RC22563 (W0209-065) September 17, 2002
Computer Science

IBM Research Report

Performance Considerations in Web Security

Arun K. Iyengar, Ronald Mraz
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Mary Ellen Zurko
IBM Software Group
5 Technology Park Drive
Westford Technology Park
Westford, MA 01886

PERFORMANCE CONSIDERATIONS IN WEB SECURITY

Arun Iyengar
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
aruni@us.ibm.com

Ronald Mraz
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
mraz@us.ibm.com

Mary Ellen Zurko
IBM Software Group
5 Technology Park Drive
Westford Technology Park
Westford, MA 01886
mzurko@us.ibm.com

Abstract

This paper discusses techniques for improving Web performance and how they are affected by security. While security is an essential component for many Web applications, it can negatively affect performance. Encryption results in significant overhead. A scalable Web site deploying SSL has special load balancing requirements in order to allow efficient use of the protocol. We discuss how fragment-based creation of Web content can allow partial caching of pages containing encrypted content. We also discuss performance issues related to security checks on mobile code.

Keywords: cryptography, load balancing, mobile code, Secure Sockets Layer (SSL), Transport Layer Security (TLS), security, Web performance

Introduction

Performance is critically important for any Web site which receives a significant amount of traffic. Highly accessed Web sites may need to serve over a million hits per minute. The infrastructure required to support such traffic is significant, and demands continue to increase at a rapid rate.

A key problem with serving Web data efficiently is that the data are often encrypted or dynamically created. Encrypted data is costly to serve. A major overhead is negotiating the session keys used for encryption. Dynamically created data is generated on-the-fly by a program which executes at the time a request is made. The overhead for satisfying a dynamic data request may be orders of magnitude more that the overhead for satisfying a static request for data in a file system. Dynamic data requests can involve complicated back-end processing involving database accesses.

Scalable Web sites typically deploy multiple Web servers and route requests to the servers via load balancers or Domain Name Servers. Such load balancing techniques can prevent repeated requests from the same client going to the same server. If requests are encrypted using the Secure Sockets Layer protocol (SSL) or Transport Layer Security (TLS) [17, 6], these load balancing techniques can cause significant overhead due to more frequent generation of encryption keys. We discuss affinity-based load balancing techniques which can prevent this overhead.

Caching is a critical component for improving performance. Caches are deployed throughout the Web storing content from remote sites. Unfortunately, caches cannot store confidential data. Confidential data would typically be fetched from the remote server each time. Using the conventional approach, a Web page containing confidential data would be encrypted in its entirety and fetched from the server each time. Encryption would thus eliminate the performance gains provided by remote caching of Web pages.

One approach which can allow caching in the presence of encryption is to assemble a Web page from fragments. Confidential information would be encapsulated within fragments. Nonconfidential fragments could be stored within caches. A client would then fetch confidential encrypted fragments from the server, nonconfidential fragments from remote caches, and assemble the page. Such a system would require appropriate communication protocols between clients, servers, and caches.

Security and performance are both quality measures of a software system. If a product is not initially designed to be secure and perform well, security and performance may be difficult to engineer in after the product is architected, designed, coded and tested. Most areas of security do not explicitly address performance or other quality implications, to the overall detriment of the uptake of the potentially beneficial security solution. Cryptography has long been the exception that proves the rule, since the performance impact of cryptography can be easily isolated and measured. More recently, the security benefits from the safety guarantees of mobile code such as Java have been well received, with the commensurate push to increase the performance of applications using mobile code. We discuss the current work on security performance in those two areas.

1. Factors Affecting Web Performance

Popular Web sites need to accommodate high request rates. Peak hit rates at popular Web sites can exceed 1 million per minute. Load at a Web site can be highly variable depending upon the time of day. The projected peak load should be taken into account when doing capacity planning and not just the average load.

