorithms

As discussed in Section 4.3, two types of split/combine algorithms, SS and ES, were derived for the security sieve system deploying PO1. A comparison of the mean total time achieved when using ES and SS to divide the data among two channels (i.e. *N=M=2)* is presented in Figure 5.12. The graphs show that the ES algorithm starts to outperform the SS algorithm when 1-P is less than 0.4 (i.e. P is greater than 0.6). When the majority of the data in the file is classified, it is more important for each channel to send an equal amount of data so that concurrency can be maximized. Recall using multiple channels helps improve response times because the security-related operations (such as encryption/decryption) can be performed by one thread while another thread is waiting for data to be sent or received. For all the other P values, the SS-based system has slightly lower response times because the processing time required to invoke the

`segmentSplit()` and `segmentCombine()` methods is lower compared to the time required to invoke the `evenSplit()` and `evenCombine()` methods. This is captured in Table 5.2, which presents a comparison of the processing times required to execute the split and combine methods for partitioning a 10MB-50 file.

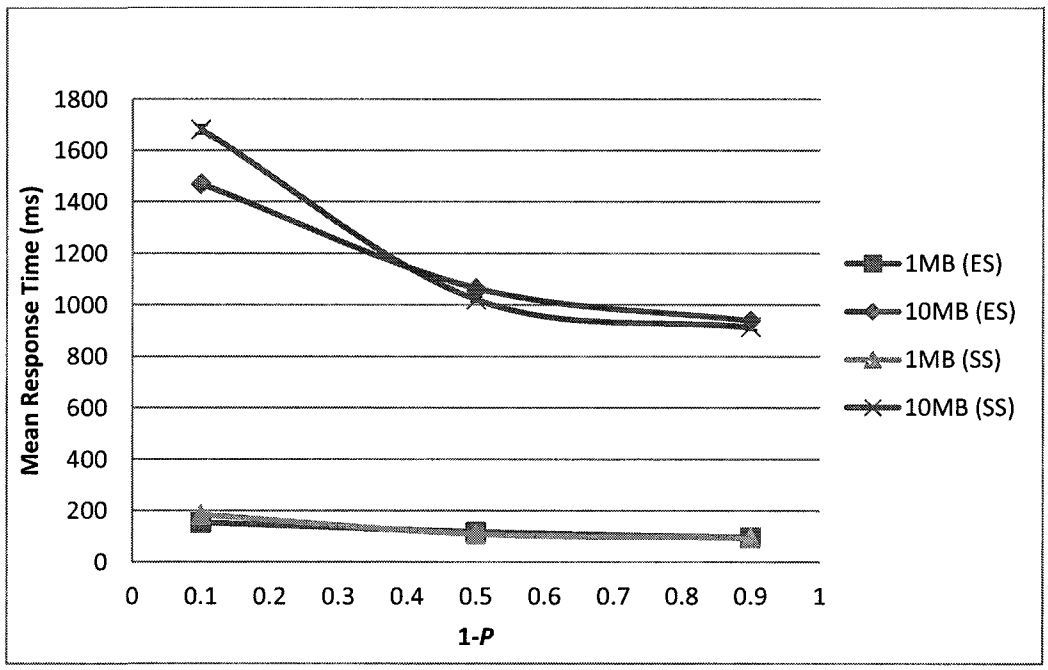


Figure 5.12: ES vs. SS: mean response time achieved by the security sieve system for various values of P (N=M=2).

Table 5.2: Processing time required for invoking the split and combine methods to divide a 10MB-50 file among two channels.

Algorithm	Split Time (ms)	Combine Time (ms)
Even Split	0.24	13.5
Segment Split	0.36	0.14

5.3.3 Comparison of the Multiple Channel-based Secure-only and Security Sieve Systems

Figure 5.13 displays the improvement in total time achieved by the security sieve system compared to the secure-only system when N is varied. In these experiments, N

equals *M,* and the files have *P* equal to 0.5. For comparison purposes, the improvement in mean total time achieved by the single channel-based security sieve system is also shown in the graph. Note that in the single channel cases no split/combine algorithms are used. This graph shows that for any number of channels, the security sieve system outperforms the secure-only system. However, as the number of channels increases, the improvement in mean total time seems to decrease, and then tends to flatten off when *N* is changed from 3 to 5. This is because the total time for both the secure-only and security sieve systems are dominated by the longer connection establishment times. This graph also shows that when transferring the 1MB-50 and 10MB-50 files, the SS algorithm outperforms the ES algorithm (as discussed in Section 5.3.2.2).

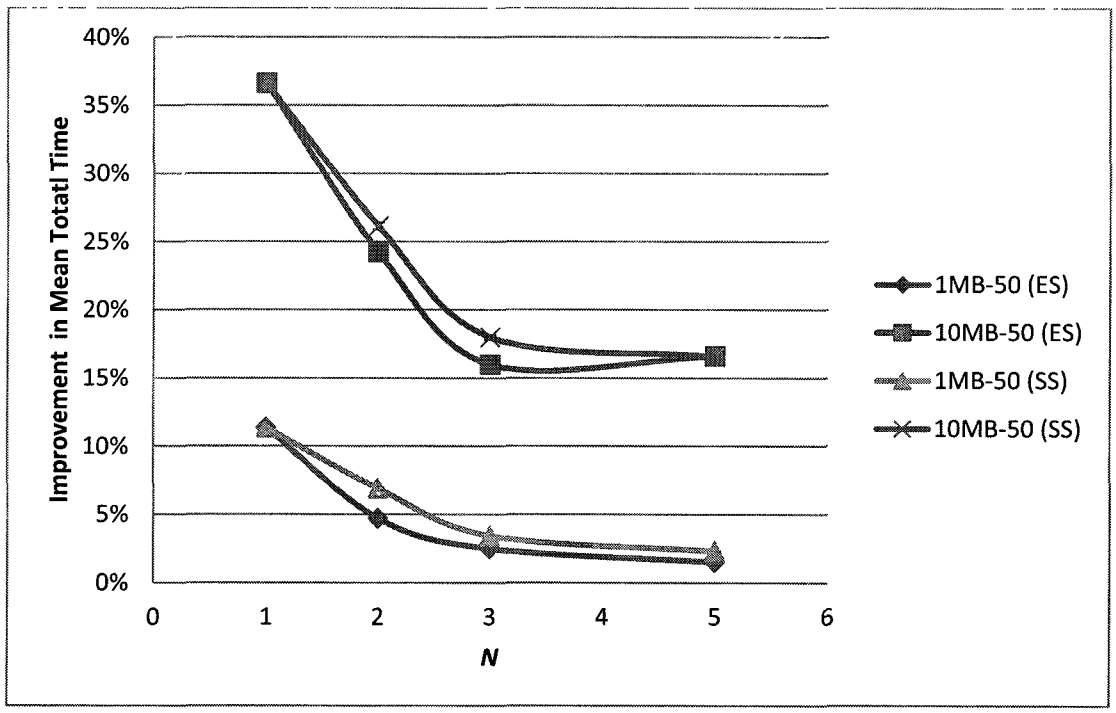


Figure 5.13: The effect of *N* on the improvement in mean total time achieved by the security sieve system.

