Securing Electronic Commerce: Reducing the SSL Overhead

George Apostolopoulos
Siara Systems

Vinod Peris
Growth Networks

Prashant Pradhan
State University of New York, Stony Brook

Debanjan Saha
Lucent Bell Lab

Abstract

The last couple of years have seen a growing momentum towards using the Internet for conducting business. Web based electronic commerce applications are one of the fastest growing segments of the Internet today. A key enabler for e-commerce applications is the ability to setup secure private channels over a public network. The Secure Sockets Layer (SSL) protocol provides this capability and it is the most widely used security protocol in the Internet. In this article, we take a close look at the working principles behind SSL with an eye on performance. We benchmark two of the popular web servers that are in wide use in a number of large e-commerce sites. Our results show that the overheads due to SSL can make web servers slower by a couple of orders of magnitude. We investigate the reason for this deficiency by instrumenting the SSL protocol stack with a detailed profiling of the protocol processing components. In light of our observations, we outline architectural guidelines for large e-commerce sites.

1 Introduction

Security is important on the Internet. Whether sharing financial, business, or personal information, people want to know with whom they are communicating (authentication), they want to ensure that what is sent is what is received (integrity), and they want to prevent others from eavesdropping their communications (privacy). The Secure Sockets Layer (SSL) protocol [9] provides one means for achieving these goals. It was designed and first implemented by Netscape Corporation as a security enhancement for their Web servers and browsers. Since then, almost all vendors and public domain software developers have integrated SSL in their security sensitive client-server applications. At present, SSL is widely deployed in many intranets as well as over the public Internet in the form of SSL-capable servers and clients and has become the de facto standard for transport layer security. Recently, the Internet Engineering Task Force (IETF) has started an effort to standardize SSL as an IETF standard under the name of Transport Layer Security (TLS) protocol [1].

One of the reasons that SSL has outgrown other transport and application layer security protocols such as SSH [18], SET [17], and SMIME [11] in terms of deployment is that it is application protocol independent. Conceptually, any application that runs over TCP can also run over SSL. There are many examples of applications such as TELNET and FTP running transparently over SSL. However, SSL is most widely used as the secure transport layer below HTTP [5]. A large number of e-commerce sites dealing with private and sensitive information use SSL as the secure transport layer. This number is expected to grow as more and more businesses and users embrace electronic commerce. As security becomes an integral feature of Internet applications and the use of SSL rises, its impact on the performance of the servers as

well the clients is going to be increasingly important. The objective of this paper is to take a close and critical look at the SSL protocol with an eye on performance.

The SSL protocol is composed of two main components: the SSL handshake protocol and the SSL record protocol. The handshake protocol is responsible for authenticating the communicating peers to each other. It is also entrusted with the job of negotiating encryption and message authentication algorithms along with the required keys. SSL allows the session state to be cached. If a client needs to setup a new SSL session while its session state is cached at the server, it can skip the steps involving authentication and key exchange and reuse the cached session state to generate a set of keys for the new session. The Record Protocol provides two basic security services: privacy and message integrity.

In the rest of the paper, we analyze the performance impact of SSL on Web based e-commerce applications and quantify the overhead associated with different components of SSL. To measure the performance impact of SSL on Web servers running e-commerce applications, we have modified the SPECweb96 [15] benchmark to generate client workload for servers and clients running HTTP transactions over SSL. Using this modified SPECweb96 benchmark, we evaluated the performance of two different secure Web servers that are in wide use in a large number of e-commerce sites. Our results show that depending on the degree of session reuse the overhead due to SSL can decrease the rate at which the server can process HTTP transactions by upto two orders of magnitude. To identify the overhead associated with different components of SSL, we have instrumented and traced the SSL handshake protocol and SSL record protocol using timers with sub-micro second granularity. Our results indicate that for a typical HTTP transaction (10-15 Kbytes), the bulk of the overhead comes from the SSL handshake protocol. For very large HTTP transactions (1 Mbytes or more), the cost of the handshake is amortized over the length of the transfer and the dominant part of the overhead is due to data encryption and message authentication.

We also investigate the performance impact of SSL on large e-commerce sites which use clusters of servers for reasons of scalability. In a typical installation, the server cluster is front-ended by a dispatcher (Figure 8) responsible for distributing connections across the nodes in the cluster [6]. Most often the dispatcher is agnostic of SSL session level information. As a result, SSL connections that can potentially reuse session states are routed to different server nodes. Unless the server nodes share their session caches, dispatching leads to poor SSL session reuse efficiency and consequently poor performance. We quantify the impact of SSL session reuse in a cluster environment and propose techniques to alleviate the problem.

To understand the details of the SSL protocol it is useful to have a working knowledge of basic cryptography. In section 2 we review a few basic cryptographic operations that are used in SSL. Next, the various message flows of the SSL protocol are described in section 3. Section 4 is devoted to the evaluation and analysis of SSL protocol performance and its impact on Web based e-commerce servers. In section 5, we address the impact and the importance of SSL session reuse in large e-commerce sites. We summarize in section 6.

2 Cryptography Basics

Conventional cryptography a.k.a. symmetric key cryptography has been used by mankind for several centuries. The most common cryptographic techniques involve a secret that is used in both the encryption and decryption of the message. The message that is to be encrypted, referred to as plaintext, in Figure 1, is input to an encryption algorithm, e.g. DES (Data Encryption Standard) [2]. In addition to the plaintext, a secret key which in the case of DES is a 56-bit binary number, is also input to the encryption algorithm.

