Performance Analysis of TLS Web Servers

Cristian Coarfa, Peter Druschel and Dan S. Wallach
Department of Computer Science
Rice University

PRELIMINARY DRAFT

Abstract

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. While a number of studies have been done on TLS performance and vendors sell a number of TLS accelerating products, no comprehensive study exists that analyzes the performance costs of TLS. Little is understood about TLS's performance overhead beyond acknowledging that the cost of its RSA operations are quite high. Our methodology is to profile TLS web servers with trace-driven workloads, replacing individual components inside TLS with no-ops, and measuring the observed increase in server throughput. From this, we can estimate the relative costs of each component within TLS and can predict the areas for which future optimizations would be worthwhile. We also consider protocol-level optimizations, measuring the limits of server performance if TLS session resumption could be performed with one fewer network roundtrip. Our results can guide both software and hardware vendors looking to tune the performance of TLS applications and the TLS protocol.

1 Introduction

Secure communication is an intrinsic demand of today's world of online transactions. Two methods have emerged as Internet standards: IPsec (IP security) and SSL/TLS [9]. Original designed at Netscape for its web browsers and servers, Netscape's Secure Socket Layer (SSL) has been standardized by the IETF and is now called Transport Layer Security (TLS). IPsec is largely transparent to networking applications, running at the network layer, and is most commonly deployed for virtual private networks. TLS, on the other hand, runs at the transport layer above existing protocols like TCP. As TLS is most commonly used for secure web applications, such as online banking and e-commerce, our goal is to provide a comprehensive performance analysis of TLS web servers. While previous attempts to address TLS performance have focused on specific processing stages, such as the RSA operations or the session cache, we analyze TLS web servers as *systems*, measuring page-serving throughput under trace-driven workloads.

TLS provides a flexible architecture that supports a number of different public key ciphers, bulk encryption ciphers, and message integrity functions. In its most common web usage, TLS uses 1024-bit RSA encryption to transmit a secret used to initialize a 128-bit RC4 stream cipher and to use MD5 as a keyed hash function. (Details of these algorithms can be found in Schneier [24] and most other introductory cryptography texts.)

TLS web servers incur a significant performance penalty relative to a regular web server running on the same platform (as little as a factor of 3.4 to as much as a factor of 9, in our own experiments). As a result of this cost, a number of hardware accelerators are offered by vendors such as nCipher or 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 [7] have designed a software system to perform RSA operations together in batches, at a lower cost than doing the operations individually. Dean *et al.* [8] have designed a network service, offloading the RSA computations from web servers to dedicated servers with RSA hardware.

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 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 [10].

Likewise, there has been considerable prior work in performance analysis and benchmarking of conventional web servers [15, 11, 16, 4, 17], performance optimizations of web servers, performance oriented web server design, and operating system support for web servers [12, 21, 5, 6, 20]. To the best of our knowledge, this is the first systematic experimental analysis of TLS web server performance.

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, whether through 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. While creating an optimized protocol and proving it to be secure would be a significant effort, we can 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.

Section 2 presents an overview of the TLS protocol. Section 3 explains how we performed our experiments and what we measured. Section 4 analyzes our measurements in detail. Our paper wraps up with future work and conclusions.

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 in quite complex. Readers looking for complete details are encouraged to read the RFC [9]. 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 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):

- Exchange hello messages to agree on algorithms, exchange random values, and check for session resumption.
- Exchange the necessary cryptographic parameters to allow the client and server to agree on a "premaster secret".
- Exchange certificates and cryptographic information to allow the client and server to authenticate themselves. [In our experiments, we do not use client certificates.]
- Generate a "master secret" from the premaster secret chosen by the client and 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. The premaster 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 premaster secret (known to client and server but not to any eavesdropper) can be used to create a new master secret, a connection-specific value from which the connection's encryption keys, message authentication keys, and initialization vectors are derived.

After the setup protocol is completed, the steady-state communication phase begins. Prior to transmission, the data is broken into packets. For each packet, the packet is optionally compressed, a keyed message authentication code is computed, added to the message with its sequence number, then the packet is encrypted and transmitted. TLS also allows for a number of control messages to be transmitted.

