

Engineering SSL-Based Systems for Enhancing System Performance

Norman Lim

Real Time and Distributed Systems
Research Centre,
Dept. of Systems and Computer Eng.,
Carleton University, Ottawa,
CANADA.

nlim@sce.carleton.ca

Shikharesh Majumdar

Real Time and Distributed Systems
Research Centre,
Dept. of Systems and Computer Eng.,
Carleton University, Ottawa, CANADA
+1-613-731-5296,

majumdar@sce.carleton.ca

Vineet Srivastava,

Cistech Limited
210 Colonnade Road, Unit#3
Ottawa, CANADA.
vineet@cistech.ca

ABSTRACT

Security in a distributed system often comes at the cost of a performance penalty. Due to the CPU time consuming security algorithms used, transferring data using SSL is known to be significantly slow. This paper presents an initial set of research results of a university-industry collaborative research focusing on a performance enhancement technique called security sieve that separates the classified and non-classified components in a document and sends these on a secure and a (faster) non-secure channel respectively. Experimental results presented in the paper demonstrate the effectiveness of the technique.

Categories and Subject Descriptors

D4 Software, D.4.4 Message sending, D4.8 Modeling and prediction, D.4.8 Measurements

General Terms

Performance, Design, Security, performance optimization, performance of security.

Keywords

Secure Sockets Layer (SSL), SSL performance, performance engineering of SSL, performance optimization, security system performance.

1. INTRODUCTION

Performance optimization as well as performance modeling and analysis are important components of performance engineering. Existing work analyzing the performance of the Secure Sockets Layer (SSL) protocol shows that the incorporation of system security slows down data transfer rates significantly. This research concerns engineering SSL-based systems for enhancing system performance.

Classified documents are often transmitted over secure channels that perform various security related operations (discussed in more detail later in this section) such as data encryption and decryption that consume a significant amount of processing time resulting in long document transmission times. Such classified documents are typically characterized by both classified and non-classified components. A classified component may correspond to

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICPE'11, March 14–16, 2011, Karlsruhe, Germany.

Copyright 2011 ACM 978-1-4503-0519-8/11/03...\$10.00.

a chapter, paragraph or even a single sentence that needs to be protected. The non-classified components contain information that need not be protected. For example, only the name, the social insurance number and the address of a patient may need to be protected when sending a lengthy document on a patient's medical history. Determining which components are classified is a responsibility of the user transmitting the document. In some cases the user may decide that protecting only certain key components as indicated in the previous example may be sufficient. In other cases designating additional information that can lead to the revelation of the classified information may be required. Sending only the classified components over the secure channel and sending the remaining components over (a faster) non-secure channel can potentially reduce the overall document transmission time and give rise to bandwidth savings. This paper proposes a *security sieve* that separates the non-classified components from the classified components and transmits them over non-secure and secure channels, respectively. The components are re-assembled at the receiving end to reconstruct the original document. The secure components can be "marked" by the author of the document. It may also be possible to automatically identify them based on a set of keywords. For the work discussed in this paper a "marked by author approach" is followed. Devising techniques for automatically marking a document can form an interesting direction for future research. A short discussion of SSL including a representative set of works on related performance issues is presented next.

SSL (also known as the Transport Layer Security, TLS, protocol) is one of the common protocols used for providing secure communications between clients and servers over the Internet [9]. There are three main aspects of security [5], [9]: (1) confidentiality (and privacy), (2) message integrity, and (3) authentication. Confidentiality ensures the information that is transferred cannot be seen by a third party, whereas message integrity ensures that the information is not modified during the transfer. Authentication ensures that the communicating parties are the entity they claim they are. SSL uses cryptography, digital signatures, and certificates to provide confidentiality, integrity, and authentication, respectively. There are two types of cryptography: symmetric (private key) cryptography (SC) and asymmetric (public key) cryptography (AC). SC, which uses the same key for encryption and decryption, is commonly used to encrypt data for bulk data transfers. AC uses different keys for encryption and decryption, and is commonly used to exchange the private key used for SC. The SSL protocol has two main phases [5]: (1) the handshake phase and (2) the bulk data transfer phase. The handshake phase consists of cipher suite negotiation,

authentication, and secret key exchange. The cipher suite specifies the algorithms to use for: authentication, bulk data encryption/decryption, and secret key exchange. The secret key exchange mutually establishes the private key that is going to be used for bulk data encryption/decryption. In the bulk data transfer phase, encrypted data is exchanged between the client and the server. Each message exchanged is appended with a message authentication code (MAC) (also known as a digital signature) to ensure data integrity.