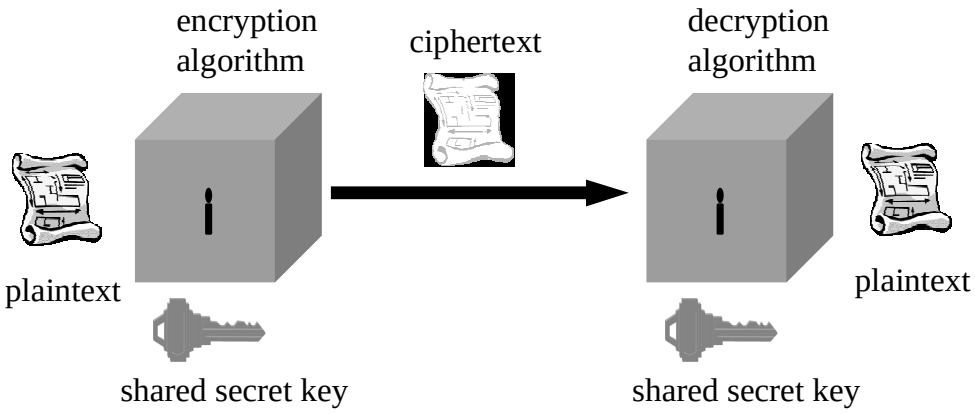


Figure 1: Block diagram depicting conventional cryptography

The resulting output is the encrypted message, commonly referred to as ciphertext. At the other end, the ciphertext as well as the secret key is input to a decryption algorithm which outputs the plaintext. Typically, the decryption algorithm is very similar to the encryption algorithm and the secret key is the same for encryption and decryption, hence the name symmetric key cryptography. A basic requirement of encryption algorithms is that it should be computationally very hard to obtain the plaintext from the ciphertext without knowledge of the secret key. Ideally, the ciphertext should appear as a random sequence of bits with very little correlation to the plaintext. In addition, a good cipher should have the property that a single bit change in the plaintext results in large number of bits being modified in the ciphertext.

There are many different kinds of ciphers and they can be broadly classified into (1) block ciphers which operate on a block of input data at a time and (2) stream ciphers which operate on the input on a bit-by-bit basis. For example, DES is a block cipher that operates on 64 bits at a time whereas RC4 [14] is a stream cipher that operates on the input on a bit-by-bit basis. Alternatively a stream cipher can be defined as a block cipher where the block size is 1 bit.

One of the deficiencies of a simple block cipher scheme is that if the same plaintext block is repeated the corresponding ciphertext blocks are also repeated. This weakens the security of the encryption scheme as it is often the case that the message that is encrypted has some known regularity and an attacker can make use of this information to gain some information about the secret keys. Block ciphers can be strengthened by applying a bitwise XOR of the output ciphertext from the previous block to the current plaintext block, prior to the encryption process. This chaining together of the ciphertext prevents repeated blocks of plaintext from resulting in identical cipher-text blocks and is known as Cipher Block Chaining (CBC). Additionally the security of block ciphers can be increased by cascading a few of them together; the most common being triple-DES [14] which involves 3 stages of encryption with DES.

Although encryption guarantees privacy, it does not ensure message integrity. An adversary can alter the encrypted messages en route to the receiver. Since the receiver does not have a-priori knowledge of the message it cannot establish the integrity of the decrypted message. SSL ensures message integrity by sending a digest of the message to the receiver along with the original message. Digest algorithms,

such as MD5 [12] and SHA-1 [10], are one-way hash functions that output a unique digest for each input message. The input message can be quite large ($< 2^{64}$ bits) but the output digest is of a fixed size, viz. 128 bits for MD5 and 160 bits for SHA-1. It is relatively easy to verify a digest given the original message. However reproducing the message given the digest is impossible since the hash function is a many-to-one mapping. By incorporating a secret key into the hash algorithms it is possible to use message digests to guarantee the authenticity of a message and they are referred to as message authentication codes (MAC). SSL guarantees message integrity by keying the message digests with a secret key shared between the sender and the receiver. Any modification to the message will result in a mismatch between the digests computed by the sender and the receiver, thus enabling the receiver to detect a compromised message.

Conventional cryptography can be readily used to create a secure authenticated channel between a sender and a receiver provided there is some way to ensure that they both share a secret key. This however, is not an easy task, particulary in the context of the Internet, where there may be no prior interaction between the sender and the receiver. One of the breakthroughs of the mid 70's was the invention of public-key cryptography that allowed two parties to exchange secret information without requiring any a-priori shared secret. The first publication of a public-key cryptography algorithm was the Diffie-Hellman key exchange algorithm which appeared in 1976 [7, 8]. It described a simple protocol by which two parties with no a-priori shared secret, could exchange some information and derive a secret key out of this information. While the Diffie-Hellman algorithm was a major breakthrough in terms of cryptography it did not lend itself readily to e-commerce applications. The RSA public key cryptosystem [13], named after its 3 inventors Rivest, Shamir and Adelman, enabled digital signatures in addition to encryption. The RSA algorithms are a critical component of today's e-commerce transactions.

Unlike symmetric key cryptography, public key cryptography uses a pair of keys, a public key and a private key. As the name suggests, the owner of the key pair publishes the public component of the key and keeps the private component secret. If the public key is used to encrypt a message then only the private key can be used to decrypt it and vice versa. In SSL, the initiator of a session, typically the client, generates the secret and encrypts it with the public key of the peer, typically the server. The server, upon receipt of this message, uses its private key to decrypt it. Since the server is the only one who possesses the private key, from this point on, the client and the server share a secret that no one else knows. The encryption and decryption operations involve modular exponentiation to a very large base (e.g. 1024 bit number) which is computationally expensive. Hence in most applications, public key cryptography is used mainly to communicate a shared secret from which various keys can be derived. These keys are used to establish a secure channel that is protected by conventional cryptographic algorithms.

The key exchange problem is solved, provided the client knows the server's public key. While this can be supplied by the server, the client has to be able to make the connection between the public key and the true identity of the server. This can be achieved by having a trusted authority issue a digitally signed certificate that binds the server's public key to its fully qualified name, (or some other distinguising feature). The trusted authority is commonly referred to as a Certificate Authority (CA) and it is assumed that there is some authenticated, out-of-band means by which the CA's public key is distributed to all the clients. In the typical example of a web browser, the software is preloaded with the public keys of well-known CA's like Verisign, IBM World Registry, etc.