Analyzing the above 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 paper aims to quantify these costs.

3 TLS performance methodology

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 are happening simultaneously and the system's bottleneck is never intuitively obvious. Instead, we chose to measure the *throughput* of the 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 that would result from optimizing the system. In effect, we simulate *ideal* hardware accelerators. Based on these numbers, we can estimate the relative cost of each operation using Amdahl's Law (see Section 4).

3.1 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 a single 933MHz Pentium III. Both machines had 1GB of RAM and a gigabit Ethernet interface. Some experiments also included a Compaq AXL300 [3] cryptography acceleration board.Three generic 800MHz Athlon PCs with gigabit Ethernet cards served as TLS web clients, and all experiments were performed using a private gigabit Ethernet switch.

All computers ran RedHat Linux 6.2. The normal web servers used were Apache 1.3.14 [2], and the TLS web server was Apache with mod_SSL 2.7.1-1.3.14 [14]. The Apache mod_SSL solution was chosen due to its wide availability and use, as shown by a March 2001 survey [25]. The TLS implementation used in our experiments by mod_SSL is the open source OpenSSL 0.9.5a [18]. The web clients were derived from s-client as used by Banga *et al.* [4], with the addition of OpenSSL. To fully expose the performance limits of the server, we ensured that disk and network I/O were not limiting the achieved throughput in any of the experiments.

3.2 Experiments performed

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

One workload simulated the 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. We purchased two books at Amazon, one as a new user and one as a returning user. By replicating these two traces in the proportions that they are experienced by Amazon, we can simulate the load that a genuine Amazon secure server might experience. Our other workload was a 100,000-hit trace taken from our departmental web server, using a 530MB set of files. 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.

In order to measure the benefits to TLS performance from increasing CPU performance, we ran each workload against two server configurations: one with a 500MHz Pentium III and one with a 933MHz Pentium III. We also ran each of these systems with and without the AXL300 crypto accelerator to compare the benefits of a dedicated accelerator card versus general improvements in CPU speed.

All experiments were first performed once and the results thrown away, resulting in warm disk caches, thus minimizing potential bottlenecks that could result from disk I/O.

In addition to replacing cryptographic operations, such as RSA, RC4, MD5, and secure pseudo-random 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 (avoiding sharing costs in the shared memory cache). Simplifying further, we created a "skeleton TLS" protocol where all TLS operations have been completely removed but the messages of the same length as the TLS handshake are transmitted. This simulates an "infinitely fast" CPU that still needs to perform all 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 simulate the effects of "perfect" optimizations, identifying an upper bound on the benefits available from optimizing each component of the TLS system.

3.2.1 Amazon-like 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 back-end processing that occur in e-commerce sites, we can still accurately model the front-end 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.com, 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 normal 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 measure is the ratio of new to returning Amazon customers. Luckily, Amazon provided this ratio (78% returning customers to 22% new customers)

[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 simulation of "perfect" acceleration.

in a recent quarterly report [1]. For our experiments, we assume that returning customers do not retain TLS session state, and will thus complete the full TLS handshake. In this scenario, based on our traces, the server must perform a full TLS handshake approximately once out of every twelve web requests.

We created files on disk to match the sizes collected in our trace and request those files in the order they appear in the trace. When replaying the traces, each client process uses at most four simultaneous Web connections, just as common web browsers do. We also group together the hits corresponding to each complete web page (HTML files and inline images) and do not begin issuing requests for the subsequent page until the current page is completely loaded. All three client machine run 24 of these processes, each, causing the server to experience a load comparable to 72 web clients making simultaneous connections.

Our measurements are presented in Figure 1 and analysis of the data begins in Section 4.

3.2.2 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 hand-deleted 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.

In our trace playback system, three client machines ran 20 processes each, generating 60 simultaneous connects, 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 original trace in its 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. This allows the system to dispatch requests in a variety of different orders, but preserves the behavior of individual traces.