5.4 Evaluation of Performance Optimization 2: Batching of Multiple Files

In this section, the secure-only and security sieve systems deploying PO2 are evaluated. There are two use cases where PO2 can be used. The first use case is when the client has multiple files available at the same time, and is ready to send the file set immediately. For example, a client can be a department in an enterprise with multiple files to send to a specific destination. Section 5.4.1 to Section 5.4.3, focus on experiments based on the first use case. In these experiments only one batch file transfer request is processed. These experiments are similar to those discussed in Sections 5.2 and 5.3. The only difference is that in these experiments a batch data list, which contains the data for multiple files, is transferred instead of just transferring a single file. There are two main purposes of these experiments: (1) to evaluate if using multiple channels is effective with PO2, and (2) evaluate the effectiveness of the batch split/combine algorithms developed for PO2 (discussed in Section 4.5).

In the second use case, there is an arrival of file transfer requests (potentially from different persons) that all have the same destination. In this use case, the client needs to wait for enough files to arrive for the formation of a batch before the batch file set containing the multiple files can be transferred. The results of the experiments based on the second use case are discussed in Section 5.4.4. The advantages and disadvantages of batching are shown by comparing: a security sieve system deploying PO2 with a security sieve system that does not deploy PO2.

5.4.1 Evaluation of Batch Split/Combine Algorithms for the Security Sieve System

This section evaluates the effectiveness of the batch split/combine algorithms used by the multiple channel-based security sieve system deploying PO2. Recall that the security sieve system has three batch split/combine algorithms: BES, BSS, and BFS (as discussed in Section 4.5). The effectiveness of the batch split/combine algorithms are evaluated by transferring two predefined file sets: an *uneven* file set (UFS), and an *even* file set (EFS). UFS contains the following files: a 1MB-10 file, a x-50 file, and a 1MB-90 file. EFS contains three x-50 files. The value of x is varied and can have the following file sizes (in MB): 1, 3, 5, 7, and 10. In EFS, all the files are identical, whereas in UFS, all the files are different. In UFS, each file has different values for x and P, which makes it more difficult to divide the data evenly among the multiple channels.

The graph in Figure 5.14 shows the mean total time achieved by the security sieve system when UFS is transferred, and x is varied. This graph compares the mean total time achieved by the single channel-based security sieve system, and a security sieve system that uses two channels. A graph showing the mean total time achieved by the security sieve system when transferring EFS, which has similar results to Figure 5.14, is presented in Appendix A (see Figure A.1).

As expected, for all cases, the mean total time increases with x because more data is being transferred. The mean total times achieved by the security sieve system when using the BSS and BES algorithm are nearly identical. Conversely, when the BFS algorithm is used, the mean total time is higher, especially for medium and high values of x. In fact, the mean total times achieved when using BFS are quite similar to the mean total times of

the single channel case. The lower performance can be attributed to the fact that the BFS algorithm, which does not partition the data within a file, is not able to divide the three files in UFS evenly between the two channels. Dividing the data evenly among the channels is important because it maximizes concurrency. As discussed in Section 5.3, using multiple channels helps improve response time because the security-related operations (such as encryption/decryption) can be performed by one thread while another thread is waiting to send/receive data. When x is 1MB, the single channel-based system outperforms the multiple channel-based systems because at smaller values of x, the improvement in performance achieved by using multiple channels is not enough to overcome the additional channel establishment/tear down overhead. As discussed in Section 5.3.2.1, using two channels tends to be most effective when at least 2MB of secure data is transferred.

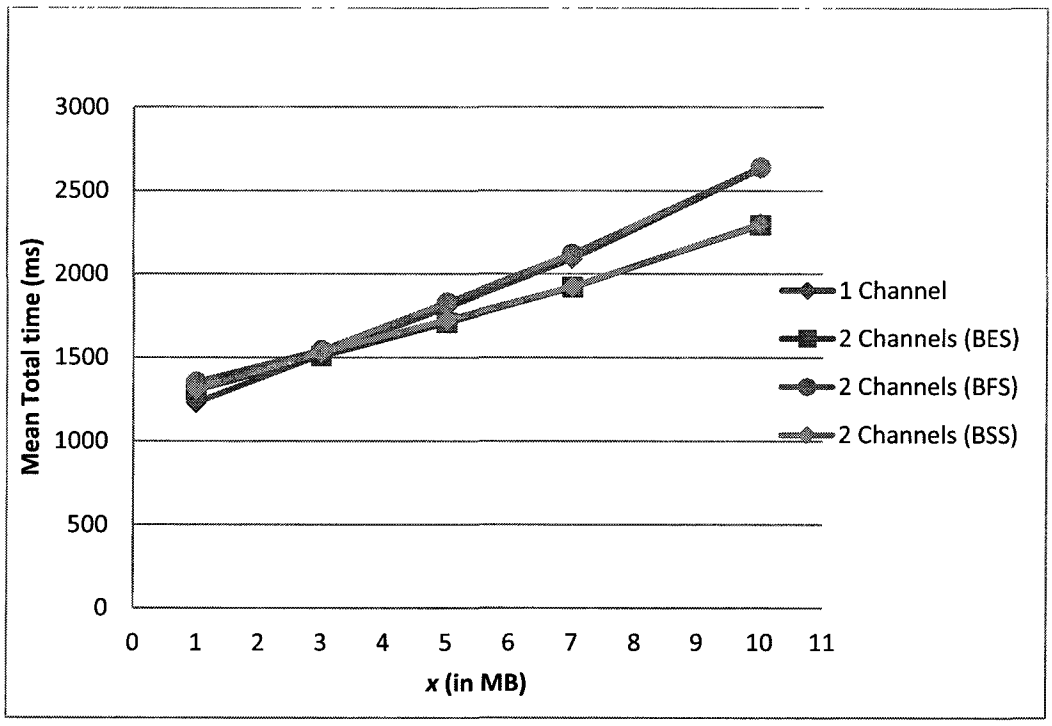


Figure 5.14: Mean total time achieved by the security sieve system when UFS is transferred.

5.4.2 Evaluation of the Batch Split/Combine Algorithms for the Secure-only System

This section evaluates the effectiveness of the batch split/combine algorithms used by the multiple channel-based secure-only system deploying PO2. Recall that the secure-only system has two batch split/combine algorithms: BFS and BES (as discussed in Section 4.5). The effectiveness of the batch split/combine algorithms are evaluated by transferring the two predefined file sets: UFS and EFS (similar to Section 5.4.1).

The graph in Figure 5.15 shows the mean total time achieved by the secure-only system when UFS is transferred using a single channel, and two channels. Similar trend in performance is achieved for EFS and is displayed in Figure A.2 of Appendix A. As expected, for all cases, the total time increases with x because more data is being transferred. In addition for all file sizes, the two channel-based systems outperform the single channel-based system because of concurrency of operations (as discussed in Section 5.3.1). When comparing the BFS and BES algorithms, the graph shows that a smaller total time is achieved when using the BES algorithm. When x is small (i.e. less data is being transferred), the total times are quite close, but as x increases, the performance improvement achieved with the BES-based system increases. The reason for this is because as file size increases, it becomes more difficult for the BFS algorithm to divide the data evenly between the two channels. For example, when x is 10MB, UFS contains two 1MB files and one 10MB file. The only way for the BFS algorithm to partition UFS is to assign the 10MB file to the first channel, and assign the two 1MB files to the second channel.