SSL makes use of X.509 certificates which are part of the ITU-T X.500 series of recommendations on directory services, to associate a public key with the real identity of an individual, server, or other entity. The X.509 certificate format includes several fields, the most important being the subject's name and its public key information. There is also a field indicating the name of the issuing CA as well as a period of

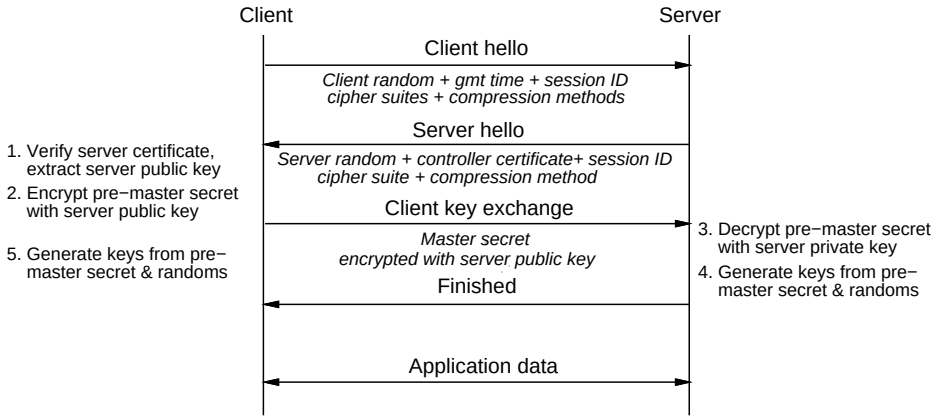


Figure 2: Message flow in SSL handshake protocol.

validity that restricts the lifetime of the certificate. Finally, the most important part of the certificate is the signature that covers all the other fields in the certificate. The signature process again involves public key cryptography and is most commonly the RSA algorithm. The signing entity (CA) computes a hash function of the data to be signed and encrypts that with its private key. The signature can be verified by performing the corresponding decryption operation with the public key of the CA and then matching the result with the freshly computed hash of the data.

3 SSL in a Nutshell

SSL is layered on top of an existing reliable transport protocol *viz.* TCP/IP. An SSL connection involves two stages. First, the communicating parties optionally authenticate each other and then exchange session keys. This phase is known as the SSL handshake. Once the handshake is completed, the two parties share a secret which can be used to construct a secure channel over which application data can be exchanged. SSL is intrinsically an asymmetric protocol. It differentiates between a client and a server. The SSL handshake sequence may vary, depending on whether the RSA or Diffie-Hellman key exchange is used. Client authentication is optional and is omitted in most cases. A typical SSL session makes use of the RSA key exchange algorithm with only the server being authenticated. This is by far the most common case and is the only key exchange algorithm considered in this article.

Figure 2 shows the message flow required to establish a new session. The client initiates the communication by sending a *Hello* message to the server. The Hello message includes a random number that is used in the handshake to prevent replay attacks. In response to the client Hello, the server replies with a Hello of its own. The server Hello message contains a Session ID field, which can be subsequently used by the client to identify a particular session with the server. The server Hello message is followed by an X.509 certificate that contains the server's public key. Optionally, the server may send a chain of certificates belonging to the parent authorities in the certification hierarchy. The client verifies the certificate (or chain of certificates) by verifying the identity of the server and checking the validity of the CA's signature. Once the client is assured that it has a valid certificate it extracts the server's public key from this certificate.

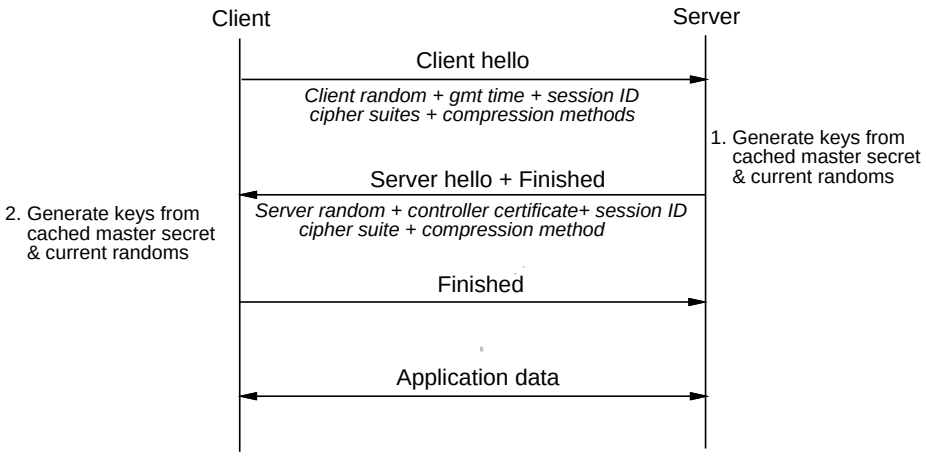


Figure 3: Message flow in SSL session reuse.

The client then generates a pre-master secret and encrypts it with the server's public key. This is sent to the server in a Client Key Exchange message. The server decrypts the Key Exchange message with its private key, thus obtaining the pre-master secret chosen by the client. Both the client and the server use a well-defined algorithm to generate a master secret from the pre-master secret as well as the client and server random numbers. The master secret is then used to generate symmetric keys for encryption and message authentication. More generally the master secret is a shared state between the client and server and this constitutes an SSL session. This session can be identified by a unique session ID that was chosen by the server and conveyed to the client in the initial Server Hello message. The session state is cached by both the server and the client for a limited amount of time.

In contrast to the initial handshake protocol, the re-establishment of an SSL connection using the cached session state is relatively simple. Figure 3 shows the messages exchanged to reestablish an SSL connection. As shown in the figure, the client simply specifies the session ID of the old or existing session it wishes to reuse when sending the Hello message. The server checks in its cache to determine if it has state associated with this session. If the session state still exists in the cache, it uses the stored master secret to create keys for the secure channel. The client repeats the same process and generates an identical set of keys. Note that multiple secure channels between the same pair of hosts can be established by reusing a single session state. In particular if a client wishes to open multiple secure channels to a server it only needs to go through the full handshake protocol for the first secure channel. All subsequent channels can be setup by using the cached SSL session state. This is a rather key feature of the SSL protocol that is particularly important in the context of the World Wide Web. A single secure web-page may be composed of multiple inline images that are obtained through separate HTTP connections. The ability to reuse an existing session state to setup multiple connections greatly reduces the overhead involved in dowloading complex web pages.