As a second constraint, we wished to enforce an upper bound on how far the final requests observed by the web server may differ from the order of requests in the original trace. If this bound were too small, it would artificially limit the concurrency that the trace playback system could exploit. If the bound were too large, there would be less assurance that the request ordering observed by the experimental server accurately reflected the original behavior captured in the trace. In practice, we needed to set this boundary at approximately 10% of the length of the original trace. Tighter boundaries created situations where the server was no longer saturated, and the clients could begin no new requests until some older large request, perhaps for a very large file, could complete.

While this technique does not model the four simultaneous connections performed by modern web browsers, it does saturate the server sufficiently that we believe the server throughput numbers would not change appreciably.

4 Analysis of experimental results

Figures 1 and 2 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 two bars, corresponding to the result obtained with the 500MHz system and the 933MHz system, respectively.

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-op alternatives, followed by configurations where each operation was enabled individually. We also measured a

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 pseudo-random 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

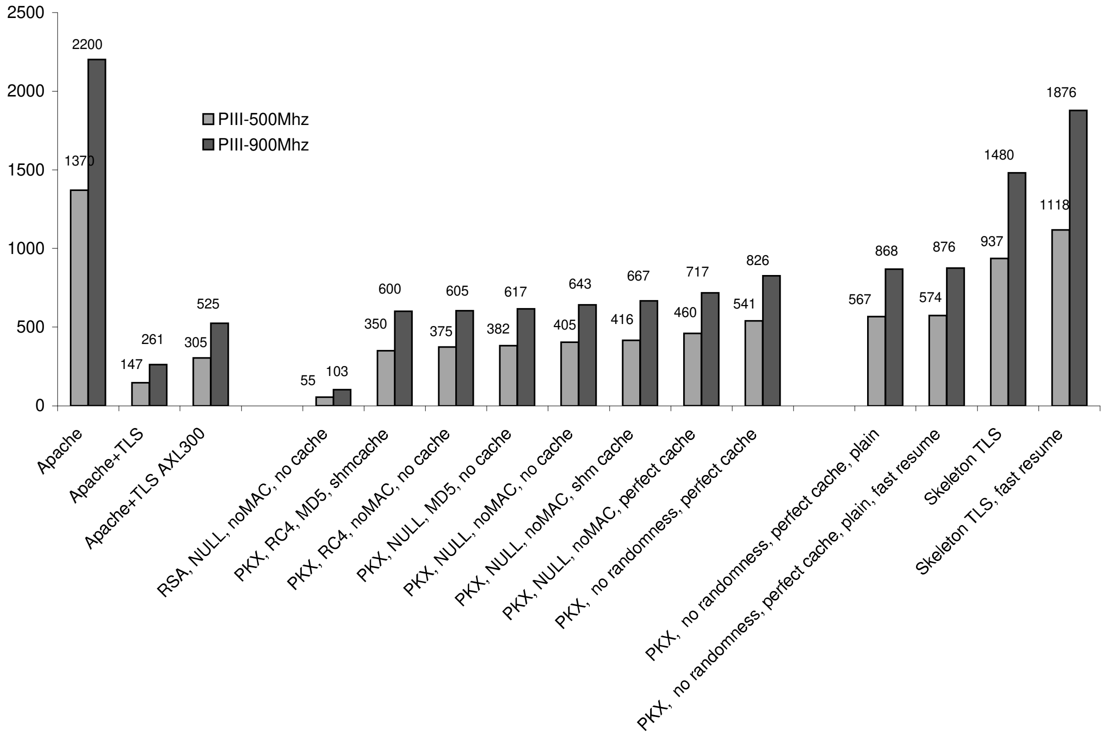

Figure 1: Throughput for Amazon trace and different server configurations, on 500MHz and 933MHz servers.