Sending data through a secure channel can result in long transfer times due to the CPU-intensive operations that need to be applied to the data. These expensive operations include: operations performed by cryptography algorithms (to encrypt/decrypt data), and hash algorithms (to sign the data) [9]. Additional overheads are also incurred when opening (e.g., during the handshake phase) and closing the SSL channel. The performance impact of using SSL/TLS and the cryptography algorithms it uses was extensively studied in [2], [3], [8], [9]. As a result, much effort has been invested into developing techniques to improve the performance of secure communications. One common technique to improve the performance of secure communications is to speed up the crypto algorithms. There are two common approaches to speedup crypto operations [5]: (1) use specialized hardware accelerators with dedicated modular arithmetic units [4], and (2) use software optimized crypto algorithms and techniques (e.g. [13]). There have also been other approaches to improving secure communications. In [1], [12], and [16] the concept of selective security is used. The idea is to apply security only to the most sensitive information. A key benefit of selective security is that the use of the costly crypto operations can be reduced by applying the operations only to the data that requires it. The authors of [1] proposed a Dynamic Key Size (DKS) architecture that can be integrated into security protocols to provide more efficient secure mobile communications. The proposed approach uses information sensitivity level (specified by the user) and device capability to select a suitable algorithm key length. In [12], the authors proposed a simple extension to SSL/TLS protocol by specifying a new record layer type for the TLS protocol stack. The new record type, Cleartext Application Data, is used to transport the non-classified documents. The main purpose of the virtual cleartext channel is to allow the non-secure information to be exposed to any intermediate system for content adaptation purposes. An extension to the SSL protocol called multiple channel SSL (MC-SSL) was proposed in [16]. The idea was to have multiple channels between the client and server at different levels of security for various levels of data sensitivity. With MC-SSL the client is able to negotiate multiple SSL channels each with its own security characteristics (e.g. cipher suite).

This research concerns engineering performance into the data transmission software that is responsible for the transfer of classified documents between two sites. The proposed *security sieve* approach is also based on the concept of selective security; however the main difference between our approach and existing research in [1], [12], and [16] is that these other techniques transfer separate documents with different security requirements over different channels. Our work however, focuses on the transmission of a single classified document using the standard SSL/TLS protocol and introduces a mechanism to separate and recombine the non-classified and the classified components in the document. The secure components are transferred over a secure

channel whereas the non-secure components over a non-secure channel. To the best of our knowledge such a technique has not been deployed by systems described in the existing literature. This paper presents some preliminary results of the research. The contributions of the paper include the following:

- *Security Sieve*: a technique for separating/re-combining the non-classified components from the classified components in a document is introduced. An experimental demonstration of the performance improvement produced by security sieve is provided.
- A discussion of the design and implementation of a prototype client-server system that supports the security sieve technique is presented.
- Performance evaluation of the security sieve technique, comparing the prototype system with a client-server system that uses a single secure channel is performed. A queuing network-based analysis is then used to investigate the impact of queuing on system performance. Insights gained into system behavior are described.

The rest of the paper is organized as follows. A detailed description of the security sieve technique is provided in Section 2. In Section 3, we describe the implementation of a prototype client-server system that supports the security sieve technique. Section 4 focuses on the performance evaluation of the security sieve technique compared to a client-server system that uses a single secure channel. Lastly, Section 5 presents our conclusions and plans for future work.

2. SECURITY SIEVE

The security sieve technique is explained with the help of Figure 1. First the original data (document to be transferred) is sieved (or separated) into classified and non-classified components. The non-classified data is sent using the non-secure channel and the classified information is sent using the secure channel (using SSL/TLS). At the receiving end, the data is reconstructed by integrating the separated data components received from the secure and non-secure channels.