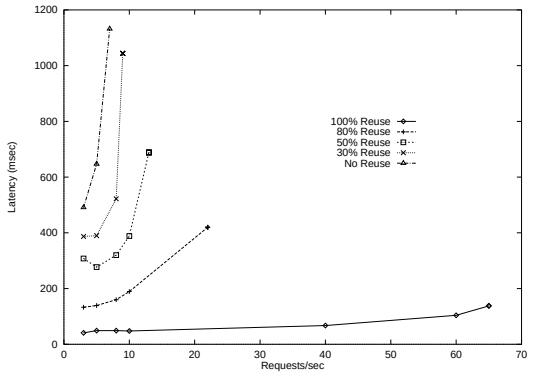


Figure 4: SPECweb96 Performance of Netscape Enterprise Sever.

4 SSL: A Performance Perspective

Although SSL can be used with a variety of application protocols, such as TELNET and FTP, the most important and common use of SSL has been to ensure privacy and authentication for HTTP transactions. Virtually all commercial web sites that require privacy and authentication use SSL. In this section, we benchmark the performance of secure web-servers and quantify the overheads of different components of SSL. We use the SPECweb96[15] benchmark as it attempts to capture real-world usage of a web-server and is based on the analysis of server logs from a few different Internet servers.

4.1 Experimental Setup

Our testbed consisted of a single IBM RS/6000 model 43P-200 running AIX 4.2, working as the server with multiple PCs working as clients. The server was equipped with a PowerPC 604e CPU running at 200 MHz with 32 KB on-chip 4-way associative instruction and data caches, a 512 KB direct mapped secondary cache, and 128 MB of RAM. The client machines were 266 MHz Pentium II PCs running Linux 2.0.35. A total of 8 client machines were directly attached to a Fast Ethernet Switch to which the server machine was also connected. This ensured that there were no bottlenecks due to network capacity during any of the experiments.

We have modified SPECweb96 to generate client workload for our secure web servers. The modified SPECweb clients make HTTP requests over SSL sessions. Since a typical web-access results in several different links being fetched from the same web-server, there is bound to be some reuse of SSL session state when setting up subsequent connections. The amount of reuse is heavily coupled with the way web-pages are setup, and we would like to investigate the server throughput with varying amounts of session reuse. Towards this end, we introduced a tunable knob that allows the SPECweb clients to control the degree of SSL session reuse.

For the experiments reported in this section, we did not modify the workload generated by SPECweb. The workload generated by SPECweb is designed to mimic the workload on regular Web servers. More specifically, the workload mix is built out of files in four classes: files less than 1KB account for 35% of all requests, files between 1KB and 10KB account for 50% of requests, 14% between 10KB and 100KB, and

finally 1% between 100KB and 1MB. There are 9 discrete sizes within each class (e.g. 1 KB, 2 KB, on up to 9KB, then 10 KB, 20 KB, through 90KB, etc.), resulting in a total of 36 different files (9 in each of 4 classes). Accesses within a class are not evenly distributed; they are allocated using a Poisson distribution centered around the midpoint within the class. The resulting access pattern mimics the behavior where some files (such as "index.html") are more popular than the rest, and some files (such as "mydog.gif") are rarely requested.

Although, the "real-life" workloads for standard and secure web servers are likely to be different, we chose to use the standard SPECweb workload for two reasons: (a) "real-life" workload for secure Web servers are not available at this time, and (b) our objective is to compare the performance of secure Web servers with that of non-secure servers and to analyze the performance impact of SSL. Using this modified SPECweb96 benchmark, we have evaluated two of the more popular web servers – Netscape Enterprise Sever 3.5.1, and Apache 1.2.4 with SSLeay 0.8.

4.2 Benchmark Results

Figures 4 and 5 show the latency versus the number of HTTPS (HTTP over SSL) requests handled by the Netscape and Apache server respectively. The servers are configured with certificates for 1024-bit keys. In all of these experiments, we used RC4 for data encryption and MD5 for message authentication, as these are the most widely used ciphers by secure web applications. Performance of other encryption and message authentication schemes are presented later in the section. We varied the degree of session reuse from 0-100%. When session reuse is 0% all SSL sessions setup between the server and the clients require a full handshake with the associated public and private key operations. When session reuse is 100%, only the first SSL session setup between the server and a client involves a full handshake. All subsequent connections reuse the already established session state between the server and the client. When the percentage of session reuse is in between 0 and 100, the clients reuse the same session for a certain number of times depending on the value of the reuse percentage. This is done by maintaining a running counter that keeps track of the number of connections that reused session state. Whenever this counter drops below the desired fraction (reuse percentage) of total connections, the client attempts to reuse an existing session ID. If the counter goes above the desired fraction of total connections, the client proceeds with the full handshake. For example, when the reuse percentage is set to 50, the sessions setup by a SPECweb client takes the form $NRNRNR\ldots$, where N stands for a new session and R stands for a reused session.

From Figure 5 it is evident that the Apache server can handle, at most, 13 requests per second when there is no session reuse. For the same case, Figure 4 indicates that the Netscape server can only handle about 7 requests per second. At these operating points the latencies are extremely high in both cases with Apache coming in at around 300 msec and Netscape hovering above the 600 msec mark. In both the figures we notice that as the amount of session reuse is increased the performance improves and with a 100% reuse the latency is fairly low even when the rate of connection requests is quite high. The measurements for 100% reuse are only provided as a reference since in all practicality a web-server is unlikely to experience such a large amount of session reuse. In comparison, the SPECweb96 measurements for Netscape and Apache for regular web-pages on the same server are around 300 and 250 requests a second, respectively.