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 pseudo-random 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

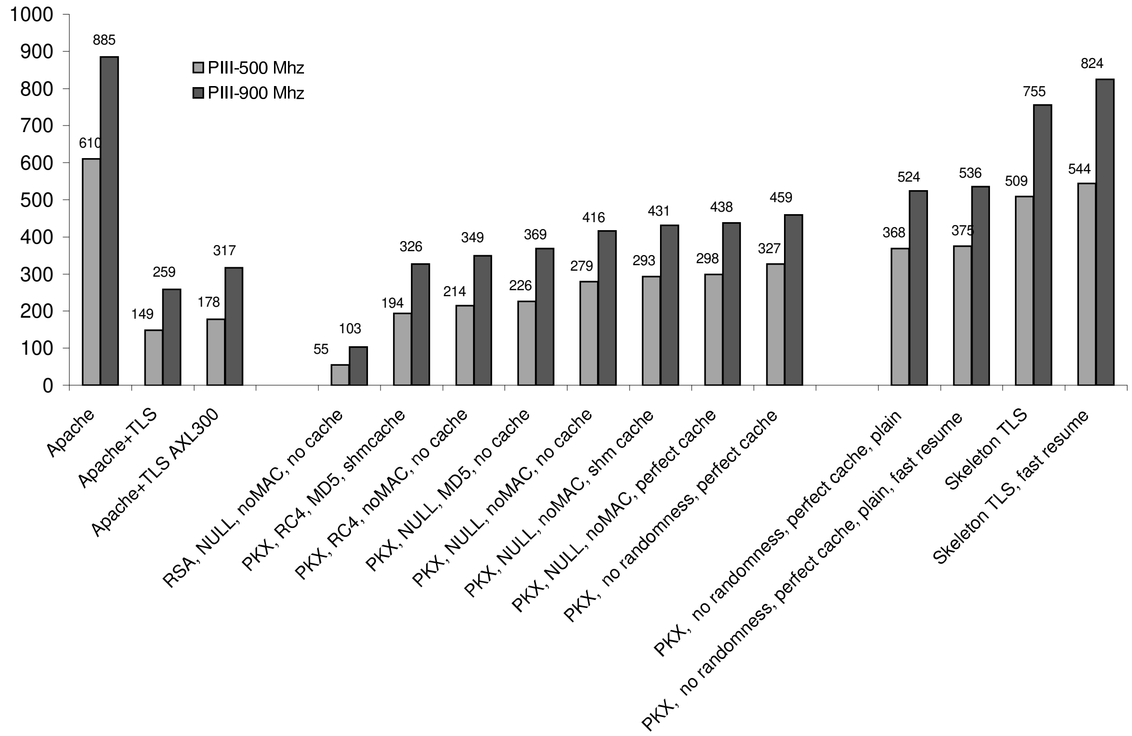

Figure 2: Throughput for CS trace and different server configurations, on 500MHz and 933MHz servers.

few additional configurations discussed below. 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, to expose the performance limits of RSA acceleration techniques.

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

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

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

For each TLS processing component, we have simulated infinite or almost infinite setup, 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 4.

In the following, we analyse the results of our measurements and calculations, and draw some broad conclusions.

4.1 Impact of TLS on server performance

The Apache server, without TLS enabled, achieves between 610 hits/sec and 885 hits/sec with the CS trace, and between 1370 hits/sec and 2200 hits/sec with the Amazon trace. The difference in throughput for the two workloads is due to the average file size: 46KB for the CS trace and only 7KB for the Amazon trace. Increasing the CPU speed from 500MHz to 933MHz leads to a substantial increase in throughput in each case.

Apache TLS without the AXL300 served between 149 hits/sec and 259 hits/sec for the CS trace, and between 147 hits/sec and 261 hits/sec for the Amazon trace. This confirms that TLS incurs a substantial cost and reduces the throughput by 70 to 89% relative to the insecure Apache. Apache TLS with the AXL300 served between 178 hits/sec and 317 hits/sec for the CS trace, and between 300 hits/sec and 525 hits/sec for the Amazon trace. This shows that, with the use of the AXL300 board, the throughput loss is now only 64 to 77% relative to the insecure Apache.

4.2 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 (45-60%) relative to the 500MHz Pentium III. 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.