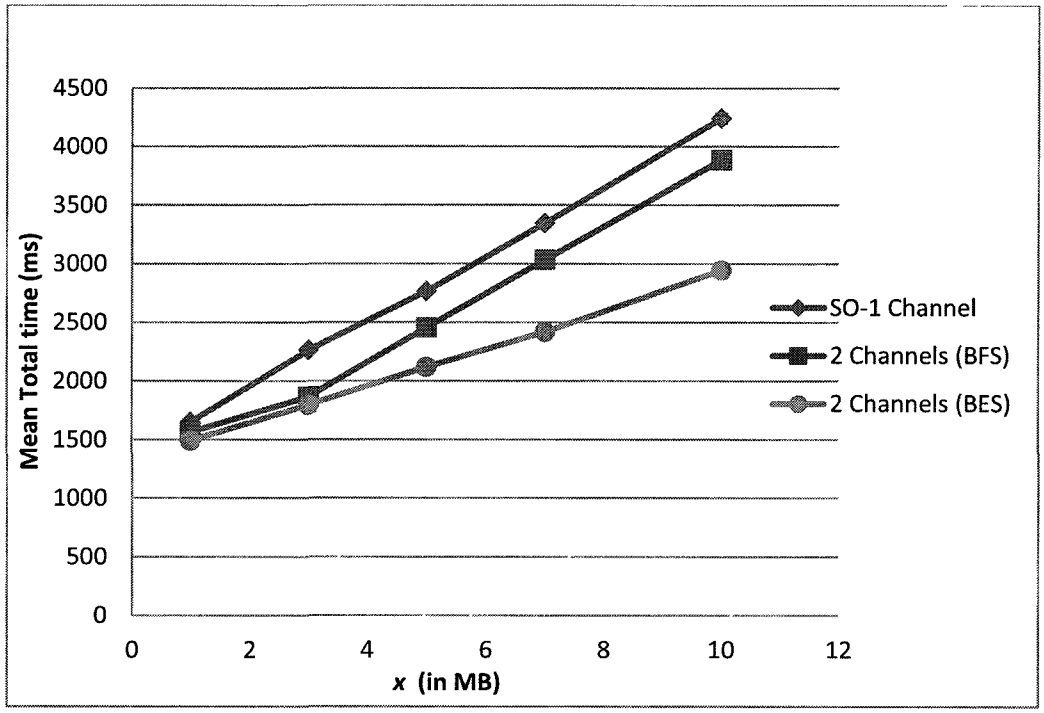


Figure 5.15: Mean total time achieved by the secure-only system when UFS is transferred.

5.4.3 Comparison of the Secure-only and Security Sieve Systems with Batching

Figure 5.16 presents the improvement in mean total time achieved by the security sieve system compared to the secure-only system when transferring the two pre-defined file sets, UFS and EFS. The improvement in mean total time shown in this graph is calculated using Eq. 1 (refer to Section 5.2). The values used in this calculation are taken from the cases that achieve the lowest total times among all the total times observed with various combinations of N, and the choice of the batch split/combine algorithms used.

For the secure-only system, the lowest total times were achieved when two channels and the BES algorithm are used. This is true for transferring both UFS and EFS. The cases for which the security sieve system achieved the lowest total times are

discussed next. When x is 1MB, for both UFS and EFS, the lowest total time is achieved by the single channel-based security sieve system. For all the other values of x, the lowest total times were achieved by the security sieve system using two channels. Specifically, for EFS, the lowest total time is achieved when the BSS algorithm is used, and for UFS, the lowest total time is achieved when the BES algorithm is used. As shown in Figure 5.16, when batching is used the security sieve system still outperforms the secure-only system. As x increases, the performance improvement increases as well, because more data is being transferred.

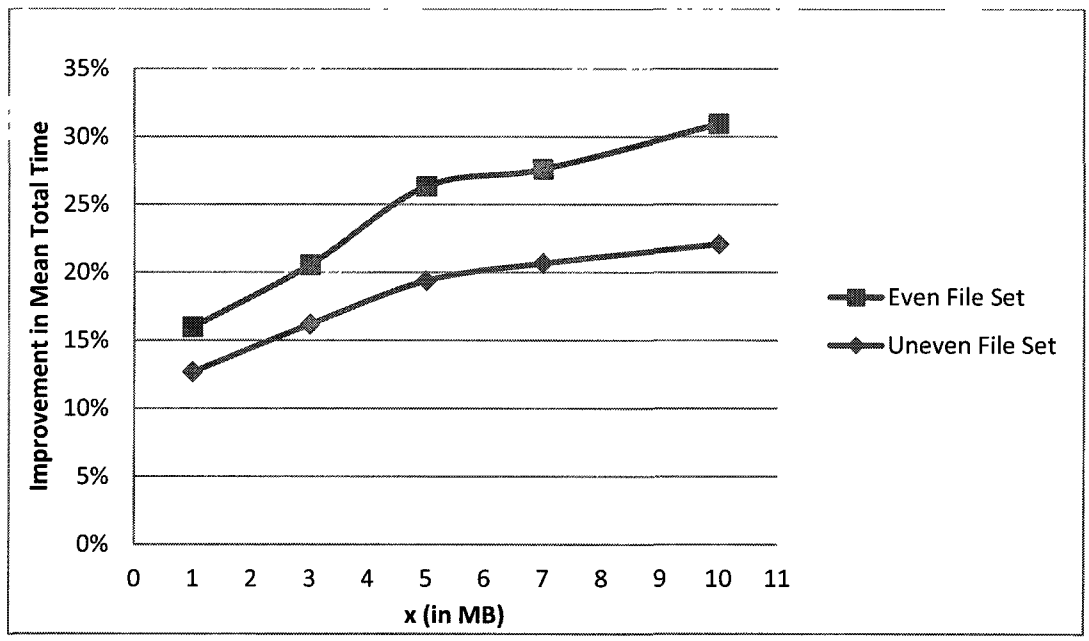


Figure 5.16: Improvement in mean total time achieved by the security sieve system when EFS and UFS are transferred.

5.4.4 Evaluation of PO2 on Systems subjected to Arrival of File Transfer Requests

In this section, PO2 is evaluated when a stream of file transfer requests arrives. Using simulation, the performance of two single channel-based security sieve systems is

compared: a system that batches multiple file transfer requests is compared to a system that does not use batching. Similar results are expected if the secure-only system is used instead of the security sieve system. The systems are subject to a Poisson stream of document transfer requests with an arrival rate, λ (in requests/second). Poisson arrival processes are commonly used to model the arrival processes of real systems [47]. If an arriving request is not able to be processed immediately, it is added to a first-come-first-serve queue.