The behavior of the Netscape server is fairly typical of what one would expect when the level of session reuse is varied. In Figure 4 we observe that the latency reduces and the sustainable throughput increases as the level of session reuse is increased. In contrast, the Apache server at light loads does not seem to exhibit any significant difference in the latency when the reuse is increased from 0 to 80%. This behavior may be

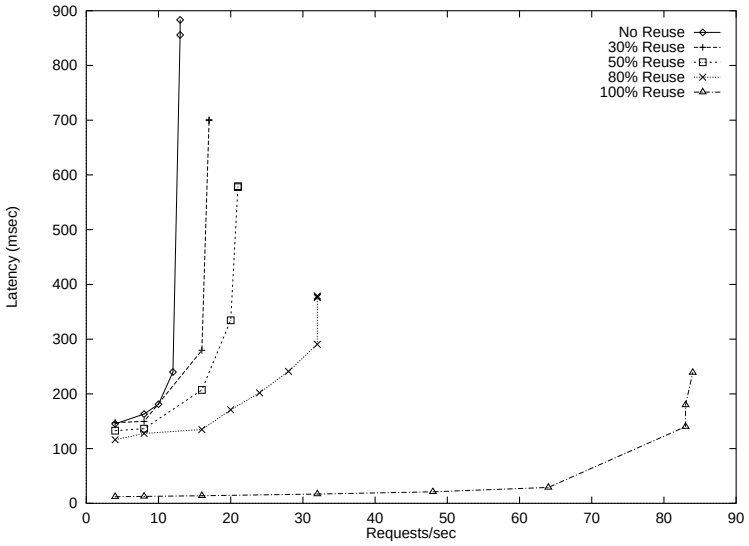


Figure 5: SPECWeb96 Performance of Apache with SSLeay.

a result of how session reuse is implemented in the Apache web server. Apache uses a process model in its web-server implementation. The web-server is composed of several, dynamically created server processes that serve web-requests. Rather than make a single entity responsible for dispatching the requests to each of the server processes, the creators of Apache chose to have each server process pick up a connection request and service it. This provides for some natural load-balancing features since a server process only picks up a request when it is free. When an SSL client wishes to reuse a session, it includes the session ID in the client Hello message. However at the time the connection is accepted by a server process, it has no knowledge of what the session ID will be since the Hello message is received only after the connection is accepted. Unfortunately, with most flavors of Unix, once a connection request is accepted there is no way to rescind it and so the server process is forced to serve the request, whether or not it has the session ID in its cache.

To get around this problem, the Apache server runs a separate process which acts as the global cache (`gcache`) server. Whenever a server process gets a session reuse request from a client, it first searches its own local session cache. If the local session cache does not have an entry for the client, the server process contacts the `gcache` server. If the `gcache` server has the specified entry in its database, it returns the cached state to the server process and session reuse is performed. Otherwise, a full handshake is performed and the session state is added to both the local cache and the global session cache. Since Apache spawns several server processes for the purpose of efficiency, at light loads it is quite likely that a newly arriving reuse request will be sent to a different server process (say process B) than the original process (say process A) with which the session state was established. As a result of this B needs to obtain the session state from the `gcache` server before setting up the new connection. Now B will not get a response from the `gcache` server until the `gcache` process is scheduled and subsequently B is scheduled to run again. This can take quite a while and so, for light loads, there is hardly any apparent reduction in latency even when there is session reuse. In fact, we ran a separate experiment where we reused the same session state over and over again and noticed that after about 16 requests (the maximum number of server processes was limited

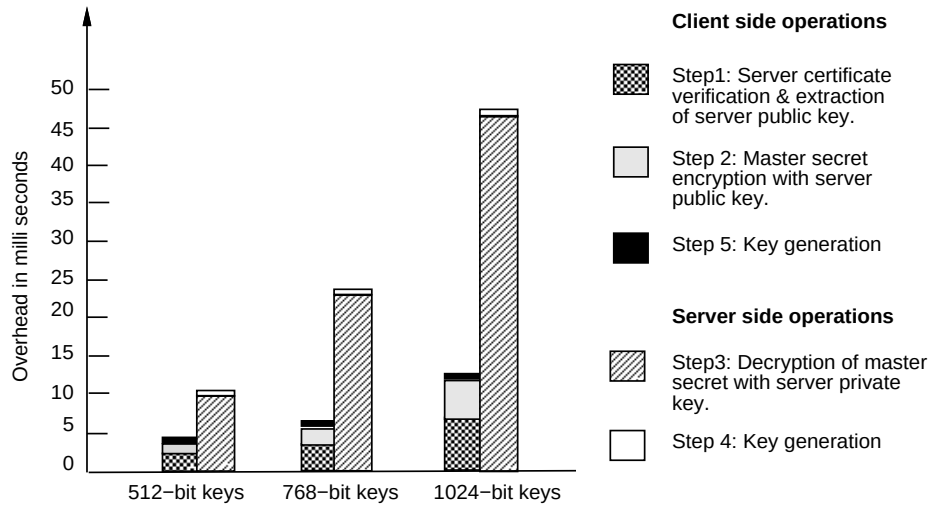


Figure 6: Client and Server Side Overheads in SSL Handshake Protocol.

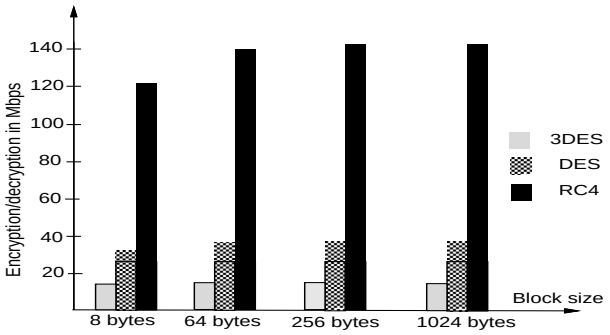
to 16) the latency to establish a secure connection dropped down significantly to little over 3 ms. This is because by this time all the server processes have a copy of the session state in their local cache and so do not need to go to the global cache to obtain the session state. This effect can also be seen in Figure 5 for the case where we have a 100 % reuse of session state. Since the same session is now being reused all the time, each of the server processes has the session state in its local cache and so the latency is really low (10-15 ms) even at fairly high rates (60-70/sec) of connection requests.