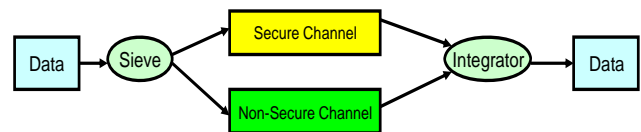


Figure 1: Overview of Security Sieve.

2.1 Sieve and Integration Algorithms

This section explains the *sieve* and *integration* algorithms (see Section 2.1.1-2.1.2). In this short paper we have used ASCII text files as the classified documents to be transferred. The performance benefits observed are expected to accrue for other types of documents as well. Adapting our techniques to handle other types of data files such as MS Word and PDF files is currently underway. To distinguish between classified and non-classified data, a document is marked by the user with the following tags: (1) *Secure Start Tag*, $\langle \$\$ \rangle$: indicates where the classified data begins and (2) *Secure End Tag*, $\langle \$E \rangle$: indicates where classified data ends. Furthermore, the classified and non-classified data are stored in two lists: (1) *Non-secure List*: stores the non-classified data and (2) *Secure List*: stores the classified data components (transmitted securely).

2.1.1 Sieve Algorithm

The sieve algorithm separates the classified and non-classified components in the original document, and stores the appropriate data in the *Secure List* or *Non-secure List* (as shown in Figure 2). The sieve algorithm uses two indices to specify the position that is being examined in the document: (1) *Current Index*: stores current position in the document, and (2) *Previous Index*: stores previous current index position in the document.

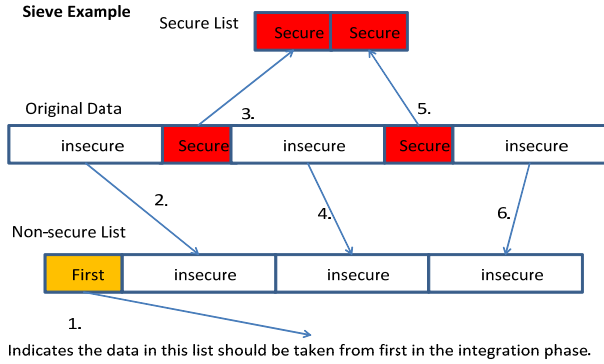


Figure 2: Example of sieve algorithm.

First, it is determined whether the original document starts with classified or non-classified data by checking for the `<$$>` tag. The *First List Tag*, `<!First!>`, is added to the appropriate list so that the receiver knows which list, data should be taken from first during the integration phase. If the document starts with classified data, the algorithm searches for the `<$E>` tag, stores the position in the *Current Index*, and copies the data between the *Previous Index* and the *Current Index* (i.e. the text segment between the `<$$>` and `<$E>` tags), and stores copied data in the *Secure List*. The *Previous Index* is updated by assigning the *Current Index* value to it. Next, the document is searched for more *Secure Start Tags* starting from the *Current Index* position. If the tag is found, the data between the *Previous* and *Current* indices is copied, and stored in the *Non-secure List*. The *Previous Index* is then updated again. Next, the algorithm searches for `<$E>` and the data between the `<$$>` and `<$E>` tags is copied, and added to the *Secure List*. The operations continue in this fashion until there are no more `<$$>` tags found. The algorithm then checks if the end of the document has been reached and if not the data between the *Previous Index* and end of the document is copied, and added in the *Non-secure List*.

2.1.2 Integration Algorithm

The integration algorithm puts the separated data elements contained in the *Secure* and *Non-secure Lists* back to its original order in a new list called the *Integrated Data List (IDT)*. First, the first entry of the *Secure* and *Non-secure lists* is checked to determine which list contains the *First List Tag*. If the first list is the *Secure List*, the first data element from *Secure List* is removed, and is added to the *IDT*. Otherwise, the first list is the *Non-secure List*, and only the *First List Tag* needs to be removed since the second phase of the algorithm starts by removing data items from the *Non-secure List*. The second phase algorithm involves removing data items from the *Non-secure* and *Secure lists* (alternating between the lists in sequence) and stores them in the *IDT* until both *Secure* and *Non-secure lists* are empty. This restores the original sequence of data segments since the *Secure* and *Non-secure lists* also keep the data segments in order.