4.3 Impact and effectiveness of accelerator hardware

The use of the AXL300 accelerator yields a noticeable throughput improvement with the CS trace (19 to 22%) relative to the normal TLS Apache, and a substantial gain with the Amazon trace (101 to 107%) 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, limiting 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, with an average file size of 7KB.

4.4 Comparative impact of accelerator hardware versus faster CPU

The next question we wish to pose is whether it is more advantageous to invest in acceleration hardware, or in a faster CPU. The answer depends strongly 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 only uses TLS for its final purchase validation and payment, will benefit from hardware RSA acceleration. Whereas, web sites that do all their user in-

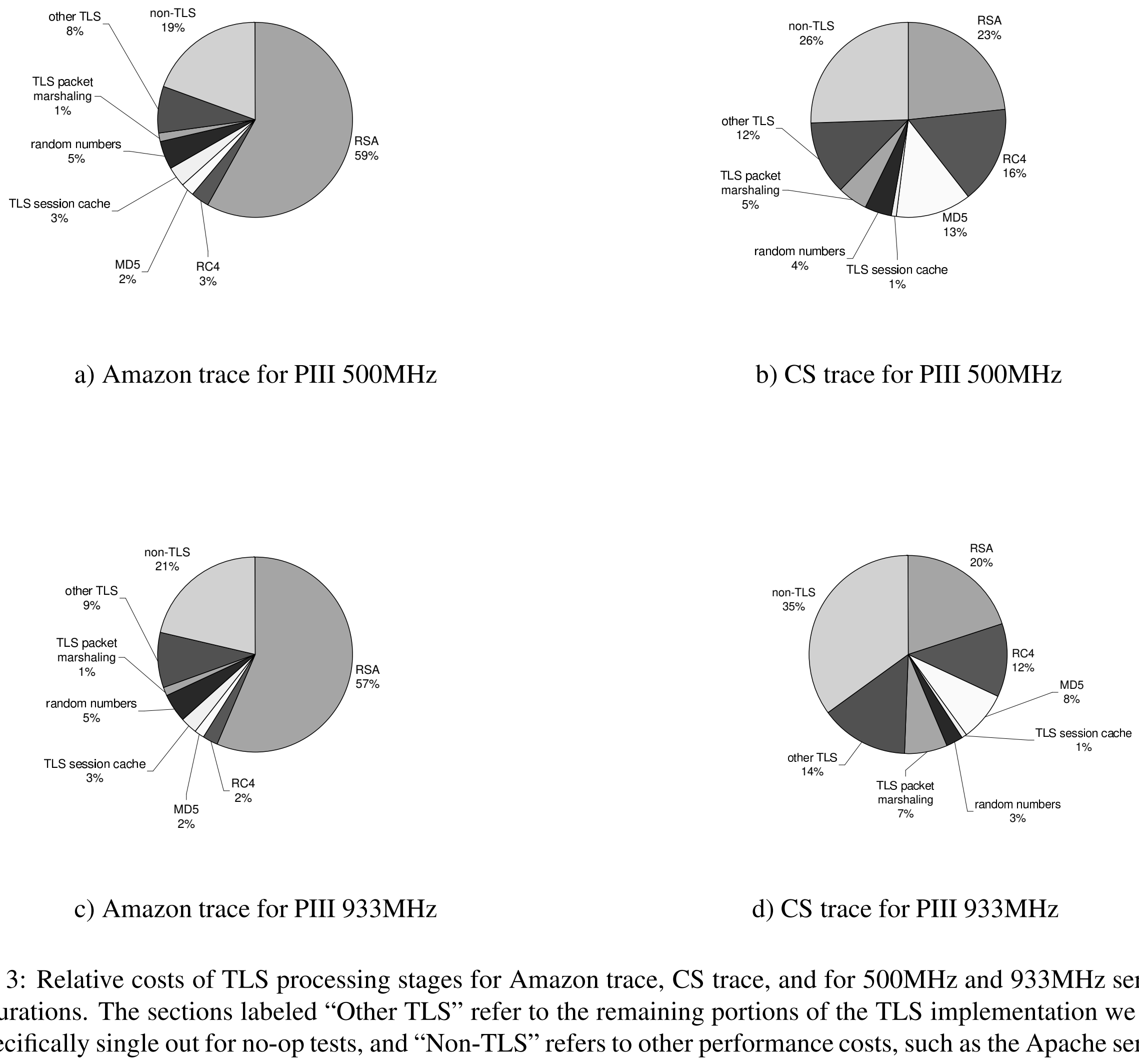

Figure 3: Relative costs of TLS processing stages for Amazon trace, CS trace, and for 500MHz and 933MHz server configurations. The sections labeled "Other TLS" refer to the remaining portions of the TLS implementation we did not specifically single out for no-op tests, and "Non-TLS" refers to other performance costs, such as the Apache server and the Linux kernel.

teraction through TLS may be better advised to invest in faster general purpose CPUs.

4.5 Impact of session caching

Our results confirm the findings of prior work that session caching substantially improves server throughput. The main reason for this gain is a reduction in the number of RSA operations as a result of session reuse. 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.

4.6 Relative cost and impact of crypto operations

Figures 1, 2, and 4 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.

4.7 Miscellaneous TLS operations

Starting with a server in which we replaced RSA, RC4, pseudo-randomness computations, 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 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. This difference, covering between 8% and 14% of the total performance cost, indicates a "catch all" of every other TLS cost, including the MD5 and SHA-1 hashes required as part of the TLS protocol setup.

Once the "other TLS" costs have been measured, the remainder must be from sources outside TLS, including the Apache web server and the Linux kernel.

4.8 Potential impact of protocol changes

In order to attempt to break down the "non-TLS" costs one step further, we finally explore the potential for performance improvement due to changes in the TLS protocol to reduce the number of message exchanges. The do this, we used a strawman 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 (1 to 2%).

4.9 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 (20% to 58%).
- Non-TLS performance costs (including Apache and the Linux kernel) range from 19 to 35% of the total cost.
