Performance Analysis of TLS Web Servers

CRISTIAN COARFA, PETER DRUSCHEL, and DAN S. WALLACH
Rice University

TLS is the protocol of choice for securing today's e-commerce and online transactions but adding TLS to a Web server imposes a significant overhead relative to an insecure Web server on the same platform. We perform a comprehensive study of the performance costs of TLS. Our methodology is to profile TLS Web servers with trace-driven workloads, replace individual components inside TLS with no-ops, and measure the observed increase in server throughput. We estimate the relative costs of each TLS processing stage, identifying the areas for which future optimizations would be worthwhile. Our results show that while the RSA operations represent the largest performance cost in TLS Web servers, they do not solely account for TLS overhead. RSA accelerators are effective for e-commerce site workloads since they experience low TLS session reuse. Accelerators appear to be less effective for sites where all the requests are handled by a TLS server because they have a higher session reuse rate. In this case, investing in a faster CPU might provide a greater boost in performance. Our experiments show that having a second CPU is at least as useful as an RSA accelerator. Our results seem to suggest that, as CPUs become faster, the cryptographic costs of TLS will become dwarfed by the CPU costs of the nonsecurity aspects of a Web server. Optimizations aimed at general purpose Web servers should continue to be a focus of research and would benefit secure Web servers as well.

Categories and Subject Descriptors: C.2.2 [**Computer-Communication Networks**]: Network Protocols—*Protocol architectures*; C.2.4 [**Computer-Communication Network**]: Distributed Systems—*Client/server*; C.4.0 [**Performance of Systems**]: General—*Measurement techniques, performance attributes*; D.2.8 [**Software Engineering**]: Metrics—*Performance measures, product metrics*; D.4.6 [**Operating Systems**]: Security and Protection—*Access controls, authentication*; D.4.8 [**Operating Systems**]: Performance—*Measurements, modeling and prediction*

General Terms: Measurement, Performance, Security

Additional Key Words and Phrases: TLS, Internet, e-commerce, RSA accelerator, secure Web servers

1. INTRODUCTION

Secure communication is an intrinsic demand in today's world of online transactions. Two methods have emerged as Internet standards: IPsec (IP security)

We gratefully acknowledge HP/Compaq for loaning us the hardware used in performing our experiments.
An earlier version of this paper was appeared in, Proceedings of the 2002 Network and Distributed System Security Symposium, San Diego, CA, Feb. 2002.
Author's address: Department of Computer Science, Rice University, 6100 Main St., MS 132, Houston, TX 77005-1892; email: ccristi@cs.rice.edu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or permissions@acm.org.
© 2006 ACM 0734-2071/06/0200-0039 $5.00

and SSL/TLS [Dierks and Allen 1999]. IPsec is largely transparent to networking applications, running at the network layer, and is most commonly deployed for virtual private networks.

The most widely used method is SSL/TLS. TLS, initially called SSL (Secure Socket Layer), was originally designed at Netscape for its Web browsers and servers [Freier et al. 1996]. The SSL protocol was designed with four target objectives: cryptographic security, interoperability, extensibility, and relative efficiency. SSL had to provide cryptographic security (secure connections between parties). SSL is an asymmetric protocol; the parties involved are classified as clients or servers. SSL supports both client and server authentication as well as secret key agreement by using public key methods. In typical use for Web servers, the server is authenticated to the client but the client is not authenticated in return. SSL protects data confidentiality by using symmetric-key ciphers to encrypt messages exchanged. SSL ensures data integrity by computing a message authentication code (MAC) of the message and appending it to the message; the receiving party recomputes the MAC and checks if the received message matches it. The second goal of SSL was interoperability: independent programmers should be able to develop applications utilizing SSL that will then be able to successfully exchange cryptographic parameters without knowledge of one another's code. By using a simple API to separate SSL from an application, SSL can be easily used within new applications, freeing application developers from worrying about many of the complexities of correctly implementing cryptographic systems. The third goal of SSL was extensibility. SSL provides a framework into which new authentication and encryption methods can be incorporated. If, for example, a new symmetric-key cipher is invented that is faster and more secure than existing ciphers, SSL can support it alongside its existing ciphers. This allows new cryptographic technologies to be introduced into commercial use without requiring complete reimplementation of the necessary support infrastructure. Finally, SSL's fourth goal was to operate efficiently. Cryptographic operations tend to be highly CPU-intensive, particularly public key operations. The SSL protocol minimizes the need for performing public key operations by incorporating an optional session caching scheme, allowing earlier public key results to be safely reused for new connections.

SSL runs at the transport layer above existing protocols like TCP. It offers an abstraction of secure sockets on top of existing TCP/IP sockets as shown in Figure 1. Applications only have to replace regular *read*/*write* calls over the TCP/IP sockets with *SSL_read* /*SSL_write* over the corresponding secure sockets. This simplicity made SSL attractive to application developers.

The SSL 1.0 protocol was designed and released in 1994. Version 2.0 was released in the same year; it fixed some bugs present in version 1.0 and clarified some aspects of the protocol. SSL version 2.0 was found to be vulnerable to several attacks [Wagner and Schneier 1996]. For example, the ciphersuite rollback attack allows an active attacker (one who can intercept and modify messages in transit between the client and server) to force an SSL session to use weaker ciphersuites than would ordinarily be negotiated. Another vulnerability allowed an attacker to compromise a message's integrity without detection.

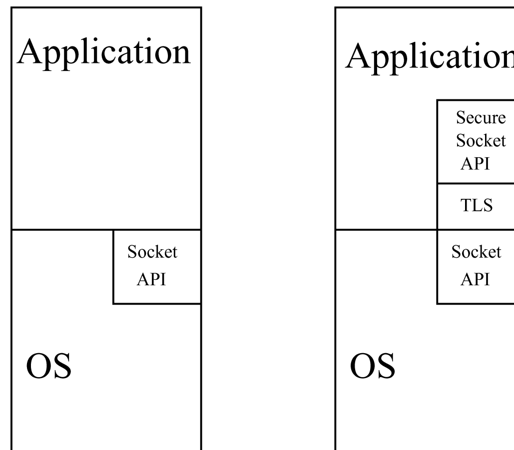


Fig. 1. TLS offers an abstraction of secure sockets on top of TCP/IP sockets.

Version 3.0 of the SSL protocol was released at the end of 1995 and fixed the flaws discovered in version 2.0. SSL 3.0 addressed these earlier attacks, and was also designed to prevent version rollback attacks, where attackers might try to trick SSL 3.0 servers into using the earlier, faulty SSL version 2.0 protocol. SSL has been standardized by the IETF and is now called Transport Layer Security (TLS). The security offered by TLS has been the subject of comprehensive analysis [Bradley and Davies 1995; Wagner and Schneier 1996; Mitchell 1998; Paulson 1999; Halevi and Krawczyk 1999; Buhler et al. 2000; Dean and Stubblefield 2001; Hess et al. 2002]. TLS is both widely used and widely analyzed. If and when flaws are found, they are generally addressed quickly by the community.

To provide secure connections for the rapidly growing field of wireless communication, a secure connection protocol, WTLS [Wireless Application Protocol Forum 2001; Shacham and Boneh 2002], based on TLS, has been proposed. Modifications were made to the TLS protocol in order to accommodate the low bandwidth communication links, limited processing power paper, and limited memory capacity characteristic of wireless platforms. Our focus in this article is on TLS, not on WTLS.

TLS is used in a variety of applications, including secure Web servers, secure shell connections, and secure mail servers. The goal of this research is to provide a comprehensive performance analysis of TLS Web servers since TLS is most commonly used for secure Web applications such as online banking and e-commerce.

TLS is computationally expensive relative to the cost of serving a Web page. Unsurprisingly, most sites delivering general public information, such as home pages, free news sites, search engines, and others, choose to operate without TLS. However, e-commerce sites must use secure communication for the obvious reason: leaks of information might eventually lead to the loss of money. The same applies to any site that handles sensitive information such as email and medical records. These sites must address the TLS performance problem.

In its most common usage on Web sites, TLS uses 1024-bit RSA for authentication and key exchange [Rivest et al. 1978]. Data privacy is protected using the RC4 stream cipher with 128-bit keys, and message integrity is verified using the MD5 hash algorithm with a secret key as a message authentication code (MAC). Details of these algorithms can be found in Schneier [1996] and most other introductory cryptography texts.

TLS Web servers incur a significant performance penalty relative to regular Web servers running on the same platform. Our experiments showed the performance penalty to range from as little as a factor of 3.4 to as much as a factor of 9. It is generally known that RSA operations are particularly expensive. As a result of this cost, a number of hardware accelerators are offered by vendors such as nCipher, Broadcom, and HP/Compaq's Atalla division. These accelerators take the modular exponentiation operations of RSA and perform them in custom hardware, thus freeing the CPU for other tasks. Researchers have also studied algorithms and systems to accelerate RSA operations. Boneh and Shacham [2001] have designed a software system to perform RSA operations together in batches at a lower cost than doing the operations individually. Dean et al. [2001] have designed a network service, offloading the RSA computations from Web servers to dedicated servers with RSA hardware. Vendors such as Alteon [2002] and Intel [2002] offer front-end switches that offload TLS protocol processing and perform Web server load balancing.

The TLS designers knew that RSA was expensive and that Web browsers tend to reconnect many times to the same Web server. To address this, they added a TLS session cache, allowing subsequent connections to resume an earlier TLS session and thus reuse the result of an earlier RSA computation. Research has suggested that, indeed, session caching helps Web server performance [Goldberg et al. 1998].

While previous attempts to understand TLS performance have focused on specific processing stages, such as the RSA operations or the session cache, we analyze TLS Web servers as *systems* and measure page-serving throughput under trace-driven workloads. Our methodology is to replace each individual operation within TLS with a no-op and measure the incremental improvement in server throughput. This methodology measures the upper-bound that may be achieved by optimizing each operation within TLS, through either hardware or software acceleration techniques. We can measure the upper-bound on a wide variety of possible optimizations, including radical changes like reducing the number of TLS protocol messages. Creating such an optimized protocol and proving it to be secure would be a significant effort, whereas our emulations let us rapidly measure an upper bound on the achievable performance benefit. If the benefit were minimal, we would then see no need for designing such a protocol.

Our results show that, while RSA operations are expensive, they don't account for the whole cost of TLS. RSA accelerators are effective, but a second CPU is at least as effective. Our results seem to suggest that faster CPUs will bridge the performance gap between secure and insecure Web servers. Optimizations aimed at general purpose Web servers should continue to be a focus of research and would benefit secure Web servers as well.