3. PROTOTYPE IMPLEMENTATION

Java was used to implement a prototype security sieve client-server system. Section 3.1 discusses the security sieve client, whereas Section 3.2 discusses the security sieve server. The sieving operations are performed by the client who transmits the components of the classified document, and the integration of the components is performed by the server.

3.1 Security Sieve Client

A class diagram of the security sieve client implemented for our prototype is shown in Figure 3. A 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. 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. Details about the client connection establishment are discussed in Section 3.1.1. The *askForInputFile()* and *readInputFile()* private methods are used to get input from the user, and read the document that will be sent to the server. The *sieve()* method implements the sieve algorithm discussed in Section 2.1.1. The *send()* method writes the supplied data to the specified channel (explained further in Section 3.1.2). *ArrayList* objects (provided by Java's Collections Framework) are used to implement the *Secure* and *Non-secure lists*.

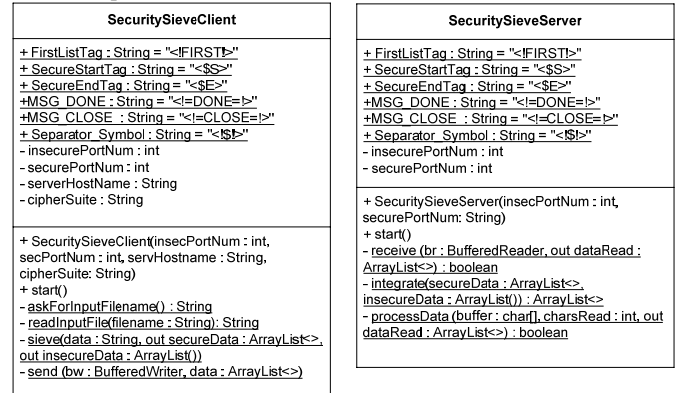


Figure 3: Class diagram of the security sieve client and server.

3.1.1 Client Connection Establishment

The non-secure channel is established using Java's TCP Socket API [10], and the secure channel is established using Java's SSL/TLS Socket API. The SSL/TLS socket API is a part of the Java Secure Socket Extension (JSSE) [11]. First, the client attempts to establish the non-secure channel with the server by creating a *Socket* object (using Java's *Socket* constructor). The non-secure channel is established after the server accepts the connection. Next, the client tries to establish a secure channel with the server by invoking the *createSocket()* method (using Java's *SSLConnectionFactory* API) with the hostname and secure port number private attributes as the parameters. Once the server accepts the connection, the secure channel is established, and the client sets the cipher suite to be used, and starts the SSL/TLS handshake. After both channels are established, the client retrieves the input/output (I/O) streams from both the secure and non-secure sockets, and creates *BufferedReader* and *BufferedWriter* objects (part of Java's I/O package) which are used to read/write data to/from the channels.

3.1.2 Client send() Method

A sequence diagram of the `send()` method is shown in Figure 4. The send method writes the supplied data list to a channel using the specified `BufferedWriter` (created during the connection establishment phase). Each element (or text segment) in the data list is sent separately by sending a `Separator Symbol`, `<!$!>`, between elements in the list (as shown in Figure 4). After all the data is written, the `MSG_DONE` message is sent to inform the receiver that all the data has been written. Each element in the data list is sent separately because the `BufferedWriter/BufferedReader` objects cannot write/read `ArrayList` objects. The `receive()` method is explained in Section 3.2.2.

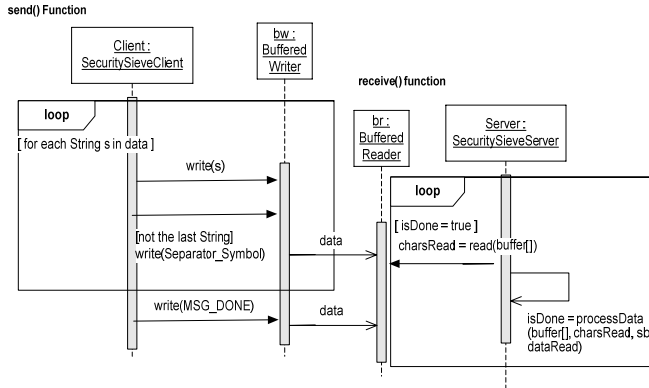


Figure 4: Sequence diagrams of security sieve client `send()` and server `receive()` methods.