- The costs of marshaling TLS data structures, computing connection keys from the premaster secret and executing other miscellaneous operations within TLS consumes between 8% and 14% of the total performance cost. Reducing the session resumption protocol by two messages and one round-trip delay indicates that that those messages and round-trip times only consumed 1% of the total cost.
- 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, the RSA accelerator can result in a 101-107% performance improvement (a factor of two improvement in hit rate). For high session reuse rates, however, the RSA accelerator only resulted in a 19-22% performance improvement.
- This improvement is bounded at 130-138% (for the Amazon trace) or 25-30% (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 2.5-2.7 for the CS trace and 2.3-2.5 for the Amazon trace, relative to a server with no cache.
- The costs of the other cryptographic operations performed by TLS are relatively low. 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. This implies that TLS performance enhancements must focus on reducing the

CPU cost of TLS and the underlying web server (or increasing the performance of the CPU).

5 Future work

This paper has studied the performance of TLS web service from a single server. It hasn't considered the larger environment that often occurs in an e-commerce site, such as load-balancing front end switches, with replicated clusters of web servers and a database back-end. A performance study of these more complex environments, with an eye to TLS scalability in the cluster would be interesting to perform. There have been some efforts in this direction already. For example, the Zeus Web server performance tuning guide [26] mentions the importance of sharing TLS sessions across Web servers in a cluster. We would like to study the interaction of different cluster load-balancing strategies (such as used in LARD [19]) with TLS web servers.

This paper 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 [23] or RSIM [22]. 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 [20] 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.

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 20-58% of the time spent in the web server. 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 ideal RSA optimization leaves a large gulf between TLS server performance and insecure server performance. No simple optimization is likely to close that gap, but faster CPUs can be expected to narrow it over time.

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 e-commerce sites that minimize requests to secure servers and therefore experience relatively lower session reuse rates. Acceleration appears to be less effective for sites where all requests are handled by a TLS server, thus having higher session reuse rates. For such sites, investing in a faster CPU may prove more effective.