In these simulation-based experiments, the systems process 2500 1MB-50 file transfer requests. Each experiment was run five times and the confidence intervals at 95% confidence level are calculated. The confidence intervals, which are all less than $\pm$ 2.5%, are shown in the graphs. Two threads were used to conduct the simulation: a traffic generator and an executor. The traffic generator produces file transfer requests at the specified arrival rate for the executor to process. Both the traffic generator and executor run concurrently. To simulate waiting for requests to arrive, the traffic generator invokes Java's `Thread.sleep()` method [44] to sleep for a duration equal to the inter-arrival time of a request. Similarly, the executor simulates processing a request by sleeping for duration equal to the mean total time required to transfer the file (or batch file set).

In the system that deploys batching, a batch size of five is used: the client waits for five requests to arrive before transferring the file set to server. The mean total time for transferring this batch file set containing five 1MB-50 files is approximately 1552 ms. For the security sieve system that does not use batching, one file is transferred at a time. The mean total time for transferring a single 1MB-50 file is approximately 933 ms. Note

that these mean total times were calculated from the results of previous experiments that were performed.

Table 5.3 compares the performance of the security sieve system using batching and the security sieve system without using batching at various arrival rates. Note that the mean turnaround time for a single file transfer request is used as a performance index in these experiments. The turnaround time is the difference between the time at which a file transfer is complete, and the time of arrival for the file transfer request. For a system using batching, the time at which a file transfer is complete is the same for all the files in the batch, and is equal to the time at which the transfer of the entire batch is formed. The turnaround time includes the total time and queuing delay. The queuing delay is the time that a request has to wait in the queue. The mean queue length is the average number of individual file requests that is in the queue.

Table 5.3: Batching vs. No Batching: Comparison of simulation results when arrival rate is varied.

Arrival Rate, λ (requests/s)	Mean Turnaround Time (s)		Mean Queue Length		Mean Queue Delay (s)	
	With Batching	No Batching	With Batching	No Batching	With Batching	No Batching
0.1	21.56	0.982	1.98	0.005	20.01	0.049
0.3	8.06	1.12	1.99	0.05	6.51	0.182
0.5	5.52	1.35	1.99	0.21	3.97	0.421
0.7	4.42	1.76	1.99	0.57	2.87	0.827
0.9	3.77	3.13	1.99	1.96	2.22	2.20
1	3.53	8.36	2.01	7.63	1.98	7.43
1.5	2.90	350.56	2.06	374.50	1.36	349.63
2	2.67	-	2.23	-	1.12	-
2.5	2.71	-	2.89	-	1.16	-
3	3.64	-	6.16	-	2.09	-
3.5	30.39	-	94.86	-	28.84	-

A graph comparing the mean turnaround times achieved by the two systems is shown in Figure 5.17. For the system not using batching, mean turnaround time increases with λ because as λ increases system contention increases, and more requests need to be queued. For the system using batching, at lower arrival rates, as λ increases, the mean turnaround time decreases. This is because the batch formation delay gets smaller as λ increases, allowing the system to process requests at a higher rate. The batch formation delay is the difference between the arrival time for the last request required to form the batch, and the time at which the first request for the batch arrives. However, for an arrival rate higher than 2.5 requests/s, the mean turnaround time starts increasing. This is because at higher arrival rates, although batches are formed faster, the queuing of batches starts occurring on the system. For higher values of λ, the batch queuing delay becomes more dominant than the batch formation delays, and the mean turnaround time increases (see the last three rows of Table 5.3, and Figure 5.18). Note that no entries are made for "No Batching" in rows 8-11 of Table 5.3 because the system with no batching is unable to handle such high arrival rates due to system saturation.

As λ increases, more and more queuing occurs on the system and the performance improvement due to using batching increases significantly. When λ is greater than approximately 1 request/s, the system that does not use batching starts to get saturated; whereas, the system that does use batching is still effective. For example, when λ is 1.5 requests/s, the mean turnaround time is approximately 350.56 seconds for the system that does not use batching, and approximately 2.9 seconds for the system that batches multiple file transfer requests. Batching is effective at high arrival rates because the batch

formation delays are smaller. The system using batching starts to get saturated when λ is greater than approximately 3 requests/s (see Table 5.3).

At very low arrival rates (e.g. λ = 0.1 requests/s), the system that does not use batching achieves a lower mean turnaround time. This is because the system with batching has to wait for five file transfer requests to arrive before the batch is formed, and the files can be transferred to the server. This tends to increase the queuing delays for the four requests that arrived earlier, thus increasing the turnaround time for those requests as well.

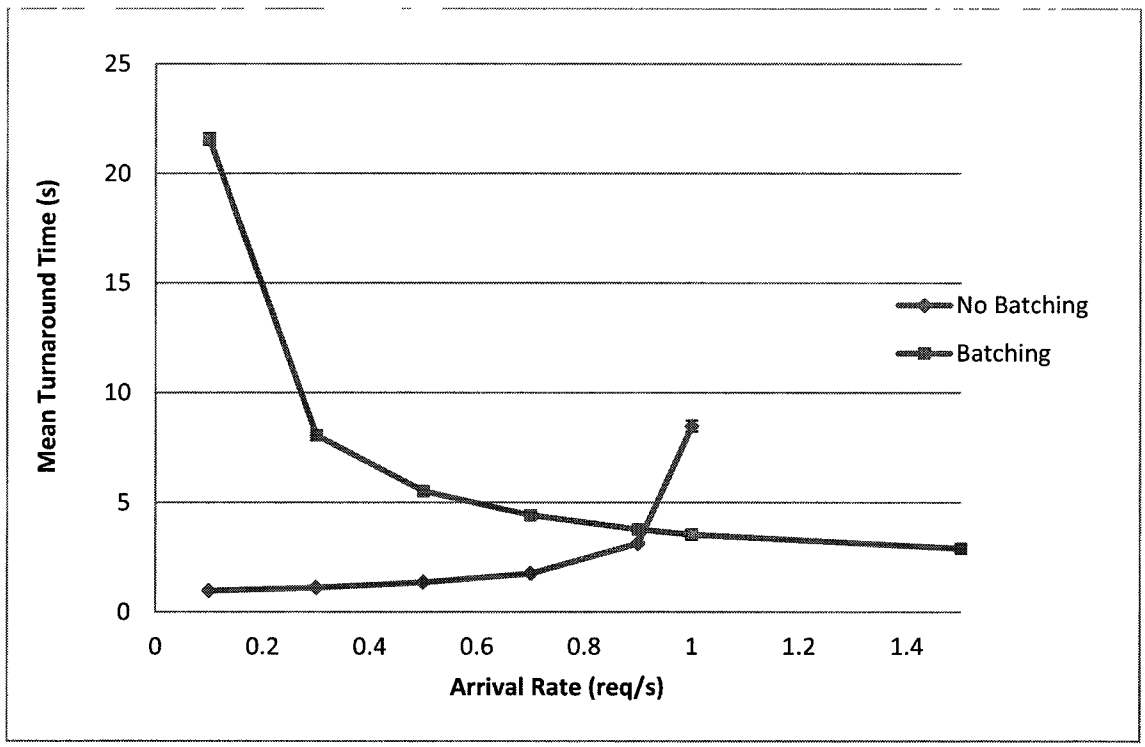


Figure 5.17: Comparison of the mean turnaround time achieved by the security sieve system using batching, and the security sieve system that does not use batching.