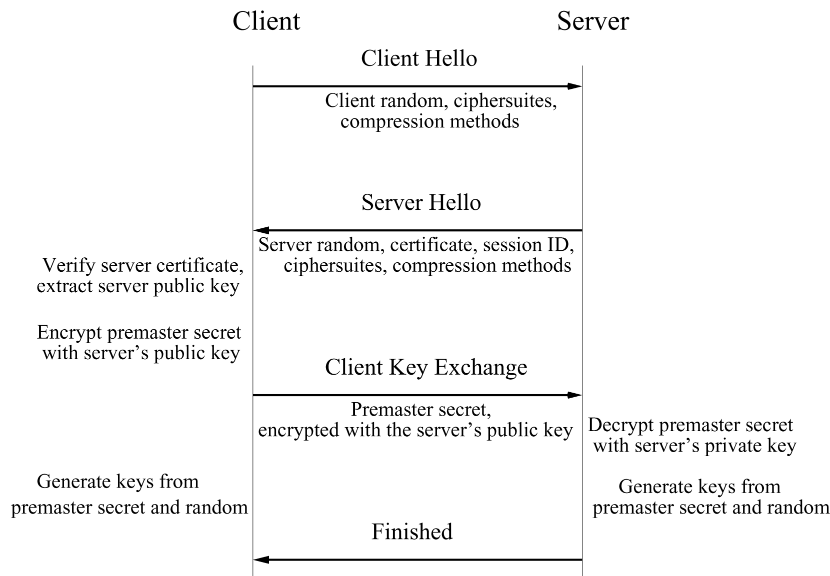


Fig. 2. New TLS connection handshake.

Section 2 presents an overview of the TLS protocol and explains how we performed our experiments and what we measured. Section 3 analyzes our measurements in detail. Section 4 describes future work. We present related work in Section 5. Section 6 concludes with a high-level view of our results.

2. TLS PROTOCOL OVERVIEW

The TLS protocol, which encompasses everything from authentication and key management to encryption and integrity checking, fundamentally has two phases of operation: connection setup and steady-state communication.

Connection setup is quite complex. Readers looking for complete details are encouraged to read the RFC [Dierks and Allen 1999]. The setup protocol must, among other things, be strong against active attackers trying to corrupt the initial negotiation where the two sides agree on key material. Likewise, it must prevent replay attacks where an adversary who recorded a previous communication (perhaps one indicating some money is to be transferred) could play it back without the server realizing that the transaction is no longer fresh (and thus, allowing the attacker to empty out the victim's bank account).

TLS connection setup has the following steps (quoting from the RFC), as shown in Figure 2:

- —Exchange hello messages to agree on algorithms, exchange random values (referred to as client random and server random), and check for session resumption.
- —Exchange certificates and cryptographic information to allow the client and server to authenticate themselves. [In our experiments, we do not use client certificates.]

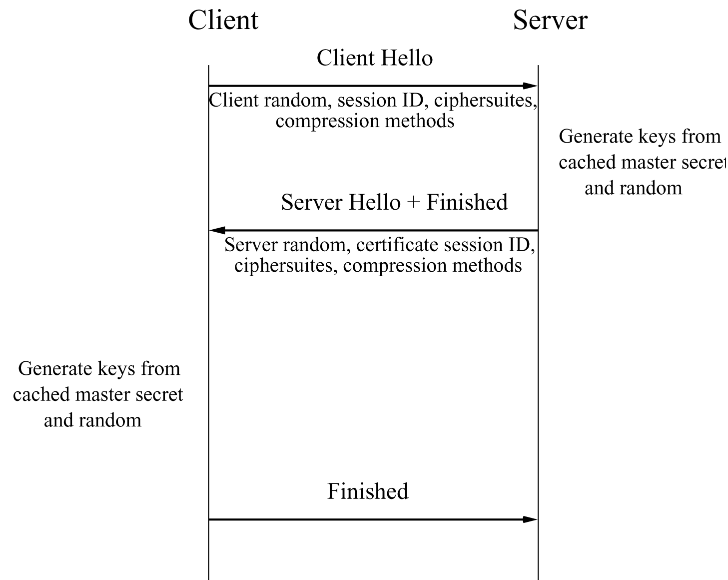


Fig. 3. Resumed TLS connection handshake.

—Exchange the necessary cryptographic parameters to allow the client and server to agree on a premaster secret.
—Generate a master secret from the premaster secret chosen by the client and the exchanged random values.
—Allow the client and server to verify that their peer has calculated the same security parameters and that the handshake occurred without tampering by an attacker.

There are several important points here. First, the TLS protocol designers were aware that performing the full setup protocol is quite expensive, requiring two network round-trips (four messages) as well as expensive cryptographic operations such as the 1024-bit modular exponentiation required of RSA. For this reason, the master secret can be stored by both sides in a session cache. When a client subsequently reconnects, it need only present a session identifier. Then, the master secret (known to the client and server but not to any eavesdropper) can be used to create the connection's encryption keys, message authentication keys, and initialization vectors. The sequence of messages exchanged is shown in Figure 3.

After the setup protocol is completed, the data exchange phase begins. Prior to transmission, the data is broken into packets of 16KB. Each packet is optionally compressed. A MAC is computed and added to the message with its sequence number. Finally, the packet is encrypted and transmitted. TLS also allows for a number of control messages to be transmitted. These processing steps are presented in Figure 4. The receiver follows the same steps in reverse order. The MAC is computed by the receiver and compared with the received MAC in order to ensure the message integrity.

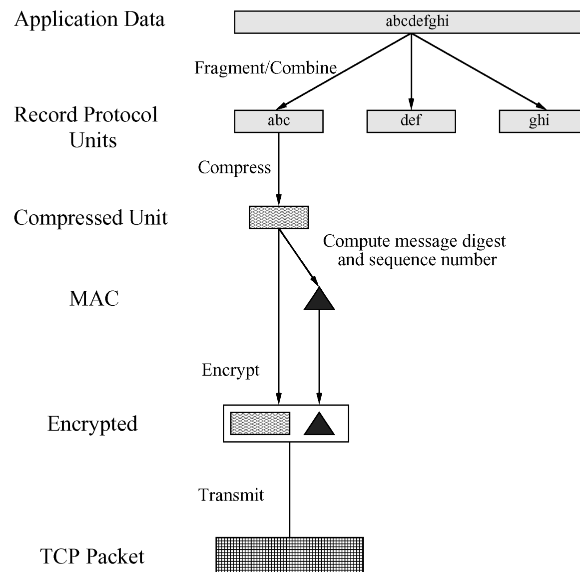


Fig. 4. TLS processing steps during data exchange.

Analyzing this information, we see a number of operations that may form potential performance bottlenecks. Performance can be affected by the CPU costs of the RSA operations and the effectiveness of the session cache. It can also be affected by the network latency of transmitting the extra connection setup messages as well as the CPU latency of marshaling, encrypting, decrypting, unmarshaling, and verifying packets. This article aims to quantify these costs.

We chose not to perform micro-benchmarks such as measuring the necessary CPU time to perform specific operations. In a system as complex as a Web server, I/O and computation overlap in complex ways, and the system's bottlenecks are never intuitively obvious. Instead, we chose to measure the *throughput* of a secure Web server under various conditions. To measure the costs of individual operations, we replaced them with no-ops. Replacing cryptographically significant operations with no-ops is obviously insecure, but it allows us to measure an upper bound on the performance improvement that would result from optimizing particular components of the system. In effect, we emulate ideal hardware accelerators. Based on these numbers, we can estimate the relative cost of each operation using Amdahl's Law (see Section 3). A similar technique of nulling out calls to hardware was proposed by Brendan et al. [1993].

2.1 Platform

Our experiments used two different hardware platforms for the TLS Web servers: a generic 500MHz Pentium III clone and a Compaq DL360 server with dual 933MHz Pentium III CPUs. The Compaq DL360 server was used in both single and dual CPU configurations. Both machines had 1GB of RAM and a gigabit Ethernet interface. Some experiments also included a Compaq AXL300

cryptography acceleration board with a throughput of 330 RSA operations per second. Three generic 800MHz Athlon PCs with gigabit Ethernet cards served as TLS Web clients, and all experiments were performed on a private gigabit Ethernet.

Every computer ran RedHat Linux 7.2. The standard Web servers used were Apache 1.3.14, and the TLS Web server was Apache with mod_SSL 2.7.1-1.3.14. We have chosen the Apache mod_SSL solution due to its wide availability and use as shown by a March 2001 survey [NetCraft 2001]. The TLS implementation used in our experiments by mod_SSL is the open source OpenSSL 0.9.5a. The HTTPS traffic load was generated using the methodology of Banga and Druschel [1999] which we modified to support TLS. As we are interested primarily in studying the CPU performance bottlenecks arising from the use of cryptographic protocols, we needed to guarantee that other potential bottlenecks, such as disk or network throughput, did not cloud our throughput measurements. To address this, we used significantly more RAM in each computer than its working set size, thus minimizing disk I/O when the disk caches are warm. Likewise, to avoid network contention, we used Gigabit Ethernet which provides more bandwidth than the computers in our study can saturate.

We didn't investigate 4-way or 8-way CPU, limiting ourselves to single and dual CPU configurations. However, these platforms are relevant because dual CPU servers are common in dense server farms.

2.2 Experiments Performed

We performed six sets of experiments, using two different workload traces against three different machine configurations.

One workload in our experiments simulated the traffic pattern of secure servers at Amazon.com. Normally, an Amazon customer selects goods to be purchased via a normal Web server and only interacts with a secure Web server when submitting credit card information and verifying purchase details. Our other workload was a 100,000-hit trace taken from our departmental Web server with a total document corpus size of 530MB. While our departmental Web server supports only normal, unencrypted Web service, we measured the throughput for running this trace under TLS to determine the costs that would be incurred if our normal Web server was replaced with a TLS Web server.

These two workloads represent extremes of the workload spectrum that TLS-secured Web servers might experience. The Amazon workload has a small average file size, 7KB, while the CS trace has a large average file size, 46KB. Likewise, the working set size of the CS trace is 530MB, while the Amazon trace's working set size is only 279KB. Even with the data cached in RAM, these two configurations provide quite different stresses upon the system. We expected the CS trace to exhibit a high relative cost for the TLS data exchange—including bulk ciphers and message digest functions—because it contained larger files (the average file size was 46KB). The Amazon trace, on the other hand, had smaller files (the average file size was 7KB); consequently, we

expected the relative cost of the connection setup, including RSA decryption, to be high.

In addition to replacing cryptographic operations such as RSA, RC4, MD5/SHA-1, and secure pseudorandom number generation with no-ops[1], we also investigated replacing the session cache with an idealized perfect cache that returns the same session every time (thus avoiding contention costs in the shared memory cache). Simplifying further, we created a skeleton TLS protocol where all TLS operations have been completely removed; during the skeleton TLS handshake, the two parties exchange messages of the same length as the corresponding messages of the original TLS handshake but no other TLS-related computation is performed. This emulates an infinitely fast CPU that still needs to perform the same network operations. Finally, we hypothesize a faster TLS session resumption protocol that removes two messages (one network round-trip), and measure its performance.