3.2 Security Sieve Server

A class diagram of the security sieve server implemented for our prototype is shown in Figure 3. The server is created by invoking the `SecuritySieveServer()` constructor, which accepts two parameters: (1) the port number for the non-secure socket, and (2) the port number for the secure socket. 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. The server connection setup is explained in more detail in Section 3.2.1. The `receive()` method reads data from the specified channel using the supplied `BufferedReader` and stores the data in the given `dataRead` parameter. The `processData()` method is invoked by the receive method, and is explained in more detail in Section 3.2.2. The `integrate()` method implements the integration algorithm discussed in Section 2.1.2. As in case of the client, `ArrayList` objects are used to implement the *Secure and Non-secure* lists.

3.2.1 Server Connection Setup

To listen for client requests, the server creates a secure (SSL/TLS) server socket by invoking the `createServerSocket()` method (provided by `SSLServerSocketFactory` API) using the `securePortNum` private attribute. Next, the non-secure (TCP) socket is created using the `ServerSocket()` constructor (provided by Java's `ServerSocket` API) using the `insecurePortNum` attribute. Afterwards, the server waits for incoming client connection requests, and invokes the `accept()` method (on the socket objects) to accept the client requests. The secure and non-secure channels between the client and the server are now established. As in the case of the client, the server then retrieves the I/O streams from the sockets, and creates `BufferedReader` and `BufferedWriter` objects.

3.2.2 Server receive() Method

A sequence diagram of the `receive()` method is shown in Figure 4. The receive method reads 8192 characters from the channel at time via the supplied `BufferedReader`. The text segment that is read is passed to the `processData()` method. This method continually stores the text segments in a temporary String until a `Separator Symbol`, which separates the elements in the data list that was sent, is found. Once found, the data in the temporary String is added to the supplied data list. The `processData()` method returns `true` if `MSG_DONE` is received to indicate all the data has been read; otherwise, `false` is returned, and reading continues. The goal is to construct a local `ArrayList` that is identical to the original `ArrayList` that the client sent.

4. PERFORMANCE EVALUATION

To evaluate the performance of the security sieve client-server system, we compared it to a conventional *secure-only* client-server system. The secure-only system is an example of the state of the art: the client sends the entire document to the server over a secure channel. The goal is to compare the end-to-end response time of sending a document, which contains both classified and non-classified data, using the security sieve technique and using the conventional approach of sending the entire file securely. The client and server were connected via a LAN using a 100Mbps Ethernet connection. The client runs on a computer equipped with a 2.0 GHz Intel Core 2 Duo CPU and 2.0 GB RAM, running under the Windows 7 operating system. The server computer uses a 3.2 GHz Intel Core 2 Duo CPU and 2.0 GB RAM, running under the Windows 7 operating system. The exact nature of the machines is not important in the context of this research and we expect similar relative performances of the systems under comparison when different machines are used. In the system deploying the security sieve technique, the client creates two threads: one thread each to send the non-classified data and classified data. Similarly, the security sieve server uses two threads: one thread to read data from the non-secure channel, and the other thread to read from the secure channel. The cipher suite used for the secure channels was `SSL_RSA_WITH_3DES_EDE_CBC_SHA`. Rivest Shamir Adleman (RSA) algorithm [14] is used for public key cryptography and key exchange (1024-bit key size). The Triple DES (3DES) algorithm [15] was used for private key cryptography. The Data Encryption Standard (DES) uses a 56-bit key. Triple DES applies the DES algorithm three times to each data block using different keys each time (for an effective key strength of 168-bit) [15]. Secure Hash Algorithm (SHA) is used for hashing to ensure message integrity [6].