4.3 Overhead Analysis

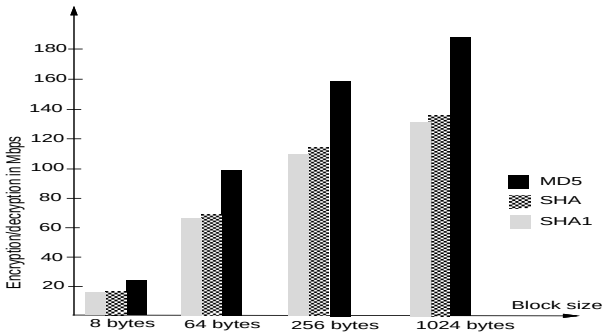
In the last section, we quantified the performance of secure Web servers and compared it with that of standard non-secure servers under the same workload, namely SEPCweb96. Our results show that the performance penalty for security is rather large. In this section, we take a closer look at the performance overhead associated with different components of SSL. For this purpose, we have instrumented the SSL protocol stack in SSLeay for a detailed profiling of various processing modules in the data path. The instrumented stack can be used to capture a sequential flow of time-stamped events on the data path. The time-stamps are of sub-microsecond granularity and are taken by reading a real-time clock which is an integral part of the PowerPC CPUs used in RS/6000s. We use a two-instruction assembly language routine to read two 32-bit clock registers with minimal overhead. In the following, we present a detailed analysis of the overhead associated with the SSL handshake protocol, and the performance impact of encryption and authentication during data transfer.

Session Setup Overhead

The SSL session setup overhead can be divided up into (1) an increase in data volume due to additional data items, such as server certificates, and (2) computational overhead for crypto functions. Data items exchanged during SSL handshake increase the latency of HTTP transactions. When the server uses a self-



(a) Encryption/decryption performance



(b) Message digest generation/verification performance

Figure 7: Crypto overhead in data transfer.

signed[1] certificate the amount of data sent by the server to the client during handshake is about 750 bytes. When the server sends a chain of certificates to the client, each certificate adds about 750 bytes to the data sent by the server. The amount of data sent by the client is about 250 bytes. The relative overhead due to increase in data volume depends on the network connectivity. For clients connected via dialup lines, the increase in latency due to increase in data volume may be significant. For clients connected via LANs, the overhead due to increase in data volume pales in comparison with the computational overhead incurred by the crypto functions.

Figure 6 shows the overheads involved in setting up a SSL session. There are 3 sets of measurements based on the size of the server's public key, i.e. 512, 768 or 1024-bit. As seen in Figure 6, the most expensive component in session setup is the private key operation at the server side. Verification of the server certificate(s) and generation and encryption of the master secret are the major operations performed on the client side. Ironically, the most expensive of the crypto operations is performed at the server, which significantly reduces the number of connections it can support. In [4], we propose modifications to SSL handshake protocol that siginificantly reduces the server side overhead. Note that both server and client side operations are more expensive when the server uses longer private keys. For US domestic use 1024-bit server keys are recommended and used.

Reusing existing session state can greatly reduce the cost of connection setup. Since there are no public key operations involved, when the session state is being reused, the time taken to setup a secure channel is about 3 to 4 msec which is one order of magnitude less than the SSL handshake overhead.

Data Transfer Overhead

Figure 7 shows the performance of crypto functions used in the data path to encrypt/decrypt message and to generate/verify message digests. Most web browsers are by default configured to use RC4 for data encryption. When higher level of security is required, DES is preferred. For the highest level of security, 3DES [16] is the recommended encryption algorithm. Figure 7(a) shows the performance of RC4, DES, and 3DES for different data block sizes. When RC4 is used as the encryption/decryption algorithm, the

[1] A self-signed certificate is one that is signed with the private counterpart of the public key that is contained in the certificate

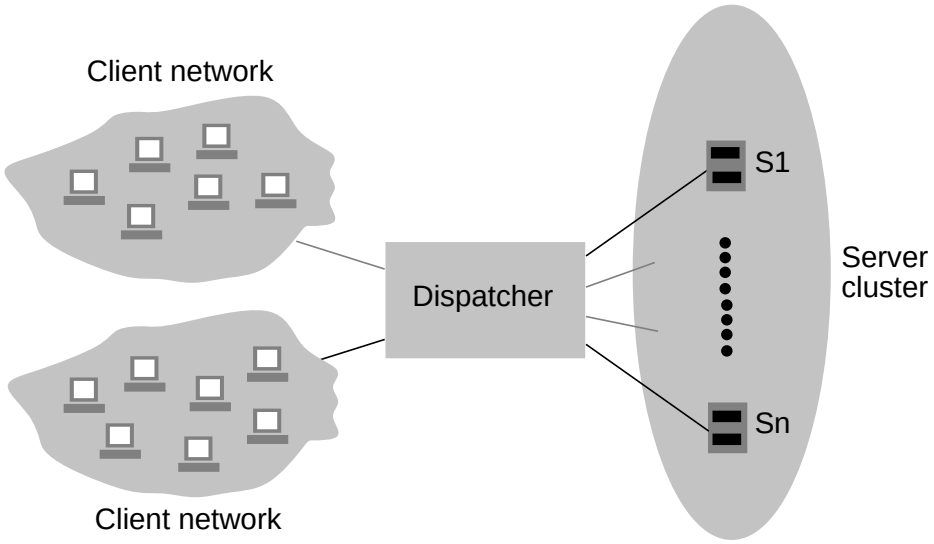


Figure 8: Typical setup of a large e-commerce site.

server can encrypt/decrypt at the rate of 120Mbps. The encryption rate for DES is between 20-40Mbps and is based on the block size. With 3DES the encryption/decryption rate goes down to about 10-15Mbps. Note that in the results reported in Figures 4 and 5 we used RC4 for data encryption. Figure 7(b) shows the performance of the message digest generation and verification algorithms. By default all web browsers use MD5 as the message digest algorithm. Browsers can also be configured to use SHA and SHA1 which are considered to be more secure. As the figure shows, MD5 can generate/verify message digests at a rate of 20Mbps for 8 byte messages and at more than 180Mbps for messages of size 1024 bytes and more. SHA and SHA1 achieve comparable performance of about 20Mbps for small (8 byte) messages and upto 120Mbps for larger messages (1024 bytes and more). We should note here that typical web transfers are about 4 Kbytes or more. In other words, the overhead of encrypting and message digest generation of a 4K byte message on the server are 0.25 ms and 0.20 ms, respectively, compared to 45 ms for SSL handshake. The results in Figures 7(a) and 7(b) show that it is neither the encryption nor the computation of message digests is the real bottleneck for SSL.