Through each of these changes, we can progressively emulate the effects of perfect optimizations, identifying an upper bound on the benefits available from optimizing each component of the TLS system.

2.2.1 *E-Commerce Workload Experiments.* We were interested in closely simulating the load that might be experienced by a popular e-commerce site such as Amazon. While our experiments do not include the database backend processing that occurs in e-commerce sites, we can still accurately model the frontend Web server load.

To capture an appropriate trace, we configured a Squid proxy server and logged the data as we purchased two books from Amazon, one as a new customer and one as a returning customer. The Web traffic to browse Amazon's inventory and select the books for purchase occurs over a regular Web server, and only the final payment and shipping portion occurs with a secure Web server. Of course, the traces we recorded do not contain any plaintext from the secure Web traffic, but they do indicate the number of requests made and the size of the objects transmitted by Amazon to the browser. This is sufficient information to synthesize a workload comparable to what Amazon's secure Web servers might experience. The only value we could not directly measure is the ratio of new to returning Amazon customers. Luckily, Amazon provided this ratio (78% returning customers to 22% new customers) in a quarterly report [Amazon.com 2001]. For our experiments, we assume that returning customers do not retain TLS session state and will thus complete one full TLS handshake every time they wish to make a purchase. In this scenario, based on our traces, the server must perform a full TLS handshake approximately once out of every twelve Web requests. This one full handshake per-purchase assumption may cause us to overstate the relative costs of performing full TLS handshakes versus resumed TLS handshakes, but it does represent a worst case that could well occur in e-commerce workloads.

[1]While TLS also supports operating modes which use no encryption (e.g., `TLS_NULL_WITH_NULL_NULL`), our no-op replacements still use the original data structures even if their values are now all zeros. This results in a more accurate emulation of perfect acceleration.

We created files on disk to match the sizes collected in our trace, and we requested these files in the order they appeared in the trace. When replaying the traces, each client process used at most four simultaneous Web connections, just as common Web browsers do. We also batched together the HTTP requests belonging to the same complete Web page (HTML files, frames, and inline images) and did not begin issuing requests for the next page until all the requests corresponding to the current page had been completed. All three client machines ran 24 of these processes each, causing the server to experience a load comparable to 72 Web clients making simultaneous connections.

2.2.2 *CS Workload Experiments.* We also wished to measure the performance impact of replacing our departmental Web server with a TLS Web server. To do this, we needed to design a system to read a trace taken from the original server and adapt it to our trace-driven TLS Web client. Because we are interested in measuring maximum server throughput, we discarded the timestamps in the server and instead replayed requests from the trace as fast as possible. However, we needed to determine which requests in the original trace would have required a full TLS handshake and which requests would have reused the sessions established by those TLS handshakes. To do this, we assumed that all requests in the trace that originated at the same IP address corresponded to one Web browser. The first request from a given IP address must perform a full TLS handshake. Subsequent requests from that address could reuse the previously acquired TLS session. This assumption is clearly false for large proxy servers that aggregate traffic for many users. For example, all requests from America Online users appear to originate from a small number of proxies. To avoid an incorrect estimation of the session reuse, we eliminated all known proxy servers from our traces. The remaining requests could then be assumed to correspond to individual users' Web browsers. The final trace contained approximately 11,000 sessions spread over 100,000 requests. Another method to differentiate users would have been to use cookies. However, our departmental Web server does not uses cookies, and we couldn't infer user identities based on cookies from our server logs.

In our trace playback system, three client machines ran 20 processes each, generating 60 simultaneous connects, and proving sufficient to saturate the server. The complexity of the playback system lies in its attempt to preserve the original ordering of the Web requests seen in the original trace. Apache's logging mechanism actually records the order in which requests *complete*, not the order in which they were received. As such, we have insufficient information to faithfully replay the trace requests in their original order. Instead, we derive a *partial ordering* from the trace. All requests from a given IP address are totally ordered, but requests from unrelated IP addresses have no ordering (as shown in Figure 5). This allows the system to interleave requests from different users but preserves the ordering of requests coming from the same user.

As a second constraint, we wished to enforce an upper bound on how far the order of requests presented to the Web server could differ from the order of requests in the original trace. Let LS be the index in the trace of the last request served by the server, and let LP be the index in the trace of the last

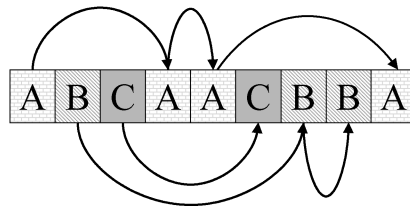


Fig. 5. Trace segment containing requests from users A, B, and C. The arrows represent ordering constraints for the requests. The order of the requests initiated by the same user is preserved. Requests initiated by different users can be interleaved.

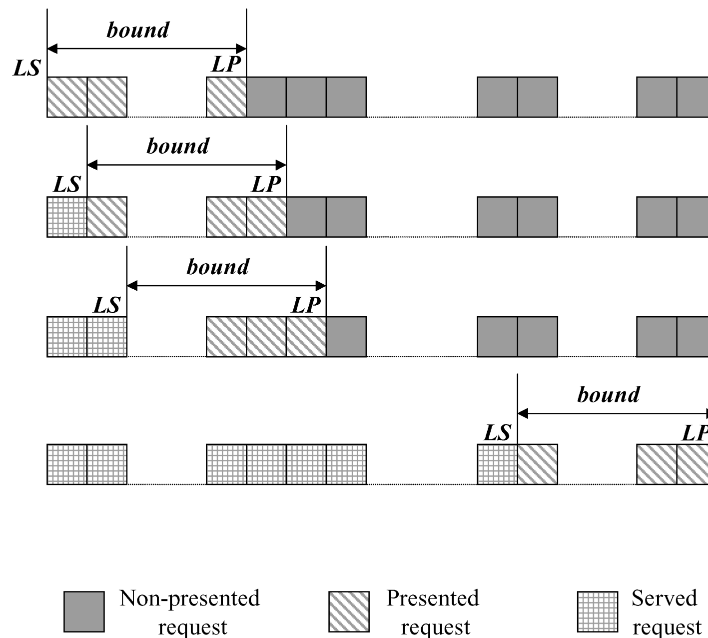


Fig. 6. The difference between the trace index of the last request served by the server, LS, and the last request presented to the server, LP, is bounded by *bound*. As LS increases, LP increases, too, and more requests are presented to the server.

request which was presented to the server by our client programs. The second constraint can then be expressed as follows:

$$LP - LS \leq bound.$$

The set of requests that can be presented to the server at a particular moment is the set of requests whose index is between LS and LP. As LS increases, LP also increases, and new requests can be presented to the Web server. The set of valid requests intuitively behaves as a sliding window over the whole trace (as presented in the Figure 6); the size of the window is equal to *bound*.

If this bound were too small, it would artificially limit the concurrency that the trace playback system could exploit. Tight boundaries create situations in which the server is no longer saturated, and the clients can not begin new

requests until some older request, perhaps for a very large file, completes. If the bound is too large, there would be less assurance that the request ordering observed by the experimental server accurately reflects the original behavior captured in the trace. In practice, we determined by trial and error the minimal boundary for which no client would stall while waiting for requests from other clients to complete. We have determined that a boundary of 10,000 requests was enough for 60 clients playing our trace.

It is impossible to precisely model and replicate the behavior of a Web server for a particular trace; we have attempted to provide a reasonable approximation of that behavior so that our throughput measurements are relevant.

2.3 Implementation Details

In this section, we describe how particular TLS stages were replaced with no-ops. We start with a discussion of replacement of stages within the TLS protocol (Sections 2.3.1 to 2.3.5), then we discuss structural modifications of the TLS protocol (Sections 2.3.6 to 2.3.9). All of the modifications required changes to both our client and server TLS implementations.

2.3.1 *Server Authentication.* The server uses public key cryptography to authenticate itself to the client. TLS provides authentication using several cryptographic methods, such as RSA and Diffie-Hellman [Diffie and Hellman 1976; Rescorla 1999]. We studied the cost of the RSA key exchange and authentication method.

During connection setup, the client performs the RSA encryption of the premaster secret using the server's public key. The server performs the RSA decryption of the premaster secret using its private key. In our TLS configuration, the RSA encryption and decryption can be optionally replaced by *memcpy* operations of the 48-bit premaster secret.

2.3.2 *Bulk Cipher.* In addition to well-known ciphers like RC4 or DES, the TLS protocol also provides a NULL cipher in its supported ciphersuites. When using the NULL cipher, the sequence of steps presented in Figure 4 still takes place; the encryption is simply replaced by a no-op (and as such behaves as the identity function).

2.3.3 *Message Authentication Codes.* MACs are used to ensure the integrity of the data both during TLS connection setup and TLS data exchange. MACs are constructed by using standard cryptographic message digest functions like MD5 or SHA-1 along with a secret key known only to the client and server. An eavesdropper, not knowing the secret key, would be unable to modify a packet and compute the appropriate MAC for that packet. During the TLS connection setup, MACs are used by both sides. The server verifies the client random and the cryptographic parameters suggested by the client, and also computes a MAC for the server random and its public key. During TLS data exchange, the MAC is used to protect the integrity of each message packet. We have instrumented our TLS clients and server to optionally replace all the MAC computations and verifications with no-ops.

2.3.4 *TLS Session Cache.* The cryptographic parameters of a TLS connection, such as the bulk cipher, the message digest algorithm, and the premaster secret, are aggregated in a TLS session which can be stored by both sides. The client can use a previous session to reconnect to the same server and perform an abbreviated TLS connection setup. mod_SSL provides a shared memory cache (using an external shared memory library [Engelschall 2000]), a DBM session cache (using a simple database implementation), and can also function without any TLS session cache at all.

We wanted to emulate a perfect TLS session cache with zero-cost search and insert operations. A basic TLS client performs one connection to the server and the resulting TLS session, which we will call the universal session, is saved by both parts. The server is then restarted and the universal session is loaded in memory by all Apache processes. The TLS clients also load the universal session in memory. Whenever a client would normally perform a full TLS handshake, it performs one; at the server side, the cache insert operation is replaced with a no-op. Whenever the client would normally use a saved TLS session, it instead uses the universal session which the server retrieves from memory. The cache insert and lookup times are thus zero. The TLS clients and servers can optionally revert between a regular TLS cache and our perfect cache.

2.3.5 *Randomness Computations.* The TLS protocol doesn't require that the TLS implementations use a particular random number generator; it requires instead that the pseudorandom number generator used is cryptographically secure. The pseudorandom number generator present in OpenSSL applies a sequence of message digest computations, such as MD5 and SHA-1, to an initial random state such as the current date. In our implementation, we used a 1-byte counter incremented modulo 256 to generate the random bytes required for the server and the client random; this is no longer cryptographically secure but is low cost. In our TLS configuration, the clients and servers can choose between the secure random number generator and our low cost version. However, in order to use the low cost generator, the two parties must also use the perfect TLS session cache; otherwise, the server might generate identical session IDs for connections from different clients.