The experiments consisted of performing a document file transfer between two machines: one acting as the client and the other acting as the server. There were two file sizes used in the data transfer: 1MB, and 10MB. For the security sieve client-server experiments, the files were transferred using various percentages, P, of classified information in the document: 10, 50, and 90 percent. We have used synthetic ASCII text documents for achieving this. The files contain the appropriate number of characters (and appropriate tags) but do not have any semantic value. In the experiments described a 1MB-P file contained 5 equal segments of secure and 5 equal segments of non-secure segments. The length of a segment is computed from the length of the file and P. A 10 MB-P file is obtained by copying the data generated for a 1MB file 10 times. We plan to investigate other

distributions of data segments in the future. For each file size, multiple experimental runs were conducted. The number of runs was chosen in such a way such that a confidence interval of $\pm 0.5\%$ at a confidence level of 95% was achieved. The client makes one file transfer, records the *total time*, and then closes the channel(s). The total time measurements include both the *connection establishment time* and *response time*. To measure the response time, the *data transfer time* is measured first. A timestamp is taken (using the *nanoTime()* method provided by Java's System class API) before the data is sent, and when the client receives an ACK from the server. The data transfer time is the difference between the two timestamps. For the secure-only system the response time is the same as data transfer time, and the connection establishment time includes the time to setup the secure channel, and the handshake time. In the experiments with the security sieve system, in addition to the data transfer time, the response time also includes the time it takes to sieve the file, and integrate the components (re-combine the file). These values are measured by taking a timestamp before and after the respective method (*sieve()* or *integrate()*) is called, and then taking the difference between the timestamps. For security sieve system, the connection establishment time includes the time to setup the secure channel, the handshake time, as well as the time to setup the non-secure channel. The average response time, and average total time of each file transfer was calculated and plotted in Figure 5 and Figure 6, respectively. Note that the "-P" suffix in an x-axis label means that P% of the data in the file was confidential and needed to be transmitted over the secure channel. Note that for the secure-only system, the entire document is transmitted over a single secure channel, and both the average response time and the average total time depend on the size of the files transferred but are independent of P, the proportion of secure data in the files.

For all the 1 MB and 10 MB file transfers, the client-server system using the security sieve technique outperforms the single secure channel client-server system. The reason for the increased performance is that fewer CPU-intensive security algorithm operations needed to be executed which translated to shorter response times. This improvement in response time was large enough to overcome the non-secure channel connection time, and sieve/integration time overheads. The average sieve and integration times for the 1MB files were 1.72 ms and 0.01 ms, respectively; and for the 10MB files, they were 16.9 ms and 0.03 ms, respectively. The average non-secure channel connection establishment time was measured to be 25 ms. As expected; the largest improvement in performance is gained when transferring a file with a small percentage of classified information. For the 1MB-10 and 10MB-10 files, the improvement in average response time was 67%, and 68%, respectively. For the 1MB-90 and 10MB-90 files, for which most of the data is classified, the improvement in average response time is lower: 7% and 8% respectively. When we examine the average total time (see Figure 6), security sieve improves the average total time only by 18% and 54% for the 1MB-10 and 10MB-10 files, respectively. The decrease in improvement (compared to Figure 5) is due to the following: (1) the secure channel connection time and handshake time (on average 755 ms) makes up a large part of the total time for both the security sieve and secure-only cases, and (2) the additional non-secure channel connection time overhead incurred due to the security sieve. The reason for the larger decrease in improvement for the 1MB-10 file compared to the 10MB-10 file is because when transferring a smaller file, the connection

establishment overhead makes up a more significant part of the total time compared to the response time; therefore, improving the response time does not have as much impact on the overall total time improvement. Note that on systems in which multiple documents are transferred between a given site the overheads related to channels set up are incurred only once and the response time becomes the performance metric of interest.

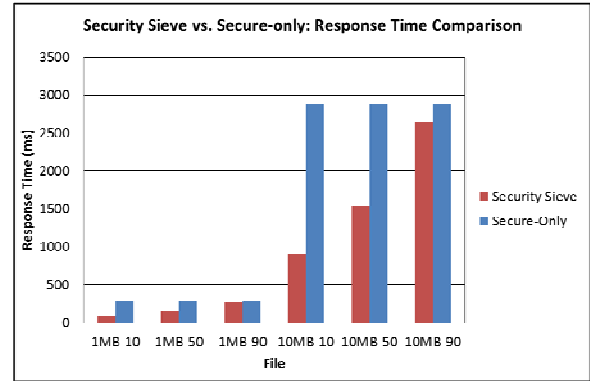


Figure 5: Security sieve vs. Secure-only response time comparison.