5 Scaling an E-commerce Site

The benchmark results presented in the last section clearly demonstrate that the SSL protocol overhead has a profound impact on the performance of e-commerce servers. A significant part of the overhead is contributed by the SSL handshake protocol. An effective way to eliminate much of that overhead is to aggressively reuse SSL session state whenever possible. In this section, we focus on techniques to improve SSL session reuse in large e-commerce installations which use clusters of servers to handle millions of transactions every day. In a typical e-commerce installation the server cluster is front-ended by a dispatcher (Figure 8) responsible for distributing connections across the nodes in the cluster [6]. The dispatcher is often unaware of SSL session level information. As a result, a connection that can potentially reuse the

SSL session state on a server node is often routed to a different node leading to poor SSL session reuse efficiency.

To understand the problem better, we need to know how dispatching in a server cluster works. In a cluster environment, all cluster nodes share a common virtual IP address, and are known to the external world through this address. Additionally, each node in the cluster also has its own unique address which is used to route traffic to specific nodes inside the cluster. Client requests are addressed to the cluster virtual address and are intercepted by the dispatcher. When the first packet of a new connection [2] arrives at the dispatcher, it decides which server node the connection should be routed to based on server loads and other policy rules. The packet is then forwarded to the appropriate server node using its unique address. The subsequent packets belonging to the same connection are routed to the same server node. Since the dispatching mechanism does not take into account any SSL session level information, SSL connections that can potentially reuse the same session state may be routed to different server nodes. This defeats the reuse of session state which is clearly detrimental to server performance.

5.1 Improving Session Reuse

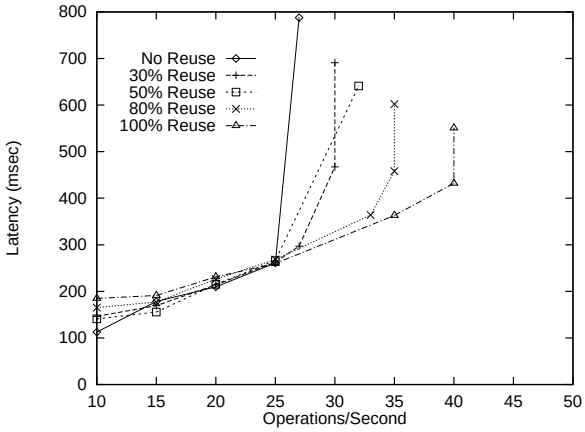
To improve SSL session reuse efficiency, some dispatchers are configured to route all connections originating from the same client to the same server node. This can be easily achieved by looking at the source IP addresses of the incoming requests. Unfortunately, this simple approach does not work very well in practice. Many clients reside behind corporate firewalls and ISP (Internet Service Provider) proxies. Connections originating from a client behind a firewall (or a proxy) bears the address of the firewall (or the proxy) as the source address. As a result, a dispatcher configured to route all connections originating from the same client to a single server node, routes connections originating form all clients behind a firewall (or a proxy) to the same server node, leading to massive load imbalance. Since a large percentage of Internet clients are behind firewalls and proxies, this poses a serious problem with no obvious solutions.

An alternative approach to improve SSL session reuse efficiency in a cluster environment is to share the session cache among all cluster nodes. While sharing of session cache is feasible, there are many technical obstacles that makes it difficult. First, for security reasons, it is not advisable to make the session cache accessible over the network. Even if one disregards the security advisory, at a minimum one has to make sure that both the session caches and their clients authenticate each other appropriately. Creating such an infrastructure requires a complex configuration and is an administrative nightmare. Second, this approach requires modifications to the SSL libraries and standardization of session cache interfaces so that different implementations of SSL can share the session state information with each other.

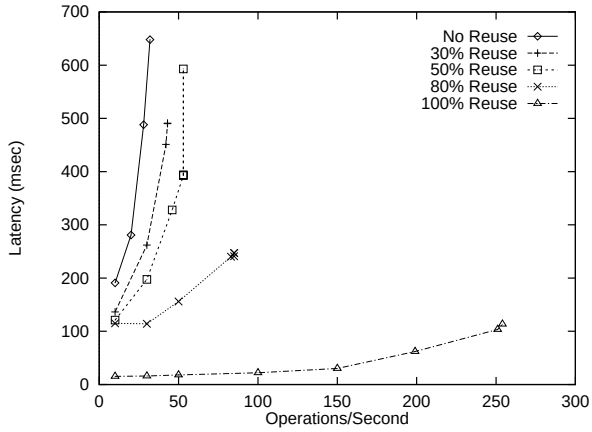
An elegant and a better alternative is to use an SSL session aware dispatcher. Such a dispatcher can learn the SSL session to cluster node mappings by snooping on SSL messages and can dispatch the session reuse requests to the appropriate server nodes using this mapping. In the following, we briefly describe the working principles of a SSL session aware dispatcher that we are developing [3].

Recall that a client initiates an SSL session by sending a client Hello message to the server. The client Hello message includes a session ID field which is set to zero when a new session is to be initiated. The server chooses the session ID and communicates that to the client in the server Hello message. When the client wants to reuse a specific session state for another connection, it sets the session ID field in the client Hello message to the session ID of that session. An SSL session aware dispatcher works like an application layer router. It intercepts the Hello messages from the client as well as the server. By snooping into the

[2] A connection is identified by source and destination IP addresses and TCP port numbers.