2.3.6 *Plain Data Exchange.* As discussed in Section 2, the TLS data exchange requires a series of steps to process the message. We replace these steps with regular *read* and *write* I/O calls, eliminating the cryptographic and data copying costs of marshaling packets for transmission. In our experiments, we can use plain data exchange or normal TLS data exchange, independently of any other no-op modifications we might make to the protocol.

2.3.7 *Skeleton TLS.* Designed as a catch-all of all the costs involved in the TLS connection setup, skeleton TLS doesn't perform any computation at all during connection setup. The size of the messages exchanged during regular connection setup was recorded, and skeleton TLS simply sends and receives messages of the same size. The subsequent TLS data exchange is replaced by plain data exchange (as described in Section 2.3.6). Skeleton TLS must be used together with plain data exchange since no security parameters are negotiated.

2.3.8 *Abbreviated Session Resumption.* We wanted to estimate the performance benefits of a hypothetical TLS protocol extension that removes one roundtrip delay from session resumption as well as sending less data. New connections (full TLS handshakes) would still proceed in the same fashion. For a connection which uses a previously saved session (resumed TLS handshakes), the client would send its Hello message along with the HTTP GET request. If the server accepts the session, then it will send its server random and Finished message (see Figure 3) along with the first part of the HTTP response. If the server does not accept the session, a full TLS handshake would occur. Our present implementation is not secure, but it helps us evaluate the performance benefit of such a protocol.

2.3.9 *Minimal TLS Connection Setup.* We wanted to measure the total cost of all the connection setup (both full and abbreviated TLS handshakes). The TLS connection setup processing was replaced with a minimal initialization of the data structures needed during the TLS data exchange. The symmetric keys used for the TLS communication were set to zero. Since the security parameters of the connection are preestablished, the TLS client and server do not need any negotiation in this case. The data exchange can then proceed with the regular TLS data exchange.

3. ANALYSIS OF EXPERIMENTAL RESULTS

Figures 7 and 8 show the main results of our experiments with the Amazon trace and the CS trace, respectively. The achieved throughput is shown on the y-axis. For each system configuration labeled along the x-axis, we show three bars, corresponding to the result obtained with the 500MHz system, the 933MHz single CPU system, and the 933MHz dual CPU system (from left to right).

Three clusters of bar graphs are shown along the x-axis. The left cluster shows three configurations of a complete, functional Web server, the Apache HTTP Web server (Apache), the Apache TLS Web server (Apache+TLS), and the Apache TLS server using an AXL300 RSA accelerator (Apache+TLS AXL300).

The center cluster of bar graphs shows results obtained with various experimental TLS configurations where basic primitives within the TLS protocol were replaced with no-ops. Each configuration is labeled to indicate the key exchange method, bulk encryption algorithm, message authentication code, and caching strategy used. Rather than measuring all possible variations, we measured the configuration where all attributes were replaced by their no-ops alternatives, followed by configurations where each operation was enabled individually. We also measured a few additional configurations discussed in the following. For instance, we measured “PKX, RC4, MD5, shm cache,” a configuration where all RSA operations have been replaced with no-ops, but other operations ran normally in order to expose the performance limits of RSA acceleration techniques.

The right cluster of bar graphs shows measurements of TLS configurations where noncrypto-related TLS functions were removed, and the session resume protocol was simplified. These measurements allow us to understand the costs of the remaining operations in TLS session setup and data exchange.

Label	Description of server configuration
Apache	Apache server
Apache+TLS	Apache server with TLS
Apache+TLS AXL300	Apache server with TLS and AXL300
RSA	RSA protected key exchange
PKX	plain key exchange
NULL	no bulk cipher (plaintext)
RC4	RC4 bulk cipher
noMAC	no MAC integrity check
MD5	MD5 MAC integrity check
no cache	no session cache
shmcache	shared-memory based session cache
perfect cache	idealized session cache (always hits)
no randomness	no pseudorandom number generation (also: NULL, noMAC)
plain	no bulk data marshaling (plaintext written directly to the network)
fast resume	simplified TLS session resume (eliminates one round-trip)
Skeleton TLS	all messages of correct size, but zero data

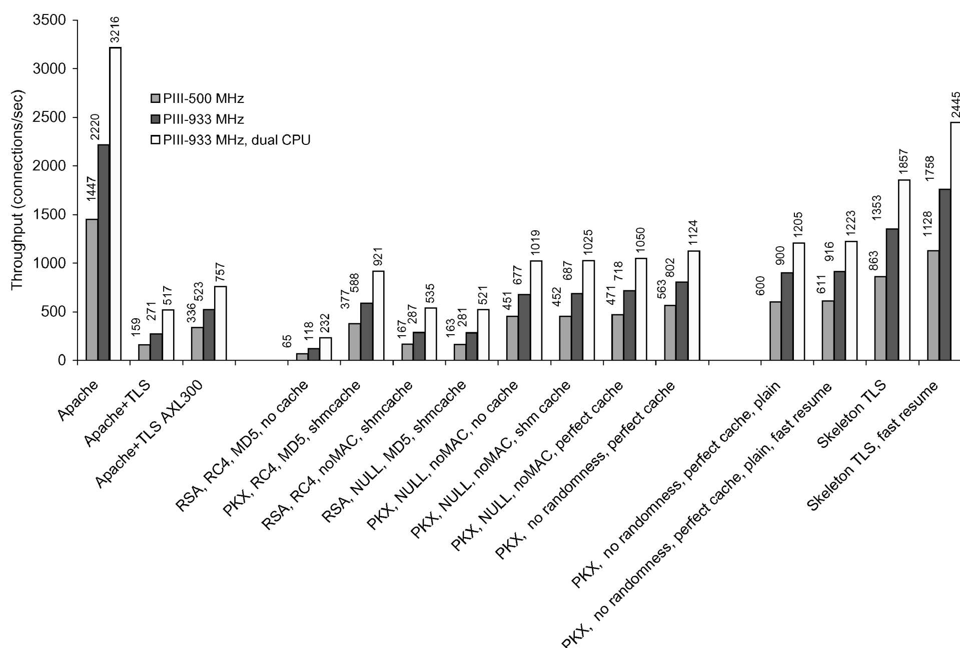


Fig. 7. Throughput for the Amazon trace and different secure Web server configurations on 500MHz, 933MHz, and dual 933MHz servers.

Additionally, we wish to estimate the relative costs of each operation performed by the TLS Web server. To do this, we use Amdahl's Law:

$$Speedup = \frac{1}{(1 - fraction_{enhanced}) + \frac{fraction_{enhanced}}{speedup_{enhanced}}}.$$

Label	Description of server configuration
Apache	Apache server
Apache+TLS	Apache server with TLS
Apache+TLS AXL300	Apache server with TLS and AXL300
RSA	RSA protected key exchange
PKX	plain key exchange
NULL	no bulk cipher (plaintext)
RC4	RC4 bulk cipher
noMAC	no MAC integrity check
MD5	MD5 MAC integrity check
no cache	no session cache
shmcache	shared-memory based session cache
perfect cache	idealized session cache (always hits)
no randomness	no pseudorandom number generation (also: NULL, noMAC)
plain	no bulk data marshaling (plaintext written directly to the network)
fast resume	simplified TLS session resume (eliminates one round-trip)
Skeleton TLS	all messages of correct size, but zero data

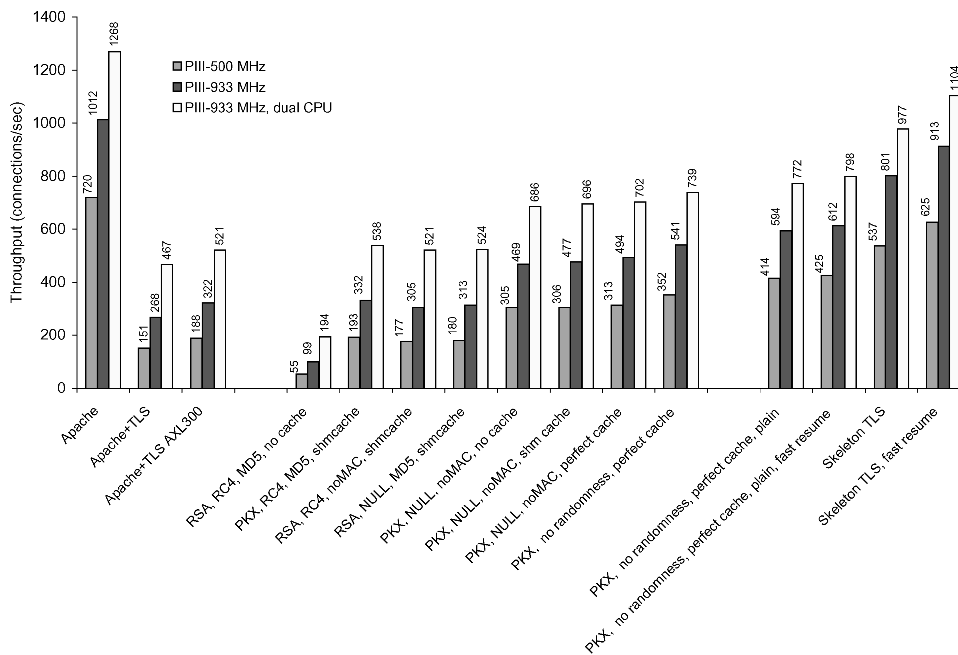


Fig. 8. Throughput for the CS trace and different secure Web server configurations on 500MHz, 933MHz, and dual 933MHz servers.

For each TLS processing component, we have emulated infinite or almost infinite speedup either by removing the component (e.g., for the key exchange method, stream cipher, and message authentication code) or by replacing the component with a much cheaper alternative (e.g., the perfect cache and the

predicted randomness). Thus, Amdahl's Law can be simplified as follows:

$$Speedup = \frac{1}{1 - fraction_{enhanced}}.$$

Since we measured speedups experimentally, we can estimate the cost of individual operations by solving this equation for $fraction_{enhanced}$. The results of these calculations are shown in Figure 9.

In order to directly determine the relative costs of RSA, RC4 and MD5, we replaced each stage individually with a no-op and measured the corresponding server throughput. Other TLS components, such as the TLS session cache, the randomness processing, and TLS packet marshaling, cannot be replaced without also affecting other TLS components. For these cases, we were forced to run some experiments with multiple TLS stages simultaneously disabled. We still estimate the relative cost of each component using Amdahl's Law.