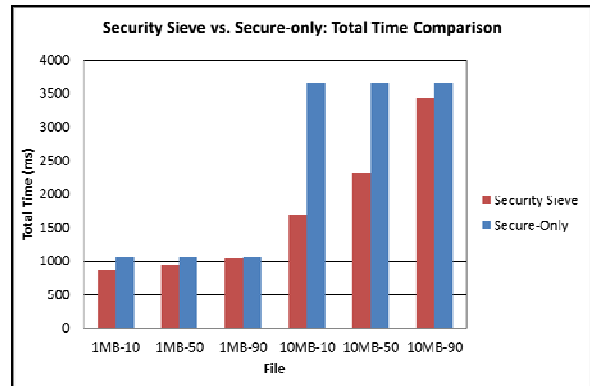


Figure 6: Security sieve vs. Secure-only total time comparison.

The performance improvement from the security sieve is dependent on the size of the document and P. Analyzing system performance to determine the minimum document size required for achieving a performance improvement for various values of P form an important direction for further research.

4.1 Effect of Queuing

The performance analysis presented in Section 4 focused on systems in which a single transfer request is processed at a time. A preliminary analysis of the impact of queuing delays incurred on systems in which multiple document transfer requests compete for the secure and non-secure channels is presented in this section. We have used a separable single class open queueing network model (QNM) [7] in this preliminary analysis for modeling both the security sieve and the secure-only systems that are subjected to a Poisson arrival stream of document transfer requests with an arrival rate of λ . As used in the queueing analysis of many real systems we have assumed that the requirements for the application of a separable QNM are met. The single server in each system models all the operations performed by the corresponding system. Using the response times obtained from the measurements on a single file transfer reported in Section 4 as service times (D_{SS})

for the security sieve system and D_{SO} for the secure-only system), the average response times can be computed as [7]:

$$R(\text{security sieve}) = D_{SS}/(1 - \lambda D_{SS}) \quad (1a)$$

$$R(\text{secure-only}) = D_{SO}/(1 - \lambda D_{SO}) \quad (1b)$$

Such an analysis that ignores the connection establishment time is important in the context of systems in which the receiver of the document is the same and multiple transfers are performed over the same connection. A sample graph for the 1MB-50 file is presented in Figure 7. As the arrival rate increases, more and more queuing occurs on the system and the performance improvement due to security sieve increases significantly. A similar relative performance between the two systems is expected when the total time is used as service times in the QNM. Such an analysis is useful when each document transfer involves a separate server and a new connection needs to be established for each transfer. Clearly, the utility of the security sieve increases substantially on systems that transfer multiple documents between sites. Almost an order of magnitude of improvement in performance is observed at $\lambda = 0.003$ requests/ms.

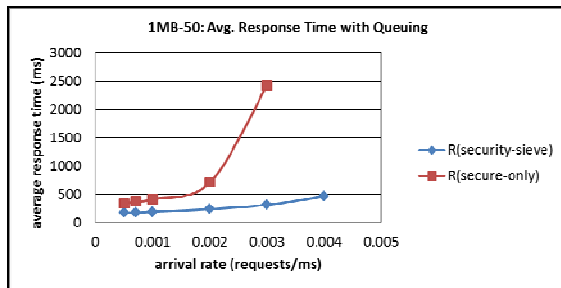


Figure 7: 1MB-50 file average response time with queuing analysis.