(a) 3-node Apache cluster with SSL unaware dispatching.



(b) 3-node Apache cluster with SSL aware dispatching.

Figure 9: Impact of session session aware dispatching on server performance.

server Hello messages, the dispatcher learns the session IDs chosen by the server nodes and creates a sever node to session ID mapping. It uses the session ID contained in the client Hello message to route the connection to the appropriate server node. If the session ID field in the client Hello message is set to zero, that means a new session has to be established. Server affinity does not dictate the connection routing decision in this case. Instead, load balancing among the cluster nodes is used as the guiding criterion. If the session ID is non-zero, the dispatcher uses the mapping between the session IDs and the server nodes to route the connection to the node that contains the session state for that session.

Session ID to server node mappings are timed out after a configurable timeout period. If the timeout value is chosen to be the same as the server's session cache timeout, it is possible to achieve near perfect reuse efficiency. If the timeout value used by the dispatcher is larger than that used by the server nodes, a reuse request may be mis-routed to a server node which no longer has the session state in its cache. On the other hand, if the dispatcher uses a smaller timeout value than the server, it may not have the session ID to server node mapping when a reuse request arrives at the system. In either case, a new session has to be established between the client and the server node. The dispatcher learns the session ID for this new session by snooping on the server Hello message. All subsequent reuse requests are routed correctly.

5.2 Experimental Results

Figure 9 shows the impact of session aware dispatching on Apache 1.2.4 with SSLeay 0.8. For this set of experiments, we used three identical servers similar to the one used for experiments in Section 4. The load was generated using a PC cluster running SPECweb96 suitably modified to generate HTTPS traffic. Figure 9(a) shows the performance of the server cluster when the load balancer is unaware of SSL session level information and dispatches connections based on layer 4 information only. Figure 9(b) shows the performance of the server cluster when the connections are routed to maximize session reuse. In both cases, we varied the degree of session reuse from 0-100%.

As Figure 9(a) shows, when SSL sessions are blindly dispatched to nodes in the cluster, the aggregate

throughput of the cluster saturates at around 30-35 connections per second depending on the degree of session reuse. As expected, the degree of session reuse has little impact on performance. There is however an interesting anomaly that can be observed at low utilizations where the latency increases with the degree of session reuse. This is due to the fact that Apache maintains a global cache to store all SSL session state in addition to the per-process cache maintained by the server processes. While processing a reuse request, the server process first checks its local cache [3] for a hit. If it fails to find a match in its local cache it searches the global cache for a hit. As the degree of session reuse increases, so does this futile search through the global cache. This results in an increased latency for the connections that request a reuse of session state. When the utilization level is sufficiently high a significant amount of time is spent waiting for the CPU and so this effect is masked at higher loads. Figure 9(b) demonstrates how SSL session aware dispatching can substantially improve the performance of the server cluster. In this case as the degree of reuse increases so does the throughput of the server cluster. With 80% session reuse the three server cluster can sustain a throughput of about 100 connections per second, almost triple the throughput achieved in the previous experiment at the same level of reuse. With 100% reuse we observe a six fold improvement in the performance.

6 Summary

SSL is the de facto standard for security in e-commerce applications. Although the security implications of SSL have been under the microscope ever since its inception, similar analysis of its performance has not been performed. In this article we presented experimental evidence that demonstrate that SSL inflicts significant performance overhead on e-commerce applications. In light of this observation, we outlined a strategy to alleviate these overheads in large e-commerce installations.

References

[1] C. Allen and T. Dierks. The TLS Protocol Version 1.0. Internet Draft, Internet Engineering Task Force, November 1997. Work in progress.

[2] ANSI X3.106. American National Standard for Information Systems-Data Link Encryption. American National Standards Institute, 1983.

[3] G. Apostolopoulos, V. Peris, P. Pradhan, and D. Saha. L5: A Self-learning Layer 5 Switch. IBM Research Technical Report RC 21461, April 1999. IBM T.J. Watson Research Center.

[4] G. Apostolopoulos, V. Peris, P. Pradhan, and D. Saha. Transport Layer Security: How Much Does It Really Cost? In *Proceedings of the IEEE INFOCOM*, March 1999.

[5] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol – HTTP/1.0, October 1995.

[6] Cisco Local Director. Technical White Paper, 1998. Cisco Systems.

[7] W. Diffie and M. E. Hellman. Multiuser cryptographic techniques. In *Proceedings of the AFIPS National Computer Conference*, June 1976.

[3] Note that `gcache` in Section 4 is a local cache as far as the cluster node is concerned.

[8] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6):74–84, June 1977.

[9] A. Frier, P. Karlton, and P. Kocher. The SSL 3.0 Protocol. Netscape Communications Corporation, November 1996.

[10] NIST FIPS PUB 180-1. Secure Hash Standard. National Institute of Standards and Technology, U.S. Department of Commerce, May 1994. DRAFT.

[11] B. Ramsdell. S/MIME Version 3 Message Specification. Internet Draft, Internet Engineering Task Force, May 1998. Work in progress.

[12] R. Rivest. RFC 1321: The MD5 Message Digest Algorithm, April 1992.

[13] R. Rivest, A. Shamir, and L. M. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[14] B. Schneier. *Applied Cryptography.* Wiley, New York, 1996.

[15] The Standard Performance Evaluation Corporation. Specweb96, 1996. http://www.spec.org/osg/web96.

[16] W. Tuchman. Hellman Presents No Shortcut Solutions To DES. *IEEE Spectrum*, 6(7):40–41, July 1979.

[17] Visa International and MasterCard International. Secure Electronic Transaction 1.0 specification, December 1997. http://www.setco.org.

[18] T. Ylonen, T. Kivinen, and M. Saarinen. SSH Protocol Architecture. Internet Draft, Internet Engineering Task Force, November 1997. Work in progress.