3.1 Impact of TLS on Server Performance

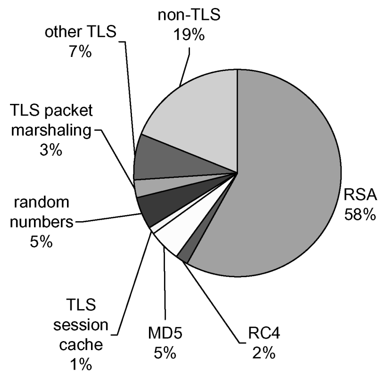
The Apache server, without TLS enabled, achieves between 720 hits/sec and 1268 hits/sec with the CS trace and between 1447 hits/sec and 3216 hits/sec with the Amazon trace. The difference in throughput for the two workloads is due to the increased average file size, 46KB for the CS trace and only 7KB for the Amazon trace, as well as to the increased working set size. Increasing the CPU speed from 500MHz to 933MHz leads to a substantial increase in throughput in each case. The same effect is observed when going from a single 933MHz to a dual 933MHz system.

Apache TLS without the AXL300 served between 151 hits/sec and 467 hits/sec for the CS trace and between 159 hits/sec and 517 hits/sec for the Amazon trace. This confirms that TLS incurs a substantial CPU cost and reduces the throughput by 63 to 89% relative to the insecure Apache. Apache+TLS with the AXL300 served between 188 hits/sec and 521 hits/sec for the CS trace and between 336 hits/sec and 757 hits/sec for the Amazon trace. This shows that, with the use of the AXL300 board, the throughput loss is now only 58 to 77% relative to the insecure Apache.

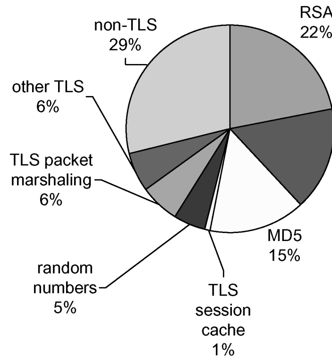
3.2 Impact of Increased CPU Speed

Consider the impact of the available server CPU cycles on the relative cost of TLS. In the configurations with a complete, functional TLS implementation, the 933MHz Pentium III achieves a sizeable increase in throughput (55–71%) relative to the 500MHz Pentium III.

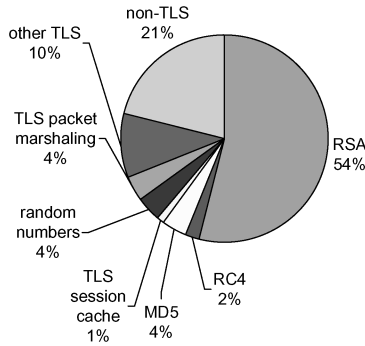
Increasing the CPU speed will not lead to a similar increase in the overall system performance. However, back-of-the-envelope estimates lead to us to believe that neither the PCI bus nor the memory were saturated. The maximum throughput of actual content provided by the servers was around 400Mb/s for regular Apache server experiments. On our server machine, the Compaq DL360, the PCI bus should not be saturated before reaching content transfer rates of around 1.2Gb/s for regular HTTP, according to Kim et al. [2002].



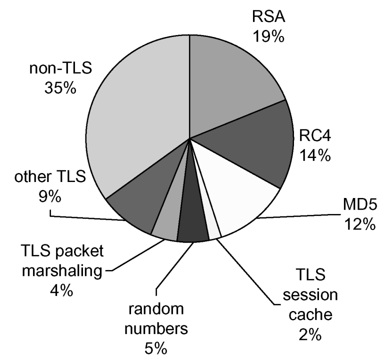
a) Amazon trace for PIII 500MHz



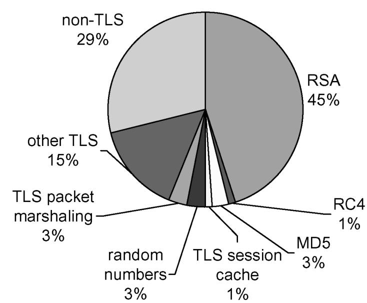
b) CS trace for PIII 500MHz



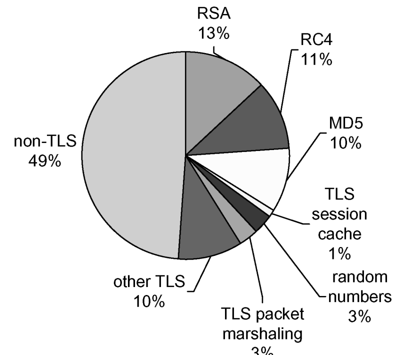
c) Amazon trace for PIII 933MHz



d) CS trace for PIII 933MHz



c) Amazon trace for dual PIII 933MHz



d) CS trace for dual PIII 933MHz

Fig. 9. Relative costs of TLS processing stages for the Amazon trace and the CS trace, using 500MHz, 933MHz, and dual 933MHz server configurations. The sections labeled “Other TLS” refer to the remaining portions of the TLS implementation that we did not specifically single out for no-op tests, and “Non-TLS” refers to other performance costs such as the Apache server and the Linux kernel.

The memory system of our server has a theoretical bandwidth of 8.5Gb/s; our 400Mb/s throughput should likewise not saturate the memory.

Another factor in the Web server performance is the CPU cache reuse. While we didn't measure the cache reuse during our experiments, it is likely that the CS trace, with a size of the document corpus of 530MB, was far too large to hit very often in the CPU cache. The Amazon trace, with a size of the document corpus of around 300KB, would certainly have a higher cache hit rate.

We conclude that the performance of the various TLS processing steps scales well with increased availability of CPU cycles. In the long run, this implies that the performance loss associated with the use of TLS should diminish as CPUs get faster. Of course, faster CPUs can potentially be used to attack cryptosystems more effectively. As a result, stronger, and potentially more CPU intensive, cryptography may become necessary in the future as well.

3.3 Impact of a Second CPU

Consider the impact of dual CPU systems on the relative cost of TLS. In the configuration with a complete, functional TLS implementation, the dual 933MHz Pentium III achieves an increase in throughput (44–61%) relative to the single 933MHz Pentium III. While the increase is smaller then the increase when going from the 500MHz Pentium III to the 933MHz Pentium III, it is still significant. Part of the cost of using dual CPUs is increased operating system overhead. It can be expected that future versions of the operating system will have lower overhead on multiprocessor systems.

3.4 Effectiveness of Accelerator Hardware

The use of the AXL300 accelerator yields a noticeable throughput improvement with the CS trace (11 to 24%) relative to the normal TLS Apache, and a substantial gain with the Amazon trace (46 to 111%) relative to the normal TLS Apache. The reason that the Amazon workload benefits more from the accelerator is that the average session is shorter. As a result, more requests in the Amazon trace require the full TLS handshake with its corresponding RSA operations. The CS trace, with its longer sessions, benefits from the session cache, lowering the effectiveness of accelerated RSA operations. Another contributing factor in the performance difference is the average file size. In the CS trace, with files averaging 46KB, the server spends proportionally more time transmitting files versus performing connection setup when compared to the Amazon trace which has an average file size of 7KB.

The accelerator used for our experiments was a PCI card; typically, going to a PCI card incurs costs such as bus latencies, transfer overheads and forced cache flushes. We don't know the behavior of the driver exactly since we do not have the driver source code. However, as CPU speeds increase, the relative cost of a round-trip on the PCI bus will only increase, reducing the effectiveness of these accelerators.

3.5 Comparative Impact of Accelerator Hardware Versus Faster CPU

A related question we wish to pose is whether it is more advantageous to invest in acceleration hardware or in a faster CPU. The answer depends largely on

the workload. With the CS trace, using a faster CPU is more effective than using an accelerator board. However, with the Amazon trace, the opposite is true. We conclude that sites that only use TLS servers for a small part of their user interaction, as Amazon does for its final purchase validation and payment, will benefit from hardware RSA acceleration. Alternatively, Web sites that use TLS for all of their user interaction and have a correspondingly higher session reuse rate or large objects to transmit may be better served by investing in faster general purpose CPUs.

3.6 Comparative Impact of Accelerator Hardware Versus a Second CPU

Another question we can answer using our methodology is whether it is more profitable to invest in a second CPU or in an RSA accelerator. For the CS trace, the system throughput for a fully-functional Apache TLS server on the dual CPU platform is 467 hits/sec, while the performance of Apache TLS on the single CPU platform and with the RSA decryption turned off is 332 hits/sec. No RSA accelerator, regardless of its speed, would ever help the single CPU system to service more than 332 hits/sec. In this case, it is clear that a second CPU always beats an RSA accelerator. It is important to notice that the performance of the dual CPU server with the RSA accelerator is 521 hits/sec; if cost is no object, one can use both a second CPU and an RSA accelerator.

The picture is somewhat different for the Amazon trace. The performance of the dual CPU server is 517 hits/sec, while the performance of the single CPU TLS server without the RSA decryption is 588 hits/sec. The single CPU TLS server with the RSA accelerator yields 523 hits/sec. Even though this performance is only marginally better than the performance of the dual CPU server, it can be argued that future accelerators will be closer to the ideal performance. At this point, it is important to remember our assumption that the secure server only provides static content. However, a real server machine will perform some other computation besides the secure Web server such as CGIs or running a database server. It is obvious that the increased computing power will be beneficial to this other computation, whereas an RSA accelerator will have no effect on this additional workload. We thus argue that, for a trace similar to the Amazon trace, a second CPU and an RSA accelerator have roughly the same value. As before, when cost is no object, the dual CPU server with the RSA accelerator yields a better performance, namely 757 hits/sec.

3.7 Impact of Session Caching

Our results confirm the findings of prior work [Goldberg et al. 1998] that session caching substantially improves server throughput. Session reuse directly leads to a reduction in the number of RSA operations. However, even in configurations where the RSA operations are assumed to be infinitely fast, session caching is still beneficial, avoiding the extra network traffic and other computations required by the full TLS handshake. The TLS session cache improved throughput by a factor of 2.4–2.7 for our workloads.

3.8 Relative Cost and Impact of Crypto Operations

Figures 7, 8, and 9 quantify the costs of various TLS processing steps. The RSA operations have the dominant cost as expected. Among the remaining operations, the "other TLS" operations stand out as do the MD5 MAC computation and the RC4 stream cipher in the case of the CS trace workload. However, these costs are sufficiently balanced that there is no obvious single candidate for optimization. We note that, even when MD5 is the message integrity function, both MD5 and SHA-1 are used in conjunction in several portions of the TLS protocol such as the pseudorandom function that is when generating key material. In our experiments, no MAC replaces all MD5 and SHA-1 computations with no-ops, throughout the entire TLS protocol with the exception of the MD5 and SHA-1 operation used in the pseudorandom number generator. The cost of the pseudorandom number generator is considered in the following.