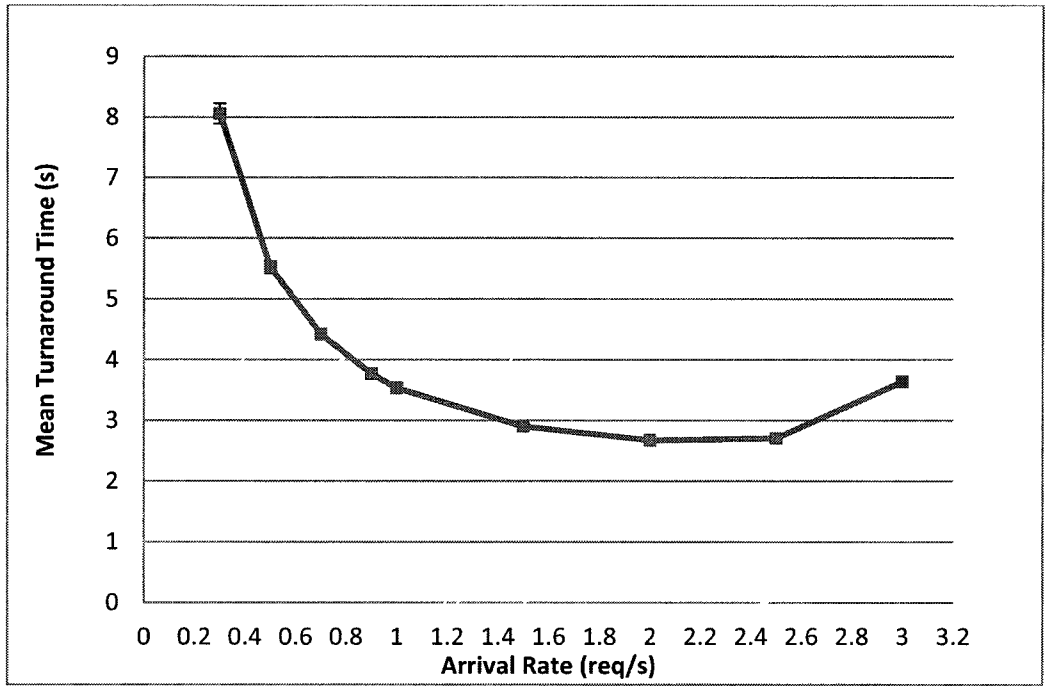


Figure 5.18: The mean turnaround time achieved by the security sieve system using batching when λ is varied.

5.5 Discussion of Experimental Results

This section discusses the experimental results for both the security sieve and secure-only systems. As expected, the performance improvement achieved by the security sieve system compared to the secure-only system increases, as P decreases. The experimental results showed that when P is less than approximately 95% (and x is at least 1MB) the security sieve system starts to outperform the secure-only system. The reason for the performance improvement is because the security sieve technique minimizes the time spent performing the security-related operations (e.g. encryption/decryption), which in turn reduces the response times. In addition, when using security sieve, concurrency occurs. For example, when the thread servicing the non-secure channel is waiting to send data, the thread servicing the secure channel can be encrypting data. Recall, that the send thread has to wait for the receive thread to read data from the buffer before more data is

transmitted. The improvement in mean total time is always a little lower compared to improvement in mean response time because of: (1) the mean secure channel connection establishment time, which is included in the total time for both the security sieve and secure-only systems, and (2) the additional non-secure channel connection time overhead.

When using AES-256 instead of 3DES, the improvement in mean response time decreases, but is still significant. The reason for the lower performance improvement is because AES is naturally a faster and more efficient encryption algorithm. Therefore, when using the AES algorithm the performance penalty for sending data securely is smaller, which is more advantageous to the secure-only system since it transmits more data over the secure channel.

The performance improvement achieved by security sieve also decreases when moving the client to a wireless environment. This can be attributed to the long data transmission times dominating the response times. In other words, the response time now depends more on the data transmission time and to a less extent on the time required to perform the security-related operations, such as encryption and decryption. However, when the processing capability of the client is reduced (i.e. power saving option is used), which increases the processing time required to perform the security-related operations, the dominance of the data transmission time on the response time is reduced, and the performance improvement achieved by the security sieve system increases. This shows that the security sieve technique can be effective in systems that need to save power.

5.5.1 Impact of Concurrency

The experimental results show that PO1 is effective in both the secure-only and security sieve systems. Using multiple channels helps improve response time because of

concurrency. The send threads have to wait for the receive threads to finish reading data from the buffer before more data is transmitted. Thus, the security-related operations (such as encryption/decryption of data) can be performed by one thread while another thread is waiting to send or receive data. However, there is a minimum amount of data that needs to be transferred for the multiple channel-based systems to achieve a smaller total time than the single channel-based systems. When the amount of data transferred is very small, the improvement in performance achieved by using multiple channels is not enough to overcome the additional channel establishment/tear down overhead. The experimental results showed that the lowest mean total time is achieved by the systems using two channels. Using more than two channels is not effective because the additional establishment overhead is too high. As discussed in Section 5.3.2.1, when using two channels, the experimental results show that approximately 2MB of secure data needs to be transferred for the system to achieve an improvement in response time that is high enough to offset the additional channel establishment/tear down overhead.

The experimental results show that the most effective splitting/combining algorithm is the algorithm that is able to divide the data most evenly among the channels with the least overhead. Recall that the overhead of a splitting/combine algorithm includes the processing times required to invoke the split and combine methods. When comparing the split/combine algorithms (used in PO1), the results show that the ES algorithm performs better than the SS algorithm when the majority of the data in a file is classified (i.e. when P is greater than 0.6). This can be attributed to the fact that when more secure data is transferred, it is more important for each channel to send an equal amount of data so that concurrency can be maximized.

5.5.2 Effect of Batching Requests

When evaluating systems subject to a stream of file transfer requests, the experimental results show that performance improvement achieved with batching increases with an increase in arrival rate. Specifically, the experimental results presented in Figure 5.17, show that when the arrival rate is greater than approximately 0.9 requests/s, the system using batching starts to outperform the system that does not use batching. At high arrival rates, there is a significant improvement achieved with batching because the system using batching can process requests at a higher rate that the system not using batching. This reduces the mean queuing delay. At very low arrival rates, batching is not as effective because of the higher batch formation delays. The system using batching has to wait for multiple requests to arrive to form a batch, before the files can be sent. This increases the queuing delay of the earlier arriving requests.