Another thing which needs to be taken into consideration is that capacity requirements can change over time. A Web site can become more popular as a company expands or gains more on-line customers, significantly increasing the traffic at the site. The general growth in Web traffic over time also adds to capacity requirements. Some sites may experience flash crowds in which one or more events cause huge increases in request rates for a limited period of time.

Requests can consume widely differing amounts of resources to satisfy. If I/O bandwidth is the bottleneck, then large objects become undesirable to serve. Image files can consume significant I/O bandwidth, so limiting the use of images can improve performance considerably. Requests for files are known as static requests and generally consume less overhead than dynamic requests which invoke programs to generate data on-the-fly for satisfying requests. Requests for dynamic data can consume orders of magnitude more CPU time to satisfy than requests for static data. Therefore, even if a Web site serves only a fraction of its requests dynamically, dynamic requests can consume the bulk of the CPU cycles.

Encryption can also add significant overhead to a Web site. Encryption is typically handled on the Web using the Secure Sockets Layer (SSL) or Transport Layer Security (TLS). The SSL protocol requires a handshake at the beginning in order for the client and server to negotiate a session key used for encrypting data via symmetrical cryptography. Session key generation is expensive. The overhead of session key generation is reduced by using the same session key for multiple transactions. In order to limit security exposure, session keys have a limited lifetime after which they must be changed.

1.1 Scalable Web Sites

In a scalable Web site, requests are distributed to multiple servers by a load balancer. The Web servers may access one or more databases or other back-end systems for creating content. The Web servers would typically contain replicated content so that a request could be directed to any server in the cluster. One way to share static files across multiple servers is to use a distributed file system such as AFS or DFS [12]. Copies of files may be cached in servers for faster access. This approach works if the number of Web servers is not too large and data doesn't change frequently. For large numbers of servers for which data updates are frequent, distributed file systems can be highly inefficient. Part of the reason for this is the strong consistency model imposed by distributed file systems. Shared file systems require copies of files to be strongly consistent. In order to update a file in one server, all other copies of the file need to be invalidated before the update can take place. These invalidation messages add overhead and latency. At some Web sites, the number of objects updated in temporal proximity to each other can be quite large. During periods of peak updates, the system might fail to perform adequately.

Another method of distributing content which avoids some of the problems of distributed file systems is to propagate updates to servers without requiring the strict consistency guarantees of distributed file systems. Using this approach, updates are propagated to servers without first invalidating all existing copies. This means that at the time an update is made, data may be inconsistent between servers for a little while. For many Web sites, these inconsistencies are not a problem, and the performance benefits from relaxing the consistency requirements can be significant.

1.2 Load Balancing

Load balancers distribute requests among multiple Web servers. One method of load balancing requests to servers is via DNS servers. DNS servers provide clients with the IP address of one of the site's content delivery nodes. When a request is made to a Web site such as `http://www.ibm.com/employment/`, "www.ibm.com" must be translated to an IP address, and DNS servers perform this translation. A name affiliated with a Web site can map to multiple IP addresses, each associated with a different Web server. DNS servers can select one of these servers using a policy such as round robin [2].

One of the problems with load balancing using DNS is that name-to-IP mappings resulting from a DNS lookup may be cached anywhere along the path between a client and a server. This can cause load imbalance because client requests can then bypass the DNS server entirely and go directly to a server [5]. Name-to-IP address mappings have time-to-live attributes (TTL) associated with them which indicate when they are no longer valid. Small TTL values can limit load imbalances due to caching. The problem with this approach is that it can increase response times [19]. Another problem with this approach is that not all entities caching name-to-IP address mappings obey TTL's which are too short.

Another approach to load balancing is using a connection router in front of several back-end servers. Connection routers hide the IP addresses of the back-end servers. That way, IP addresses of individual servers won't be cached, eliminating the problem experienced with DNS load balancing. Connection routing can be used in combination with DNS routing for handling large numbers of requests. A DNS server can route requests to multiple connection routers. The DNS server provides coarse grained load balancing, while the connection routers provide finer grained load balancing. Connection routers also simplify the management of a Web site because back-end servers can be added and removed transparently.