3.9 Miscellaneous TLS Operations

Starting with a server in which we replaced RSA, RC4, pseudorandomness computations (which use the MD5 and SHA-1 hash functions), and the session cache with no-ops (labeled PKX, no randomness, perfect cache on the bar charts), we still observed a significant performance deficit relative to the original Apache performance. Removing TLS packet marshaling costs (which involve message fragmentation, memory copy of message fragments into TLS records and sequence number computation) and performing regular *write* calls on the plaintext (labeled PKX, no randomness, perfect cache, plain) resulted in only modest gains, so we decided to try something more radical. We created a Skeleton TLS system that transmitted network messages of the same length as genuine TLS, but otherwise performs no TLS processing whatsoever (see Section 2.3.7). The difference between PKX, NULL, no MAC, no randomness, perfect cache, plain text communication and skeleton TLS covers between 8% and 11% of the total performance cost. Since we have already replaced the data exchanging TLS operations with plain text, the above difference indicates a catch all of every other connection setup-related TLS cost.

Once the "other TLS" costs have been measured, the remainder must be from sources outside the computation associated with TLS. This includes the cost of the underlying Apache Web server, the Linux costs associated with the static content movement, and the Linux costs associated with the extra network traffic required during the TLS handshake.

3.10 Overall Costs of TLS Connection Setup and Data Exchange

To determine the relative cost of connection setup, we have modified the TLS protocol to perform a minimal connection setup and regular encrypted data exchange as described in Section 2.3.9. This involves establishing a generic TCP connection between the client and server, then initializing the data structures used by TLS with the session keys set to zero. We can then compare this with the the full setup/plain data exchange described in Section 2.3.6. The results are presented in Figure 10.

Experiment	CS trace		
	500MHz	**933MHz**	**dual 933MHz**
Apache + TLS	151 (21%)	268 (26%)	467 (37%)
Full setup, plain communication	228 (32%)	367 (36%)	594 (47%)
Minimal setup, encrypted communication	406 (56%)	629 (62%)	850 (67%)

Experiment	Amazon trace		
	500MHz	**933MHz**	**dual 933MHz**
Apache + TLS	159 (11%)	271 (12%)	517 (16%)
Full setup, plain communication	167 (12%)	283 (13%)	531 (17%)
Minimal setup, encrypted communication	772 (53%)	1086 (49%)	1501 (47%)

Fig. 10. Throughput in hits/sec for Apache + TLS full setup with plain communication and minimal setup with encrypted communication for the CS trace and the Amazon trace, using 500MHz, 933MHz, and dual 933MHz servers. Percentages show the throughput relative to non-TLS Apache on the same platform.

Again using Amdahl's Law, we show the cost of the TLS connection setup from 45% to 62% of the total cost for the CS trace and from 65% to 79% of the total cost for the Amazon trace. Replacing the connection setup with a minimal initialization of the data structures yields a throughput improvement of 82 to 168% for the CS trace and 190 to 385% for the Amazon trace.

Likewise, we show that the cost of the encrypted TLS data exchange ranges from 21 to 33% from the total cost for the CS trace and from 3 to 5% from the total cost for the Amazon trace. Replacing the encrypted TLS data exchange with plain communication yields a throughput improvement of 27 to 50% for the CS trace and of 3 to 5% for the Amazon trace. We note that, in this experiment, replacing the encrypted TLS data exchange with plain data exchange only eliminates a portion of the costs associated with RC4 and MD5 which are also used as part of the connection setup protocol.

These measurements imply that optimizations aimed at the connection setup phase of TLS will have a more significant impact on system performance than optimizations aimed at the data exchange phase. It is important to mention that the results could be different for different cipher suites. For example, if RC4 is replaced with a more expensive cipher such as DES, the relative cost of the bulk cipher would increase significantly.

3.11 Potential Impact of Protocol Changes

When considering optimizations for the TLS connection setup phase, we wish to explore potential changes to the TLS protocol aimed at reducing the amount of network traffic. To do this, we used a straw-man implementation of a fast resume TLS variant that optimizes the session resume protocol phase in such a way that two messages and one round-trip network delay are eliminated. The results indicate that the potential throughput improvement of such a hypothetical protocol change is minimal (the server throughput improved by 1 to 3% between PKX, no randomness, perfect cache, plain and PKX, no randomness, perfect cache, plain, fast resume). Therefore, optimizations aimed at

reducing the volume of network traffic will have little effect on TLS server throughput. However, such optimizations could have other benefits, particularly for bandwidth-limited clients. Shacham and Boneh [2002] propose a fast-track TLS handshake mechanism that performs TLS session resuming with fewer messages.

3.12 Dynamic Content Generation

A common question is how to apply performance measurements such as those performed in this article with static content to the relatively common case of dynamic content generation which often involves running custom server plug-in code that makes database queries and assembles HTML on the fly. Our experiments focus on TLS Web servers that serve static content, discerning among the TLS and non-TLS costs. If the TLS Web server is generating dynamic HTML, then the new load will obviously impact server throughput. In the pie charts of Figure 9, this additional overhead should be reflected in the non-TLS sections of the pie charts which could be increased appropriately, scaling down the TLS sections of the pie chart such that their relative costs remain the same.

3.13 Summary of Results

We can summarize the result of our experiments as follows.

- TLS imposes a factor of 3.4 to 9 overhead over an insecure Web server.
- The largest performance cost in the TLS Web server is the public key cryptography (13% to 58%).
- Non-TLS performance costs range from 19 to 45% of the total cost. These costs reflect the Apache and Linux kernel involvement in serving the static content but also include the kernel costs of the extra network traffic needed by the TLS connection setup. However, these costs don't cover any computation associated with TLS. The Skeleton TLS, fast resume represents a stripped version of the TLS protocol in which the server receives a single message from the client and then the regular HTTP protocol takes place. By removing this single message necessary to establish a connection—and saving the cost of an extra `read` system call per resumed session—we achieved the same performance as the regular HTTP server.
- The costs of marshaling TLS data structures, computing connection keys from the premaster secret, and executing other miscellaneous operations within TLS consumes between 6% and 15% of the total performance cost. Reducing the session resumption protocol by two messages and one round-trip delay had a negligible impact on performance (1 to 3%).
- Adding an RSA accelerator, a common commercial approach to addressing TLS server performance issues, has widely different effects on server throughput depending on the session reuse rate of the requests seen by the TLS server. For low session reuse rates and with smaller Web objects, the RSA accelerator can result in a 46–111% performance improvement (a factor of 2 improvement in hit rate). For high session reuse rates and with

larger Web objects, however, the RSA accelerator only resulted in a 11–24% performance improvement.

—This improvement is bounded at 78–137% (for the Amazon trace) or 15–27% (for the CS trace), regardless of how fast the RSA accelerator can run.

—The TLS session cache is effective; it improved throughput by a factor of 2.4–2.7 for the CS trace and 2.2–2.4 for the Amazon trace relative to a server with no cache.

—The costs of the non-RSA cryptographic operations, such as RC4, MD5, and pseudorandom number generation, performed by TLS are relatively balanced. Hardware acceleration for any individual operation would yield only modest performance improvements.

—TLS appears to be purely CPU bound as optimizations intended to reduce network traffic have little effect on server throughput. Such optimizations might have an impact for faster CPUs or for network with large delays and packet losses.

—The CPU costs associated with TLS connection setup have a more significant impact on TLS server throughput than the CPU costs associated with TLS data exchange.

—As CPUs become faster, the relative cost of cryptographic components of TLS (RSA, MD5, RC4) decreases, shifting the load to non-TLS components such as the Web server and the operating system kernel. This implies that faster CPUs will eventually bridge the gap between secure and nonsecure Web servers. In the long term, as CPU performance continues to grow, TLS overhead will diminish. Even before then, TLS Web performance is strongly related to the performance of unencrypted Web servers. Faster regular Web servers make for faster TLS Web servers.

—If given the choice between a dual CPU server and a single CPU server with an RSA accelerator, the dual CPU server often dominates, and at worst equals, the performance of the single CPU server with an RSA accelerator. If adding a cryptographic card costs at least as much as the second CPU, than investing in faster or additional CPUs seems to be a preferable strategy for maximizing TLS Web server throughput.

4. FUTURE WORK

This article has studied the performance of TLS Web service from a single server. It has not considered the larger environment that often occurs in an e-commerce site such as load-balancing frontend switches with replicated clusters of Web servers and a database backend. There have already been some efforts to study these environments. For example, the Zeus Web server performance tuning guide [Zeus Technology 2001] mentions the importance of sharing TLS sessions across Web servers in a cluster. We plan to study the interaction of different cluster load-balancing strategies (such as used in LARD [Pai et al. 1998]) with TLS Web servers.

This article also presents data that predicts what might happen to TLS performance as CPUs become faster in the coming years. Rather than our

no-op approach to performance measurement, a more accurate technique would be to measure TLS performance using a precise machine simulator such as SimOS [Rosenblum et al. 1997] or RSIM [Pai et al. 1997]. Such simulators would allow us to predict the effects of future architectural trends on TLS performance. Likewise, many Web servers such as Zeus and Flash [Pai et al. 1999a] are known to radically outperform Apache. As the available CPU increases and cryptographic operations are no longer the primary performance bottleneck, these other server architectures may also prove to be faster at TLS Web service than Apache.

5. RELATED WORK

The related work falls into two main categories:

—TLS performance analysis and optimizations,
—benchmarking and improvements of underlying Web server technology.

5.1 TLS Performance Analysis and Optimizations

We have measured the performance of our TLS Web servers using home-brew workloads. Efforts toward the standardization of SSL/TLS benchmarks have resulted in SPECWeb99_SSL Standard Performance Evaluation Corporation [2002]. SPECWeb99_SSL is an extension of SPECWeb99 Standard Performance Evaluation Corporation [1999]. While we are measuring server throughput for a fixed number of clients (enough to saturate the server CPU), SPECWeb99_SSL measures the maximum number of simultaneous connections requesting the predefined benchmark workload that a secure Web server is able to support while still meeting specific throughput and error rate requirements. The SPEC99_SSL workload includes both static and dynamic content, while our workload only includes static files. Under the session resumption scheme used by the SPEC benchmark, the TLS Web server performs a full handshake approximately once out of every five Web requests. This means that under the SPEC workload, the server is performing twice as many full TLS handshakes as either the Amazon or the CS workloads.

Researchers have studied algorithms and systems to accelerate RSA operations. Boneh and Shacham [2001] have designed a software system to perform RSA operations, together in batches, at a lower cost than doing the operations individually. Batching b RSA operations yields good results for $b = 4$ (a speedup improvement of 2.5), and $b = 8$ (a speedup improvement of 5). The speedups are relative to the RSA decryption only. Integrating the RSA operations-batching with TLS Web servers involves changes similar to adding support for a hardware accelerator. While a batching factor of 8 yields better results, it imposes an increased latency on TLS handshakes to collect the batches for processing.

Dean et al. [2001] have designed a network service, offloading the RSA computations from Web servers to dedicated cryptoservers with RSA hardware. The connections between the clients of the cryptoserver and the cryptoserver itself are protected by symmetric key encryption. The costs of the cryptoserver operations can be amortized among a large number of clients.