When comparing the batch split/combine algorithms, experimental results also show that the most effective splitting/combining algorithm is the algorithm that is able to divide the data evenly among the channels with the least overhead. The experimental results seem to indicate that the BFS algorithm is only able to divide a batch file set, containing files with the exact same sizes, if at least one of these two conditions is true: (1) the number of files in the batch is divisible by the number of channels used, or (2) the number of files in the batch is less than or equal to the number of channels used. In general, the BFS algorithm is not able to evenly partition a file set containing files with different sizes. However, there are specific cases where the BFS algorithm can evenly divide a file set containing different files sizes. For example, a file set containing these 3

files: 3MB, 7MB, and 10MB, can be evenly divided between two channels using the BFS algorithm.

However, none of the conditions discussed above need to be met for the BES algorithm because the BES algorithm does partition the data within a file, unlike the BFS algorithm. The disadvantage of using the BES algorithm is that it has the highest overhead among all the batch split/combine algorithms. Thus, it should only be used when the other algorithms are not effective. The BSS algorithm, which is exclusive to the security sieve system, has similar performance to the BES algorithm, and also has a smaller processing overhead (see Table B.1-B.3 in Appendix B). When transferring most file sets with the security sieve system, using the BSS algorithm is a good choice, except for the extreme cases where the file sizes in the file set are very different, or the majority of the data in the files is classified.

Chapter 6: Conclusions

This chapter summarizes the thesis and presents conclusions. Future work and directions for further research are also discussed.

6.1 Summary and Conclusions

This thesis introduces a technique, called security sieve, for enhancing the performance of SSL/TLS-based document transmission. Specifically, security sieve improves the response time of transferring classified documents (which may also contain non-sensitive information) by reducing the number of security-related operations (i.e., encryption/decryption and hashing) that needs to be performed on the transmitted data. This is achieved by separating the classified and non-classified information in a document, and transferring the separated information over a secure channel and a (faster) non-secure channel, respectively. This thesis follows the approach where the author of the document identifies the classified information in the document.

A prototype of a client-server security sieve system has been developed, and performance measurements made on the prototype demonstrates the effectiveness of the security sieve technique. A significant performance improvement was observed when the performance of the security sieve system was compared to the performance of a secure-only system. The secure-only system represents the state of the art: all the data (sensitive or non-sensitive) is transmitted using a single secure channel. Files with various sizes, and proportion of classified information were experimented with. For the workload and system parameters experimented with, the security sieve system outperforms the secure-only system when the proportion of classified information in a document is less than approximately 95%. The security sieve system, which minimizes the security-related

operations that need to be performed, achieves the highest performance improvement over the secure-only system when the processing time required to perform the security-related operations is high. This is the reason why the security sieve system was more effective when 3DES is used (instead of AES), and when a wireless client characterized by a lower processing capability is used.

This research also explored using two other performance optimization techniques: (1) PO1: adding multiple channels to achieve concurrency of operations, and (2) PO2: batching of multiple files transfer requests to amortize the channel establishment/tear down overhead over multiple file transfers. These two performance optimization techniques were deployed in both the secure-only and security sieve systems, and experiments were conducted to investigate how system performance is affected.

When evaluating PO1, experimental results show that using multiple channels is effective in reducing response times due to an increase in concurrency of operations. For example, the encryption/decryption of data can be performed by one thread while another thread is waiting to send or receive data. Recall that the send threads have to wait for the receive threads to empty the buffer before more data is transferred. Specifically, the results of the experiments performed with various workload and system parameters (see Section 5.3) show PO1 tends to be most effective when using two channels, and sending at least 2MB of secure data. When file sizes are small using multiple channels is not effective because the improvement in response time is offset by the additional channel establishment/tear down overhead.

To evaluate the effectiveness of batching, the performance of a single channel-based security sieve system that batched five file transfer requests was compared to a

single channel-based security sieve system that did not use batching. The systems were subject to a Poisson stream of file transfer requests at various arrival rates. The experimental results show that as the arrival rate increases, the performance improvement due to using batching increases significantly. Specifically, for the workload and system parameters experimented with, the results showed that when the arrival rate is greater than approximately 0.9 requests/s, the system using batching starts to outperform the system that does not use batching. At very low arrival rates, the system that does not use batching is more effective because the system using batching has to wait for multiple requests to arrive to form a batch, before the files are transferred. This increases the waiting time of the earlier arriving requests, and thus increases the turnaround time for those requests as well.

Experiments were also conducted to explore how the secure-only and security sieve systems using batching (and batching with multiple channels) perform when only a single batch file set is transferred. Two predefined file sets containing various files were experimented with. The transmission of the two predefined batch files set were used to evaluate the effectiveness of using multiple channels with PO2, and evaluate the effectiveness of the batch split/combine algorithms. The experimental results show that using multiple channels is effective when enough data is transferred, and that the most effective splitting/combining algorithm is the algorithm that is able to divide the data evenly among the channels with the least overhead. In general, the BES algorithm is most effective for the secure-only system, and the BSS algorithm is most effective for the security sieve system.

6.2 Future Work

Directions for future research are discussed next.

- Devise techniques for automatically marking a document. For example, it may be possible to automatically identify classified information based on a set of keywords.
- Adapt the security sieve technique to handle structured data (e.g. databases), and other types of data files such as MS Word and PDF files that can contain text and images.
- Investigate how performance is affected when using processors with multiple cores, and understand the impact that the number of cores has on system performance.
- Experiment with security sieve on mobile devices, and investigate the impact of device mobility on performance.
- Investigate multiple channel-based security sieve systems where the number of secure channels is not equal to number of non-secure channels.

Automatically marking a document based on a set of keywords is an interesting direction for future research. Such a technique can be based on a database of keywords and/or phrases that are associated with confidential information. When such a keyword (or phrase) is found in the document, the chapter, page, paragraph, or sentence containing the keyword (or phrase) can be identified as sensitive information. Whether a chapter, a page, a paragraph, or a sentence, in which a keyword (or phrase) appears, is to be marked as confidential information can be based on a set of policies that is specified by the user. To handle documents containing text and images, the entire page containing a confidential image, which may be difficult to separate from the text in the page, can be marked as classified information. A tool that searches the document and marks the confidential data can be used prior to submitting the document to a security sieve system.

References

[1] Z. Li, R. Iyer, S. Makineni and L. Bhuyan, "Anatomy and Performance of SSL Processing," *International Symposium on Performance Analysis of Systems and Software*, Austin, Texas, USA, March 2005, pp.197-206.

[2] T. Dierks and C. Allen, "The TLS Protocol Version 1.0", *IETF RFC 2246*, January 1999. Available at: http://www.ietf.org/rfc/rfc2246.txt.

[3] N. Lim, S. Majumdar and V. Srivastava, "Engineering SSL-Based Systems for Enhancing System Performance." *International Conference on Performance Engineering (ICPE)*, Karlsruhe, Germany, March 2011, pp. 469-474.

[4] The SANS Technology Institute, "Security Laboratory: SSL/TLS," [Online]. Available at: http://www.sans.edu /resources/securitylab/ssl_tts.php. [Accessed January 2011].

[5] Cicso Systems, Inc., "White Paper: Introduction to Secure Sockets Layer," 2002.