IBM's Network Dispatcher [8] is one example of a connection router which hides the IP address of back-end servers. Network Dispatcher uses Weighted Round Robin for load balancing requests. Using this algorithm, servers are assigned weights. All servers with the same weight receive a new connection before any server with a lesser weight receives a new connection. Servers with higher weights get more connections than those with lower weights, and servers with equal weights get an equal distribution of new connections.

With Network Dispatcher, requests from the back-end servers go directly back to the client. This reduces overhead at the connection router. By contrast, some connection routers function as proxies between the client and server in which all responses from servers go through the connection router to clients.

1.3 Caching

One technique for improving Web performance is to cache data at remote points in the network. If a request can be satisfied from a cache, this reduces load on the server and can also reduce the latency for fetching objects since remote caches can be placed closer to clients. Several clients can share a proxy cache. That way, repeated requests for the same document from different clients might be satisfiable from a proxy cache. In addition to proxy caches, content distribution networks (CDN's) such as Akamai exist which cache content from Web sites. Web sites pay a fee to use CDN's.

In order to reduce the overhead for generating dynamic data, it is often feasible to generate data corresponding to a dynamic object once, store the object in a cache, and subsequently serve requests to the object from cache instead of invoking the server program again [10]. Using this approach, dynamic data can be served at about the same rate as static data.

However, there are types of dynamic data that cannot be precomputed and served from a cache. For instance, dynamic requests that cause a side effect at the server such as a database update cannot be satisfied merely by returning a cached page. As an example, consider a Web site that allows clients to purchase items using credit cards. At the point at which a client commits to buying something, that information has to be recorded at the Web site; the request cannot be solely serviced from a cache.

Personalized Web pages can also negatively affect the cacheability of dynamic pages. A personalized Web page contains content specific to a client, such as the client's name. Such a Web page could not be used for another client. Therefore, caching the page is of limited utility since only a single client can use it. Each client would need a different version of the page.

One method which can reduce the overhead for generating dynamic pages and enable caching of some parts of personalized pages is to define these pages as being composed of multiple fragments [3]. In this approach, a complex Web page is constructed from several simpler fragments. A fragment may recursively embed other fragments. This is efficient because the overhead for assembling a Web page from simpler fragments is usually minor compared to the overhead for constructing the page from scratch, which can be quite high.

The fragment-based approach also makes it easier to design Web sites. Common information that needs to be included on multiple Web pages can be created as a fragment. In order to change the information on all pages, only the fragment needs to be changed.

In order to use fragments to allow partial caching of personalized pages, the personalized information on a Web page is encapsulated by one or more fragments that are not cacheable, but the other fragments in the page are. When serving a request, a cache composes pages from its constituent fragments, many of which are locally available. Only personalized fragments have to be created by the server. As personalized fragments typically constitute a small fraction of the entire page, generating only them would require lower overhead than generating all of the fragments in the page.

Generating Web pages from fragments provides other benefits as well. Fragments can be constructed to represent entities that have similar lifetimes. When a particular fragment changes but the rest of the Web page stays the same, only the fragment needs to be invalidated or updated in the cache, not the entire page. Fragments can also reduce the amount of cache space taken up by multiple pages with common content. Suppose that a particular fragment is contained in 2000 popular Web pages which should be cached. Using the conventional approach, the cache would contain a separate version of the fragment for each page resulting in as many as 2000 copies. By contrast, if the fragment-based method of page composition is used, only a single copy of the fragment needs to be maintained.

2. Effects of Encryption on Performance

Encryption is essential for preserving confidentiality of information sent via the Web. The HTTP protocol used for Web traffic sends information in the clear. It is not difficult for someone to monitor Web traffic and obtain information exchanged via HTTP.