5. SUMMARY AND CONCLUSIONS

This short paper introduces a security sieve technique for engineering secure document transfer systems for enhancing system performance. The technique separates the document into secure and non-secure components and transfers them over a secure and a non-secure channel respectively. An initial prototype of the system has been built. Performance measurements made on the prototype demonstrates the effectiveness of the security sieve that leads to a significant performance improvement. A single class QNM based analysis is performed to investigate performance of systems handling a stream of multiple document transfer requests arriving on the system. The modeling results show that the performance improvement that accrues from the security sieve increases substantially with the request arrival rate. Further research is being planned and includes the following:

- Improving the system design by incorporating additional performance optimization techniques such as batching of multiple requests to be transferred, and using parallel transmission over multiple secure and non-secure channels.
- A detailed modeling of the prototype using a layered queuing model [7] for identifying and stretching software bottlenecks. This will include modeling of systems that do not satisfy the assumptions underlying separable queuing networks.
- Devising a tool for marking document components as classified, a graphical user interface, and an API (to be used when the request for transfer is made by an application).

- Devising techniques for automatically marking a document, and adapting our algorithms to handle other types of data files such as MS Word and PDF files.

6. REFERENCES

- [1] Almuhaideb, A.; Alhabeeb, M.; Le, P.D; Srinivasan, B., "Beyond Fixed Key Size: Classifications Toward a Balance Between Security and Performance," 24th IEEE International Conference on Advanced Information Networking and Applications, pp.1047-1053, 20-23 April 2010.
- [2] Argyroudis, P.G.; Verma, R.; Tewari, H.; O'Mahony, D., "Performance analysis of cryptographic protocols on handheld devices," Third IEEE International Symposium on Network Computing and Applications, 2004., pp. 169- 174, 30 Aug.-1 Sept. 2004.
- [3] Berbecaru, D., "On Measuring SSL-based Secure Data Transfer with Handheld Devices," 2nd International Symposium on Wireless Communication Systems, 2005., pp.409-413, 7-7 Sept. 2005.
- [4] Chou, W., "Inside SSL: the secure sockets layer protocol," IT Professional, vol.4, no.4, pp. 47- 52, Jul/Aug 2002.
- [5] Cisco Systems, Inc. "White Paper: Introduction to Secure Sockets Layer," 2002.
- [6] D. Eastlake and P. Jones, *US Secure Hash Algorithm (SHA1)*, IETF RFC 3174, September 2001; <http://www.ietf.org/rfc/rfc3174.txt>.
- [7] E. D. Lazowska, J. Zahorjan, G. S. Graham, K. C. Sevcik, Quantitative System Performance, Prentice Hall, 1984.
- [8] Kant, K.; Iyer, R.; Mohapatra, P., "Architectural impact of secure socket layer on Internet servers," International Conference on Computer Design, pp.7-14, 2000.
- [9] Li, Z.; Iyer, R.; Makineni, S.; Bhuyan, L., "Anatomy and Performance of SSL Processing," International Symposium on Performance Analysis of Systems and Software, 2005., pp.197-206, 20-22 March 2005.
- [10] Oracle Corporation, "All about Sockets," [Online]. Available: <http://download.oracle.com/javase/tutorial/networking/socket/s/> [Accessed October 16, 2010].
- [11] Oracle Corporation, "Java Secure Socket Extension: Reference Guide," [Online]. Available: <http://download.oracle.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html> [Accessed Oct. 16, 2010].
- [12] Portmann, M.; Seneviratne, A., "Selective security for TLS," Networks, 2001. Ninth IEEE International Conference, pp. 216- 221, 10-12 Oct. 2001.
- [13] Potlapally, N.R.; Ravi, S.; Raghunathan, A.; Lakshminarayana, G., "Optimizing public-key encryption for wireless clients," International Conference on Communications, 2002., vol.2, pp. 1050- 1056.
- [14] R. L. Rivest, A. Shamir, and L. Adleman. "A method for obtaining digital signatures and public-key cryptosystems," Commun. ACM 21, 2 (February 1978), 120-126.
- [15] The SANS Technology Institute, "Security Laboratory: SSL/TLS," [Online]. Available: http://www.sans.edu/resources/securitylab/ssl_tts.php [Accessed Jan. 7, 2011].
- [16] Y. Song, V. Leung, and K. Beznosov, "Supporting End-to-end Security Across Proxies with Multiple Channel SSL," in IFIP18th World Computer Conference (WCC'2004), Toulouse, France, 2004, pp. 32.