[6] Tropical Software, "Triple DES Encryption," [Online]. Available at: http://www.tropsoft.com/strongenc/des3.html. [Accessed Jan. 2011].

[7] NIST, "Advanced Encryption Standard (AES)," *FIPS Pub.197*, November 2001. Available at: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[8] VOCAL Technologies, Ltd. "RC4 Encryption Algorithm," [Online]. Available at: http://www.vocal.com/cryptography/rc4.html [Accessed July 2011].

[9] R.L. Rivest, A. Shamir and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 21, February 1978, pp 120-126.

[10] D. Eastlake and P. Jones, "US Secure Hash Algorithm (SHA1)," *IETF RFC 3174*, September 2001. Available at: http://www.ietf.org/rfc/rfc3174.txt.

[11] Oracle Corporation, "X.509 Certificates and Certificate Revocation Lists," [Online]. Available at: http://download.oracle.com/javase/1.5.0/docs/guide/security/cert3.html [Accessed October 2010].

[12] K. Kant, R. Iyer and P. Mohapatra, "Architectural impact of secure socket layer on Internet servers," *International Conference on Computer Design*, Austin, Texas, USA, September 2000, pp.7-14.

[13] A. Goldberg, R. Buff and A. Schmitt, "A Comparison of HTTP and HTTPS Performance," *Computer Science Department, New York University*, 1998.

[14] P.G. Argyroudis, R. Verma, H. Tewari and D. O'Mahony, "Performance analysis of cryptographic protocols on handheld devices," *Third IEEE International Symposium on Network Computing and Applications*, Cambridge, Massachusetts, USA, August 2004, pp. 169- 174.

[15] D. Berbecaru, "On Measuring SSL-based Secure Data Transfer with Handheld Devices," *2nd International Symposium on Wireless Communication Systems*, Siena, Italy, September 2005, pp.409-413.

[16] W. Chou, "Inside SSL: the secure sockets layer protocol," *IT Professional*, vol.4, no.4, pp. 47- 52, Jul/Aug 2002.

[17] N.R. Potlapally, S. Ravi, A. Raghunathan and G. Lakshminarayana, "Optimizing public-key encryption for wireless clients," *International Conference on Communications*, New York, New York, USA, May 2002, vol.2, pp. 1050- 1056.

[18] J. Lopez and R. Dahab, "An Overview of Elliptic Curve Cryptography," *Technical Report*, Institute of Computing State University of Campinas, May 2002.

[19] V. Gupta, D. Stebila and S. Fung, "Speeding Up Secure Web Transactions Using Elliptic Curve Cryptgraphy," *11th Annual Network and Distributed System Security Symposium*, San Diego, California, USA, 2004, pp. 231–239.

[20] B. Wang, H. Zhang and Y. Wang, "An Efficient Elliptic Curves Scalar Multiplication for Wireless Network," *Network and Parallel Computing Workshops,* Dalian, China, Sepetmeber 2007, pp.131-134.

[21] M. Portmann and A. Seneviratne, "Selective security for TLS," *Ninth IEEE International Conference on Networks*, Bangkok, Thailand, October 2001, pp. 216-221.

[22] A. Almuhaideb, M. Alhabeeb, P.D. Le and B. Srinivasan, "Beyond Fixed Key Size: Classifications Toward a Balance Between Security and Performance," *24th IEEE International Conference on Advanced Information Networking and Applications*, Perth, Austrailia, April 2010, pp.1047-1053.

[23] Y. Song, V. Leung and K. Beznosov, "Supporting End-to-end Security Across Proxies with Multiple Channel SSL," *IFIP18th World Computer Conference*, Toulouse, France, August 2004, p. 32.

[24] S. Tak. And E. Park, "Adaptive secure software architecture for electronic commerce," *Software: Practice and Experience, vol. 33, no.* 14, pp. 1343-1357, November 2003.

[25] H.M. Alaidaros, M.F.A. Rasid, M. Othman and R.S.A. Abdullah, "Enhancing security performance with parallel crypto operations in SSL bulk data transfer phase," *Telecommunications and Malaysia International Conference on Communications,* Penang, Malaysia, May 2007, pp.129-133.

[26] E. Nahum, E., D.J. Yates, S. O'Malley, H. Orman, R. Schroeppel, "Parallelized network security protocols," *The Symposium on Network and Distributed System Security,* San Diego, California, USA, February 1996, pp.145-154.

[27] Oracle Corporation, "NetBeans IDE," [Online]. Avaialble: http://netbeans.org/. [Accessed October 2010].

[28] Oracle Corporation, "Class SSLSocket," [Online]. Available at: http://download.oracle.com/javase/6/docs/api/javax/net/ssl/SSLSocket.html. [Accessed October 2010].

[29] Oracle Corporation, "Java Secure Socket Extension: Reference Guide," [Online]. Available: http://download.oracle.com/javase/6/docs/technotes/guides/security/jsse/JSSERefG.uide.html [Accessed October 2010].

[30] Oracle Corporation, "Class BufferedReader," [Online]. Available at: http://download.oracle.com/javase/1.4.2/docs/api/java/io/BufferedReader.html. [Accessed October 2010].

[31] Oracle Corporation, "Class BufferedWriter," [Online]. Available at: http://download.oracle.com/javase/1.4.2/docs/api/java/io/BufferedWriter.html. [Accessed October 2010].

[32] Oracle Corporation, "Class SSLServerSocketFactory," [Online]. Available at: http://download.oracle.com/javase/6/docs/api/javax/net/ssl/SSLServerSocketFactory.html. [Accessed October 2010].

[33] Oracle Corporation, "Class SSLServerSocket," [Online]. Available at: http://download.oracle.com/javase/1.4.2/docs/api/javax/net/ssl/SSLServerSocket.html [Accessed October 2010].

[34] Oracle Corporation, "keytool - Key and Certificate Management Tool," [Online]. Available at: "http://download.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html. [Accessed October 2010].

[35] Oracle Corporation, "All about Sockets," [Online]. Available at: http://download.oracle.com/javase/tutorial/networking/sockets/. [Accessed October 2010].

[36] Oracle Corporation, "Class SSLSocket," [Online]. Available at: http://download.oracle.com/javase/1.4.2/docs/api/javax/net/ssl/SSLSocket.html [Accessed October 2010].

[37] Oracle Corporation, "Class ArrayList<E>," [Online]. Available at: http://download.oracle.com/javase/6/docs/api/java/util/ArrayList.html. [Accessed October 2010].

[38] Oracle Corporation, "Class ServerSocket," [Online]. Available at: http://download.oracle.com/javase/6/docs/api/java/net/ServerSocket.html. [Accessed October 2010].

[39] Oracle Corporation, "Class SSLSocketFactory," [Online]. Available at: http://download.oracle.com/javase/6/docs/api/javax/net/ssl/SSLSocketFactory.html. [Accessed October 2010].

[40] Oracle Corporation, "Class System," [Online]. Available at: http://download.oracle.com/javase/1,5.0/docs/api/java/lang/System.html. [Accessed November 2010].