The Secure Sockets Layer (SSL) is the protocol commonly used for encrypting information on the Web. SSL was designed by Netscape Communications Corporation for use with the Netscape Navigator. SSL 3.0 was used as the basis for the Transport Layer Security (TLS) protocol [6] developed by the Internet Engineering Task Force (IETF). In addition to confidentiality, SSL also provides authentication of servers using digital signatures. It can also provide authentication of clients using digital signatures, although this feature is often not used.

SSL runs between the TCP/IP layer and the application layer. Although it was designed to run on top of TCP/IP, it can also run on top of other reliable connection-oriented protocols such as X.25 or OSI. It is not designed to run on top of unreliable protocols such as the IP User Datagram (UDP). Other protocols can use SSL besides HTTP, such as SMTP, Usenet news, LDAP, and POP3. HTTP traffic encrypted via SSL uses port 443. [17]

Before transmitting data, a phase known as a handshake takes place in which the client and server agree on a cryptographic algorithm and exchange keys. The client and server may use different cryptographic algorithms. During the handshake, the client and server determine the strongest cryptographic protocol they have in common for encrypting information.

The SSL handshake proceeds in the following fashion. The client first sends a hello message to the server. The server then responds to the client with its own hello message followed by a certificate that contains the server's public key. The client then verifies the certificate of the server, and if the server is valid, the client generates a premaster secret. The premaster secret is encrypted using the server's public key and sent to the server. The server decrypts the premaster secret using its private key. The premaster secret is used to generate a master secret from which encryption and authentication keys are derived.

Public key cryptography is computationally expensive. Therefore, it isn't feasible for the client and server to encrypt all of their communications via public key cryptography. Instead, public key cryptography is only used to agree on *session keys* which are used for encrypting the bulk of the content using symmetric key encryption algorithms such as DES, triple-DES, RC2, RC4, etc. [18, 7].

In order to reduce the overhead of encryption, some sites use special hardware for performing cryptographic operations in order to offload the compute-intensive functions. Sites with predominately static content to serve can benefit most from offloading encryption since the overhead to support the SSL protocol and encryption in the server would typically exceed the overhead of static content serving. Sites such as this could be selling subscriptions to content that is replicated but protected through encryption. These server applications could include subscription magazine articles, newsletters, etc. and in such Web content servers, the overhead of encryption predominates. (We address the issues of serving secure personalized content later.)

The most common ways to offload SSL operations include: adding encryption hardware to the server, placing one or more SSL proxy(s) between the router (or load balancer) and the Web content server, or placing one or more SSL proxy(s) with hardware encryption between the router (or load balancer) and Web content server.

If there are relatively few unique connections but each connection requires large amounts of data, then a data encryption card such as the IBM 4197 Cryptographic Accelerator [9] within the Web content server can be useful if the server utilization is high. We define this case to be when the server provides 10 to 100 or more objects (or transactions) per negotiated SSL session key. [14]

Offloading SSL to a proxy server is desirable since this eliminates the data encryption, public key overhead and SSL handshake operations previously handled by the Web content server. An SSL offload proxy can be a card that is installed in the server and ties into the TCP/IP communication stack in place of a network communication card. This type of offload is most useful when there are a high number of public key exchanges for the server to support relative to the number of objects served. We define a high number of public key exchange operations to be when the number of public key operations is within an order of magnitude of the objects (or transactions) served per session when the server is highly utilized. Multiple proxies (hardware assisted or otherwise) can provide scalability [14] if the content server is underutilized without resident SSL operations.

When architecting or crafting a custom SSL proxy, overhead due to TCP/IP connection establishment can be reduced if the connections between the proxy and the Web content server are persistent. Connection establishment can add significant overhead to data connections [13], and this would be successfully offloaded to the proxy in addition to SSL processing. Advanced architectures such as [15] can scale individual components of SSL operations, such as the handshake, encryption, connection establishment and content serving rather than multiple operations within a single server or proxy.