7 Acknowledgements

We gratefully acknowledge Compaq for loaning us the hardware used in performing our experiments. 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.

References

[1] Amazon.Com releases 2001 first quarter results. Press Release, Apr. 2001. `http://www.sec.gov/Archives/edgar/data/1018724/000095010901500823/dex99%1.htm`.

[2] Apache. http://www.apache.org/.

[3] The AXL300 RSA accelerator. `http://www.compaq.com/products/servers/security/axl300/`.

[4] BANGA, G., AND DRUSCHEL, P. Measuring the capacity of a Web server under realistic loads. *World Wide Web Journal (Special Issue on World Wide Web Characterization and Performance Evaluation)* (1999).

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

[6] BANGA, G., MOGUL, J. C., AND DRUSCHEL, P. A scalable and explicit event delivery mechanism for UNIX. In *Proceeding of the Usenix 1999 Annual Technical Conference* (Monterey, California, June 1999).

[7] BONEH, D., AND SHACHAM, H. Improving SSL handshake performance via batching. In *Proceedings of the RSA Conference* (San Francisco, California, Apr. 2001).

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

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

[10] GOLDBERG, A., BUFF, R., AND SCHMITT, A. Secure web server performance dramatically improved by caching SSL session keys. In *Proceedings of the Workshop on Internet Server Performance* (Madison, Wisconsin, June 1998).

[11] HU, J. C., PYRALI, I., AND SCHMIDT, D. C. Measuring the impact of event dispatching and concurrency models on web server performance over high-speed networks. In *Proc. 2nd Global Internet Conf.* (Nov. 1997).

[12] KAASHOEK, M. F., ENGLER, D. R., GANGER, G. R., AND WALLACH, D. A. Server Operating Systems. In *Proceedings of the 1996 ACM SIGOPS European Workshop* (Connemara, Ireland, Sept. 1996), pp. 141–148.

[13] KELSEY, J., SCHNEIER, B., WAGNER, D., AND HALL, C. Cryptanalytic attacks on pseudorandom number generators. In *Proceedings of Fifth International Workshop on Fast Software Encryption* (Paris, Mar. 1998).

[14] modSSL. `http://www.modSSL.org/`.

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

[16] NAHUM, E. M., BARZILAI, T., AND KANDLUR, D. Performance issues in WWW servers. *IEEE/ACM Transactions on Networking* (2001). to appear.

[17] NAHUM, E. M., ROSU, M., SESHAN, S., AND ALMEIDA, J. 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, Massachusetts, June 2001).

[18] OpenSSL. `http://www.OpenSSL.org/`.

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

[20] PAI, V. S., DRUSCHEL, P., AND ZWAENEPOEL, W. Flash: An efficient and portable Web server. In *Proceeding of the Usenix 1999 Annual Technical Conference* (Monterey, California, June 1999), pp. 199–212.

[21] PAI, V. S., DRUSCHEL, P., AND ZWAENEPOEL, W. I/O-Lite: A unified I/O buffering and caching system. In *Proc. 3rd USENIX Symp. on Operating Systems Design and Implementation* (New Orleans, Louisiana, Feb. 1999).

[22] PAI, V. S., RANGANATHAN, P., AND ADVE, S. V. RSIM: An execution-driven simulator for ILP-based shared-memory multiprocessors and uniprocessors. In *Proceedings of the Third Workshop on Computer Architecture Education* (Feb. 1997). Also appears in IEEE TCCA Newsletter, October 1997.

[23] ROSENBLUM, M., BUGNION, E., DEVINE, S., AND HERROD, S. Using the SimOS machine simulator to study complex computer systems. *ACM TOMACS Special Issue on Computer Simulation* (1997).

[24] SCHNEIER, B. *Applied Cryptography*, 2nd ed. John Wiley and Sons, New York, New York, 1996.

[25] The Netcraft Secure Server Survey. `http://www.netcraft.com/ssl/`.

[26] Zeus performance tuning guide. `http://support.zeus.com/faq/entries/ssl_tuning.html`.