An alternative approach is to distribute TLS processing stages among multiple machines. Mraz [2001] has designed an architecture that offloads all of the cryptographic operations to proxy servers that sit in front of Web servers. A Web server itself only performs unencrypted HTTP transactions. Similar approaches are taken by Intel [2002], Alteon [2002], and others.

Apostolopoulos et al. [1999] studied the cost of TLS connection setup, RC4 and MD5, and proposed TLS connection setup protocol changes. The methodology employed focused on the time spent during the TLS connection and on the maximal throughput of the RC4 and MD5 stage rather than on the impact of that particular stage in the throughput of the whole server. One optimization of the TLS handshake protocol involves the caching of server certificates by clients which obviates the need for the server to resend its certificate to the client. A second approach reverses the roles of client and server with respect to generating the premaster secret.

Research has suggested that, indeed, session caching helps Web server performance. Goldberg et al. [1998] have shown that TLS session caching can reduce TLS interaction time (connection setup and data exchange time) by 15% to 50% to geographically diverse U.S. sites. On the real Internet, characterized by delays and packet losses, reducing the number of messages also reduces the TLS connection setup latency.

For the emerging wireless Internet, even small reductions in the number of messages needed for TLS connection establishment might have a significant impact. Shacham and Boneh [2002] propose a fast-track handshake mechanism for TLS connection resumption which might be beneficial in such low-bandwidth and high-latency environments.

The cost of the IPsec protocol was investigated by Miltchev and Ioannidis [2002]. Their experiments showed that using cryptographic cards was effective for large packet sizes but less effective for small (up to 40 bytes) packets; they proposed using a hybrid solution (software encryption for small packets, hardware encryption for large packets) or integrating cryptographic functionality into the network interface.

5.2 Web Servers Benchmarking and Design

Considerable work has been done on improving the raw I/O processing speed of operating systems. Most of this work has focused on improving message latency and on delivering the network's full bandwidth to application programs [McKenney and Dove 1992; Anderson and Pasquale 1995; Bas et al. 1995; Chen and Bershad 1993; Druschel 1994; Druschel et al. 1993; Druschel et al. 1994; Druschel and Peterson 1993; Druschel et al. 1992; Thadani and Khalidi 1995; Montz et al. 1994; Mosberger et al. 1996; Pai et al. 1999b; Banks and Prudence 1993; Brendan et al. 1993; Smith and Traw 1993; Edwards et al. 1994].

More recently, researchers have started to look specifically at the performance of Internet servers on general purpose operating systems. There has been considerable prior work in performance analysis and benchmarking of conventional Web servers. Nahum et al. [2001] showed that a useful metric for server performance is the maximum throughput (capacity); packet loss in wide-area networks could reduce server capacity by 50%.

The Standard Performance Evaluation Corporation proposed SPECWeb99 Standard Performance Evaluation Corporation [1999] as a standardized workload that measures simultaneous connections rather than HTTP operations. Its workload contains both static and dynamic files. Banga and Druschel [1999] propose and evaluate a new method for Web traffic generation that can generate bursty traffic with peak loads that exceed the capacity of the server.

Mogul [1995] studied a California election server. The study showed that the use of a new TCP connection for each HTTP request was wasting server resources. This problem was addressed by HTTP 1.1, in which several HTTP requests can be served over the same TCP connection. Another early study performed at NCSA showed that servers using preforked processes perform better than servers that fork a new process for each request [McGrath 1995].

Web server developers also provide tuning guides for their systems [Laurie and Laurie 1999]. One suggestion for BSD and SunOS operating systems is to increase SOMAXCONN, the kernel parameter that specifies the maximum number of unaccepted socket connections that can be waiting in queue (SOMAXCONN is often referred to as the *listen()* queue limit).

Also, there have been some proposals for enhancing the Unix system call interface to support I/O in Internet servers more efficiently. These proposals address support for zero-copy I/O [Pai et al. 1999b; Thadani and Khalidi 1995; Chu 1996], and reducing the number of system calls in the typical I/O path [Hu et al. 1997]. Experimental results [Nahum et al. 2002] have also shown that reducing the number of TCP packets improves Web server performance.

In response to observations about the large context-switching overhead of process-per-connection servers, many recent servers [Chankhunthod et al. 1996; Network Appliance, Inc. 2002; Wessels 2002; Poskanser 2002; Zeus Technology 2002; Pai et al. 1999a] have used event-driven architectures. Measurements of these servers under laboratory conditions indicate an order of magnitude performance improvement [Chankhunthod et al. 1996; Schechte and Sutaria 1997]. However, some studies of such servers under real workloads [Maltzahn et al. 1997; Fox et al. 1997] indicate significant state-management overhead and poor scalability. Another study [Banga et al. 1998; Banga and Mogul 1998] discovered that event-driven Web servers suffer a throughput decrease and poor scalability under delays characteristic of wide area networks, caused by an inefficient implementation of the *select* system call. A scalable implementation of *select* [Banga et al. 1999] was proposed and evaluated; results show that it improved both server throughput and scalability. In a similar vein, Welsh et al. [2001] proposed a system for building Web services that would be robust under overload conditions by monitoring queues between internal processing stages.

6. CONCLUSIONS

We have presented a systematic analysis of the performance of the Apache Web server with the mod_SSL extension for secure TLS delivery of Web pages. Our methodology was to exercise the Web server with a trace-based workload, while selectively replacing TLS operations with no-ops. By measuring the differences in the resulting server throughput, our measurements are more accurate than

results that could otherwise be obtained through traditional CPU profilers or microbenchmarks.

Our measurements show that RSA computations are the single most expensive operation in TLS, consuming 13–58% of the time spent in the Web server with the variation coming from both available server CPU cycles and from workload variations. Other TLS costs are balanced across other the various cryptographic and protocol processing steps. Optimizations aimed at improving RSA operation throughput, whether through algorithmic enhancements, cryptographic co-processors, or simply increasing raw CPU speed, will continue to be profitable. However, even with ideal zero-cost RSA operations, there is still a large gulf between TLS server performance and traditional, unencrypted server performance.

Hardware acceleration is fairly effective in absorbing the cost of the RSA operations. Our results indicate that accelerators have a significant impact on the throughput of dedicated secure servers for e-commerce sites since such sites generally do most of their user interaction with cryptography disabled and only use cryptography for the final checking-out phase. As a result, these systems experience relatively low session reuse rates and benefit from RSA acceleration hardware. Acceleration appears to be less effective for sites for which all requests are handled by a TLS server. These sites experience higher session reuse rates. For such sites, investing in a faster CPU may prove more effective.

If given the choice between a dual CPU server and a single CPU server with an RSA accelerator, the dual CPU server often dominates and, at worst, equals the performance of the single CPU server with an RSA accelerator. Dollar for dollar, investing in faster or additional CPUs seems to be a preferable strategy for maximizing TLS Web server throughput.

Our results suggest that future efforts to optimize TLS server throughput should focus on reducing the CPU costs of the TLS connection setup phase rather than the TLS data exchange phase. Likewise, efforts to redesign or extend the TLS protocol should consider the CPU costs of all operations performed during connection setup not just the RSA operations.

However, it should be noted that as CPUs become faster, the relative cost of cryptographic components of TLS (RSA, MD5, RC4) decreases, shifting the load to non-TLS components such as the Web server and the operating system kernel. This means that faster CPUs will eventually bridge the gap between secure and nonsecure Web servers. In the long term, as CPU performance continues to grow, TLS overhead will diminish, and research should focus on designing more efficient Web servers. Even today, TLS Web performance is strongly related to the performance of unencrypted Web servers; faster regular Web servers make for faster TLS Web servers.

ACKNOWLEDGMENTS

Vincent Vanackere and Algis Rudys contributed to early versions of this work. Eric Nahum, Mohit Aron, and Adam Stubblefield also shared many useful ideas and opinions. We also thank our anonymous referees for all of their valuable suggestions.

REFERENCES

ALTEON. 2002. Alteon web switching Portfolio. http://www.nortelnetworks.com/products/01/alteon/alt180/.

AMAZON.COM. 2001. Amazon.Com releases 2001 first quarter results. Press Release. http://www.sec.gov/Archives/edgar/data/1018724/000095010901500823/dex991.htm.

ANDERSON, E. W. AND PASQUALE, J. 1995. The performance of the container shipping I/O system. In *Proceedings of the 15th ACM Symposium on Operating System Principles*. Copper Mountain, CO, ACM, 229.

APOSTOLOPOULOS, G., PERIS, V., AND SAHA, D. 1999. Transport layer security, How much does it really cost? In *Proceedings of the 18th Conference on Computer Communications*. New York, NY.

BANGA, G. AND DRUSCHEL, P. 1999. Measuring the capacity of a Web server under realistic loads. *World Wide Web J.* (Special Issue on World Wide Web Characterization and Performance Evaluation) *2*, 1–2, 69–83.

BANGA, G., DRUSCHEL, P., AND MOGUL, J. C. 1998. Better operating system features for faster network servers. In *Proceedings of the Workshop on Internet Server Performance*. Condensed version appears in ACM SIGMETRICS *Performance Evaluation Review* 26, 3, 23–30.

BANGA, G. AND MOGUL, J. C. 1998. Scalable kernel performance for Internet servers under realistic loads. In *Proceedings of the 1998 Usenix Technical Conference*.

BANGA, G., MOGUL, J. C., AND DRUSCHEL, P. 1999. A scalable and explicit event delivery mechanism for UNIX. In *Proceeding of the Usenix 1999 Annual Technical Conference*. Monterey, CA.

BANKS, D. AND PRUDENCE, M. 1993. A high-performance network architecture for a pa-risc workstation. *IEEE J. Selected Area Comm. 11*, 2 (Feb), 191–202.

BAS, A., BUCH, V., VOGELS, W., AND VON EICKEN, T. 1995. U-Net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th ACM Symposium on Operating System Principles*. 40–53.

BONEH, D. AND SHACHAM, H. 2001. Improving SSL handshake performance via batching. In *Proceedings of the RSA Conference*. San Francisco, CA.

BRADLEY, J. AND DAVIES, N. 1995. Analysis of the SSL protocol. Tech. Rep. CSTR-95-021. University of Bristol.

BRENDAN, C., TRAW, S., AND SMITH, J. M. 1993. Hardware/software organization of a high-performance atm host interface. *IEEE J. Selected Area Comm. 11*, 2 (Feb), 240–253.

BUHLER, P., EIRICH, T., STEINER, M., AND WAIDNER, M. 2000. Secure password-based cipher suite for TLS. In *Proceedings of the 6th Network and Distributed Systems Security Symposium*. San Diego, CA, 129–142.

CHANKHUNTHOD, A., DANZIG, P. B., NEERDAELS, C., SCHWARTZ, M. F., AND WORRELL, K. J. 1996. A hierarchical Internet object cache. In *Proceedings of the 1996 Usenix Technical Conference*.