Efficiency can also depend upon the rate that unique SSL sessions are started versus the session duration. This is because the public key operations are the most computationally expensive part of the handshake. The decryption operations performed on the server, which are done using the server's private key, are particularly expensive [1]. Because of this overhead, it is desirable for performance reasons to re-use session keys over several transactions.

Once way to increase the number of times a session key is reused would be to configure each negotiated session key to have an infinite lifetime, assuming that at least some requests, over time, are from repeat users. Given that such an arrangement would impose a security risk if the session key were ever determined by a malicious party, session keys have a finite lifetime after which they are renegotiated. The lifetime is typically set long enough (on the order of 100-300 seconds) for several transactions to take place between a client and a server using the same session key if the transactions occur within close temporal proximity to each other.

Key lifetime duration for a reasonable key length can actually be set to several hours without risk of security breaches. This will enable an intermittent user's request to re-use the previous session key when reconnection is done after several minutes or even hours. To make this approach effective, one needs to consider how many keys the server can efficiently store and search versus the number of users that actually request content in an intermittent fashion over an extended period of time. This is because every newly created session key during this lifetime period should be saved for potential reuse.

When the content is predominately personalized as in "shopping carts" or "on-line brokerage" Web sites, the content generation overhead can be more expensive than encryption if advanced caching techniques described in this paper are not employed. This is because each request requires a personalized database lookup (or perhaps a real-time request for data) that is subsequently formatted into a personalized table of custom HTML generated at request time. Since the encryption overhead no longer dominates, adding encryption hardware to each Web content server is not efficient use of the hardware, and gains of only 10% or less may be seen from such an arrangement. In this case, an effective use of SSL offload would be for a single SSL proxy (with encryption hardware) to support several Web content servers with some type of load balancing affinity as discussed later in this section.

Deciding on the cryptographic strength of a particular use of confidential enciphering through cryptography is particularly sensitive to the performance assumptions we can make about the security's attacker. Before computers, cryptanalysis (breaking ciphers) was limited by the skill, experience, and ingenuity of its practitioners. The introduction of computers added a new tool to the discipline: the brute force attack of trying to decrypt a piece of enciphered text by applying every possible decryption key to the decryption algorithm until one works. As computer processing of mathematical operations gets faster, the search space for a brute force attack must get larger for that attack to be ineffective.

First, we want to consider how long a key length is “long enough”. A very short key of 8 bits in length would mean that there are 256 different possible values, which a brute force attack on a modest computer could find quite quickly. In addition, on average, there would be a 50% chance the brute force attack would find the key in half of those attempts. A key that is 128 bits long would take a supercomputer that can check a million keys a second 10^{25} years to check all keys, which is more than twice the age of the universe. That seems like a good starting point for “long enough”. The current NIST-sponsored Advanced Encryption Standard (AES) in symmetric key cryptography is based on Rijndael, which supports keys of 128, 192, and 256 bits. It’s hard to see why someone would take the performance hit of using a key longer than 128 bits, given the mathematics of a pure brute force attack. Those key lengths may be supported because it is possible to combine brute force attacks with algorithms that help narrow the search space more quickly, or with newly found flaws in existing algorithms. Schneier [18] presents a good discussion of the many security factors that go into choosing key lengths based on the desired length of time something must stay encrypted and the presumed future advances in computer processing power. It is difficult to make accurate assumptions about future advances in cryptanalysis, which is why a margin of safety is desirable.

The need to think in detail about choosing a key length is motivated by the desire to minimize the impact cryptographic operations have on the overall performance and cost of the systems that require it. The impact of key length on both encryption and brute force attack is not proportional [11]. Increasing the number of bits in a key by n slows the encryption speed by $O(n)$ while it slows the brute force attack by $O(2^n)$. So, all things being equal, faster computers favor cryptographers (often called “good guys”) over cryptanalysts (often called “bad guys”, since they stand in for the attackers).