[41] P. Chown, "Advanced Encryption Standard (AES) Ciphersuites for Transport Layer Security (TLS)", *IETF RFC 3268*, June 2002. Available at: http://tools.ietf.org/html/rfc3268.

[42] J. Reavis, " Feature: Goodbye DES, Hello AES," *Network Word*, 2001. Available at: http://www.networkworld.com/research/2001/0730feat2.html.

[43] B. Butter, ACE Team, "AES vs. 3DES Block Ciphers," [Online]. Available at: http://blogs.msdn.com/b/ace_team/archive/2007/09/07/aes-vs-3des-block-ciphers.aspx [Acessed July 2011].

[44] Oracle Corporation, "Class Thread," [Online]. Available at: http://download.oracle.com/javase/1.4.2/docs/api/java/lang/Thread.html#sleep(long). [Accessed May 2011].

[45] Tech Autos, "Making Sense of Smartphone Processors: The Mobile CPU/GPU Guide," March 2010. [Online]. Available at: http://www.techautos.com/2010/03/14/smartphone-processor-guide/. [Accessed July 2010].

[46] Phonegg, "Fastest Processor: Cell Phones Top List," [Online]. Available at: http://www.phonegg.com/Top/Fastest-Processor-Cell-Phones.html. [Accessed July 2010].

[47] J. Banks; J.C. Carson; B.L. Nelson; D.M.Nicol, "Discrete-Event System Simulation," 5th Edition, Pearson – Prentice Hall, 2010, ISBN 0-13-606212-1.

Appendices

Appendix A : Additional Performance Evaluation Results

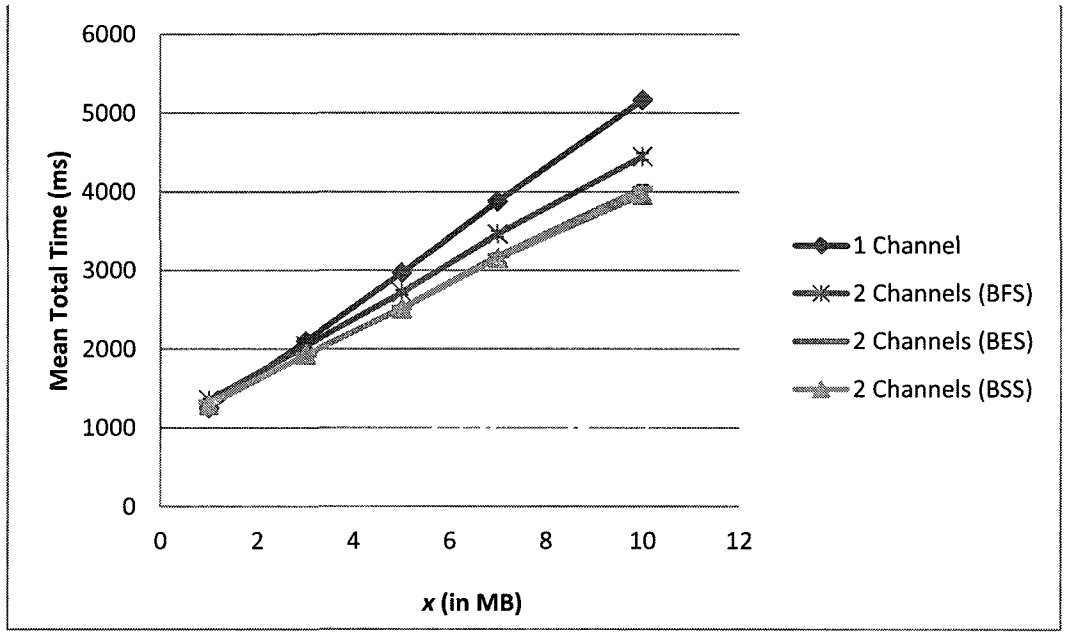


Figure A.1: Mean total time achieved by the security sieve system when EFS is transferred.

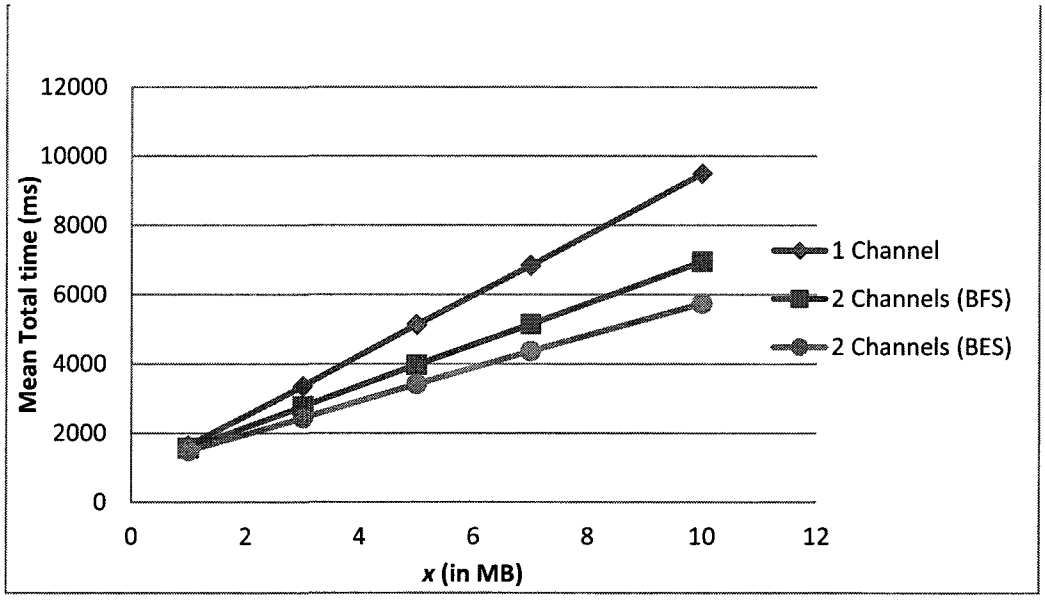


Figure A.2: Mean total time achieved by the secure-only system when EFS is transferred.

Appendix B : Sample Execution Times for the Security Sieve Batch Split Algorithms

The following tables present the execution times required for the BFS, BSS, BES, to divide UFS when x is 10MB. Note that the time required to split/combine the non-secure and secure batch data lists are added together and shown as a single value.

Table B.1: Execution time required to invoke the batchFileSplit() and batchFileCombine() methods on UFS with x =10MB.

Number of Channels (*N=M*)	Split Time (ms)	Combine Time (ms)
2	0.14	0.05
3	0.12	0.07
5	0.12	0.07

Table B.2: Execution time required to invoke the batchSegmentSplit() and batchSegmentCombine() methods on UFS with x =10MB.

Number of Channels (*N=M*)	Split Time (ms)	Combine Time (ms)
2	0.38	0.14
3	0.39	0.19
5	0.39	0.26

Table B.3: Execution time required to invoke the batchEvenSplit() and batchEvenCombine() methods on UFS with x =10MB.

Number of Channels (*N=M*)	Split Time (ms)	Combine Time (ms)
2	0.18	18.94
3	0.21	20.23
5	0.28	22.58