CHEN, J. B. AND BERSHAD, B. N. 1993. The impact of operating system structure on memory system performance. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*. 120–133.

CHU, J. 1996. Zero-copy TCP in Solaris. In *Proceedings of the 1996 USENIX Technical Conference*. San Diego, CA.

COMPAQ. 2001. The AXL300 RSA accelerator. http://www.compaq.com/products/servers/security/axl300/.

DEAN, D., BERSON, T., FRANKLIN, M., SMETTERS, D., AND SPREITZER, M. 2001. Cryptology as a network service. In *Proceedings of the 7th Network and Distributed System Security Symposium*. San Diego, CA.

DEAN, D. AND STUBBLEFIELD, A. 2001. Using client puzzles to protect TLS. In *Proceedings of the 7th Network and Distributed System Security Symposium*. San Diego, CA.

DIERKS, T. AND ALLEN, C. 1999. *The TLS Protocol, Version 1.0*. Internet Engineering Task Force. RFC-2246, ftp://ftp.isi.edu/in-notes/rfc2246.txt.

DIFFIE, W. AND HELLMAN, M. E. 1976. New directions in cryptography. *IEEE Trans. Inform. Theory 22*, 6, 644–654.

DRUSCHEL, P. 1994. Operating systems support for high-speed networking. Tech. Rep. TR 94-24, Department of Computer Science, University of Arizona.

DRUSCHEL, P., ABBOTT, M. B., PAGELS, M. A., AND PETERSON, L. L. 1993. Network subsystem design. *IEEE Network 7*, 4 (July) 8–17.

DRUSCHEL, P., DAVIE, B. S., AND PETERSON, L. L. 1994. Experiences with a high-speed network adaptor: A software perspective. In *Proceedings of the SIGCOMM 1994 Conference*. London, UK, 2–13.

DRUSCHEL, P. AND PETERSON, L. L. 1993. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*. 189–202.

DRUSCHEL, P., PETERSON, L. L., AND HUTCHINSON, N. C. 1992. Beyond micro-kernel design: Decoupling modularity and protection in Lipto. In *Proceedings of the 12th International Conference on Distributed Computing Systems*. Yokohama, Japan.

EDWARDS, A., WATSON, G., LUMLEY, J., BANKS, D., CALAMVOKIS, C., AND DALTON, C. 1994. User-space protocols deliver high performance to applications on a low-cost Gb/s LAN. In *Proceedings of the SIGCOMM 1994 Conference*. London, UK.

ENGELSCHALL, R. S. 2000. mm - Shared Memory Library. http://www.engelschall.com/sw/mm/.

FOX, A., GRIBBLE, S. D., CHAWATHE, Y., BREWER, E. A., AND GAUTHIER, P. 1997. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating System Principles*. San Malo, France.

FREIER, A. O., KARLTON, P., AND KOCHER, P. C. 1996. The SSL Protocol, Version 3.0. Netscape. http://home.netscape.com/eng/ssl3/draft302.txt.

GOLDBERG, A., BUFF, R., AND SCHMITT, A. 1998. Secure Web server performance dramatically improved by caching SSL session keys. In *Proceedings of the Workshop on Internet Server Performance*. Madison, WI.

HALEVI, S. AND KRAWCZYK, H. 1999. Public-key cryptography and password protocols. *ACM Trans. Inform. Syst. Secur. 2*, 3, 230–268.

HESS, A., JACOBSON, J., MILLS, H., WAMSLEY, R., SEAMONS, K. E., AND SMITH, B. 2002. Advanced client/server authentication in TLS. In *Proceedings of the 8th Network and Distributed System Security Symposium*. San Diego, CA.

HU, J. C., PYRALI, I., AND SCHMIDT, D. C. 1997. Measuring the impact of event dispatching and concurrency models on Web server performance over high-speed networks. In *Proceedings of the 2nd Global Internet Conference.*

INTEL. 2002. Intel(R) AAD8125Y and AAD8120Y e-Commerce Directors. http://developer.intel.com/design/network/products/security/aad812x.htm.

KIM, H., PAI, V. S., AND RIXNER, S. 2002. Increasing Web server throughput with network interface data caching. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*. San Jose, CA.

LAURIE, B. AND LAURIE, P. 1999. *Apache: The Definitive Guide*, 2nd Ed. O'Reilly, Cambridge, MA.

MALTZAHN, C., RICHARDSON, K. J., AND GRUNWALD, D. 1997. Performance issues of enterprise level Web proxies. In *Proceedings of the ACM SIGMETRICS '1997 Conference*. Seattle, WA.

MCGRATH, R. E. 1995. Performance of several HTTP demons on an HP 735 workstation. http://www.ncsa.uiuc.edu/InformationServers/Performance/V1.4/report.html.

MCKENNEY, P. AND DOVE, K. 1992. Efficient demultiplexing of incoming TCP packets. In *Proceedings of the SIGCOMM 1992 Conference*. Baltimore, MD, 269–279.

MILTCHEV, S. AND IOANNIDIS, S. 2002. A study of the relative costs of network security protocols. In *Proceedings of the 2002 USENIX Technical Conference*. Monterey, CA.

MITCHELL, J. C. 1998. Finite-state analysis of security protocols. In *Proceedings of the Computer Aided Verification*. 71–76.

MOGUL, J. C. 1995. Network behavior of a busy Web server and its clients. Tech. Rep. WRL 95/5, DEC Western Research Laboratory, Palo Alto, CA.

MONTZ, A. B., MOSBERGER, D., O'MALLEY, S. W., PETERSON, L. L., AND PROEBSTING, T. A. 1994. Scout: A communications-oriented operating system. Tech. Rep. TR 94-20, Department of Computer Science, University of Arizona.

MOSBERGER, D., PETERSON, L., BRIDGES, P., AND O'MALLEY, S. 1996. Analysis of techniques to improve protocol latency. In *Proceedings of the SIGCOMM '1996 Conference*. Palo Alto, CA.

MRAZ, R. 2001. Secure Blue: An architecture for a high volume SSL Internet server. In *Proceedings of the 17th Annual Computer Security Applications Conference*. New Orleans, LA.

NAHUM, E. M., BARZILAI, T., AND KANDLUR, D. 2002. Performance issues in WWW servers. *IEEE/ACM Trans. Network. 10*, 1, 2–11.

NAHUM, E. M., ROSU, M., SESHAN, S., AND ALMEIDA, J. 2001. The effects of wide-area conditions on WWW server performance. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. Cambridge, MA.

NETCRAFT. 2001. The Netcraft Secure Server Survey. http://www.netcraft.com/ssl/.

NETWORK APPLIANCE, INC. 2002. Netcache. http:/www.netapp.com/products/netcache.

PAI, V. S., ARON, M., BANGA, G., SVENDSEN, M., DRUSCHEL, P., ZWAENEPOEL, W., AND NAHUM, E. 1998. Locality-aware request distribution in cluster-based network servers. In *Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, San Jose, CA.

PAI, V. S., DRUSCHEL, P., AND ZWAENEPOEL, W. 1999a. Flash: An efficient and portable Web server. In *Proceeding of the Usenix 1999 Annual Technical Conference*. Monterey, CA, 199–212.

PAI, V. S., DRUSCHEL, P., AND ZWAENEPOEL, W. 1999b. I/O-Lite: A unified I/O buffering and caching system. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*. New Orleans, LA.

PAI, V. S., RANGANATHAN, P., AND ADVE, S. V. 1997. RSIM: An execution-driven simulator for ILP-based shared-memory multiprocessors and uniprocessors. In *Proceedings of the 3rd Workshop on Computer Architecture Education*.

PAULSON, L. C. 1999. Inductive analysis of the Internet protocol TLS. *ACM Trans. Inform. Syst. Secu. 2*, 3, 332–351.

POSKANSER, J. 2002. thhtpd. http:/www.acme.com/software/thttpd/.

RESCORLA, E. 1999. Diffie-Hellman Key Agreement Method. Internet Engineering Task Force. RFC-2631, http://www.ietf.org/rfc/rfc2631.txt.

RIVEST, R., SHAMIR, A., AND ADLEMAN, L. M. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Comm. ACM 21*, 2 (Feb.) 120–126.

ROSENBLUM, M., BUGNION, E., DEVINE, S., AND HERROD, S. 1997. Using the SimOS machine simulator to study complex computer systems. *ACM Trans. Model. Comput. Simul.* Special Issue on Computer Simulation 7, 1, 78–103.

SCHECHTE, S. E. AND SUTARIA, J. 1997. A study of the effects of context switching and caching on HTTP server performance. http:/www.eecs.harvard.edu/ stuart/Tarantula/FirstPaper.html.

SCHNEIER, B. 1996. *Applied Cryptography*, 2nd Ed. John Wiley and Sons, New York, NY.

SHACHAM, H. AND BONEH, D. 2002. Fast-track session establishment for TLS. In *Proceedings of the 8th Network and Distributed System Security Symposium*. San Diego, CA.

SMITH, J. M. AND TRAW, C. B. S. 1993. Giving applications access to Gb/s networking. *IEEE Network 7*, 4 (July), 44–52.

STANDARD PERFORMANCE EVALUATION CORPORATION. 1999. SPECWeb99. http://www.specbench.org/osg/Web99/.

STANDARD PERFORMANCE EVALUATION CORPORATION. 2002. SPECWeb99_SSL. http://www.specbench.org/osg/Web99ssl/.

THADANI, M. N. AND KHALIDI, Y. A. 1995. An efficient zero-copy I/O framework for UNIX. Tech. Rep. SMLI TR-95-39, Sun Microsystems Laboratories, Inc.

VIEGA, J., MESSIER, M., AND CHANDRA, P. 2002. *Network Security with OpenSSL*, 1st Ed. O'Reilly, Cambridge, MA.

WAGNER, D. AND SCHNEIER, B. 1996. Analysis of the SSL 3.0 protocol. In *Proceedings of the 2nd USENIX Workshop on Electronic Commerce*. Oakland, CA.

WELSH, M., CULLER, D., AND BREWER, E. 2001. Seda: An architecture for well-conditioned, scalable Internet services. In *Proceedings of the 18th ACM Symposium on Operating System Principles*. ACM, Chateau Lake Louise, Canada.

WESSELS, D. 2002. Squid Web proxy cache. http:/www.squid-cache.org.

WIRELESS APPLICATION PROTOCOL FORUM. 2001. Wireless Transport Layer Security. WAP forum. http://www1.wapforum.org/tech/terms.asp?doc=WAP-261-WTLS-20010406-a.pdf.

ZEUS TECHNOLOGY. 2001. Zeus performance tuning guide. http://support.zeus.com/faq/entries/ssl_tuning.html.

ZEUS TECHNOLOGY. 2002. Zeus Web server. http://www.zeus.co.uk/.

Received July 2002; revised May 2004; accepted February 2005