System configurations can alter the overall performance characteristics of this relationship. Client-side encryption and decryption distributes the performance impact of cryptography across client machines when application-level end-to-end protection is used. The use of S/MIME to protect email content is an example of this. However, pushing previously traditional client-side operations into the server increases the overall burden on the server. If the application content cannot be protected directly in an end-to-end fashion, it can be instead transferred over a protected channel, such as SSL or IPsec, which forces the server to perform cryptographic operations that it might otherwise not need to.

SSL presents problems for load balancing. When a Web site contains many servers, the load balancer is likely to send requests from the same client to a variety of different servers. If the requests are encrypted using SSL, each of the servers that the client communicates with will have to generate its own session keys, and this will result in significantly more overhead than if a single server generates session keys.

IBM's Network Dispatcher has special features for handling client affinity to selected servers which are critical for improving performance when SSL is being used. Network dispatcher recognizes SSL requests by the port number (443). It allows certain ports to be designated as "sticky". Network Dispatcher keeps records of old connections on such ports for a designated affinity life span (e.g. 100 seconds for SSL). If a request for a new connection from the same client on the same port arrives before the affinity life span for the previous connection expires, the new connection is sent to the same server that the old connection utilized.

Using this approach, SSL requests from the same client will go to the same server for the lifetime of a session key, obviating the need to negotiate new session keys for each SSL request. This can cause some load imbalance, particularly since the client address seen by Network Dispatcher may actually be a proxy representing several clients and not just the client corresponding to the SSL request. However, the reduction in overhead due to reduced session key generation is usually worth the load imbalance created.

Many Web sites make gratuitous use of SSL. For example, some sites will encrypt all of the image files associated with an HTML page and not just the HTML page itself. This is often unnecessary. The image files might contain content which can safely be passed in the clear. A reason why this problem occurs is that the image tags specifying the images to be embedded within an HTML page are often specified as relative links to a base location and do not include the protocol. When this is done, the browser assumes that the protocol is the same as that of the document including the image. Therefore, if an image included within an HTML document can be sent in the clear, the image tag should explicitly specify the HTTP protocol as opposed to the HTTPS protocol for SSL.

Conventional caching techniques cannot be used for confidential documents. It would involve too much of a security risk to let a third party cache have access to confidential information. Storing encrypted data within a cache would also be problematic because there would have to be some way for a client obtaining the encrypted data to obtain the keys to decrypt the data. A large repository of encrypted data at a cache would also present a target for a malicious hacker to try to steal data and decrypt somehow.

In order to allow caching of at least some parts of confidential documents, the fragment-based techniques described in Section 1.3 can be used. Web pages are composed using fragments in which the confidential parts of Web pages are encapsulated in confidential fragments and other parts of the Web page which may be shared are encapsulated in shared fragments. Shared fragments may be cached. Confidential fragments are encrypted by the server and decrypted by the client. The client reconstructs complete Web pages from the various fragments. If a significant amount of HTML text in a Web page can be shared and passed in the clear, then this method of dealing with fragments can result in significant performance gains.

A problem with deploying this method on the Web today is the lack of standard protocols for fragment-based assembly of Web pages by clients. This method could be implemented and deployed in a proprietary system.

Recent work on security for high-performance computing [4] targets creating low overhead security for a particular set of application assumptions. The data for control flow of high-performance applications has only short term value, and therefore needs a shorter cover time than document-based application data, which is meant for human consumption. Buffer copies are a major source of overhead to avoid. To eliminate buffer copies, the encryption techniques of transposition, substitution and data padding while the message is being marshalled onto the wire may be integrated. The prototype in [4] performs a Diffie-Hellman key exchange at connection setup time. This key is used to perform simple dynamic substitution of method identifiers, SHUFFLE transpositions in the data marshalling order, and data padding on the original message. Performance evaluations indicate less than 10% additional overhead on the messages and less than 25 microseconds to obtain all of the necessary random numbers. From the security point of view, more work will be needed to determine both how long the protected data needs to stay protected, and how long it is protected with their techniques.

3. Mobile Code

In the case of mobile code systems, such as Java, part of the job of the security sub-system is to protect the host machine from malicious code. In Java's case, the bytecode verifier of the Java virtual machine is invoked to attempt to prove that a given series of Java bytecodes represents a legal set of Java instructions. This ensures that no illegal data conversion or casting occurs, which might provide illegal access to member variables or other data. It also ensures that no operand stack overflows or underflows occur. Stack overflows can be used to execute malicious instructions on the target machine, using data as code. Proving the legality of Java bytecode can be an expensive operation. Java 2 Platform, Micro Edition (J2ME) does a preverification of the Java code, so that the runtime verifier just needs to verify the proof instead of generate it, reducing the footprint from 10KB to about 200 bytes.

Other mobile code protection systems use verification of a proof to ensure more extensive security properties. Proof-Carrying Code (PCC) [16] is a technique by which the host establishes a set of safety rules that guarantee safe behavior of programs. The code producer is responsible for creating a formal safety proof that proves that untrusted code adheres to the host's safety rules. The costly generation of the proof is distributed to the less performance critical code production time, while the less costly checking of the proof (and ensuring the code adheres to the presented proof) is done at run time. The checking does not rely on cryptographic techniques or input from an external trusted entity. The proofs and checks are based on logic, type theory, and formal verification. A pragmatic difficulty with using PCC is automatically generating the required proofs. In one experiment [16], Necula and Lee implemented several network packet filters in DEC Alpha assembly language, and used a special prototype assembler to create PCC binaries for them. They checked the proofs with a validator they implemented, which loads the code if it passes the check. The cost of loading and checking the validity of the proofs generated was between 1 and 3 milliseconds. In performance tests, they found the cost of validation was balanced by the run-time performance benefits. In their examples, the cost was amortized after 1200 packets when compared to a BSD Packet Filter (BPF) architecture, and 28,000 packets when compared to a Software Fault Isolation (SFI) architecture. The BPF approach made simple static checks on reading and writing. The SFI approach parses binaries and inserts run-time checks on memory operations. A difficulty not addressed by this work is determining the range of host safety rules that can be supported, and determining which sets of safety rules are both useful and provide some reasonably coherent safety model or properties.

4. Conclusion

Security can have a significant effect on performance. Encryption adds significant overhead to Web performance. We discussed various techniques for reducing these overheads as well as the effects of encryption on load balancing and caching.

In closing, it should be noted that the performance gains which can be achieved by optimizing encryption are dependent on the application. If the application is serving only static data, then encryption may constitute the major bottleneck to the system. Significant throughput improvements may be possible by optimizing encryption techniques. For applications serving significant quantities of dynamic data, the performance bottleneck may be the overhead for generating the dynamic data. Optimizing encryption techniques will in this case only result in limited throughput improvements.

References

[1] G. Apostolopoulos, V. Peris, and D. Saha. Transport Layer Security: How much does it really cost? In *Proceedings of IEEE INFOCOM'99*, March 1999.

[2] T. Brisco. DNS Support for Load Balancing. Technical Report RFC 1974, Rutgers University, April 1995.

[3] J. Challenger, A. Iyengar, K. Witting, C. Ferstat, and P. Reed. A Publishing System for Efficiently Creating Dynamic Web Content. In *Proceedings of IEEE INFOCOM 2000*, March 2000.

[4] K. Connelly and A. Chien. Breaking the Barriers: High Performance Security for High Performance Computing. In *New Security Paradigms Workshop*, 2002.

[5] D. Dias, W. Kish, R. Mukherjee, and R. Tewari. A Scalable and Highly Available Web Server. In *Proceedings of the 1996 IEEE Computer Conference (COMPCON)*, February 1996.

[6] T. Dierksand and C. Allen. The TLS Protocol (RFC 2246). http://www.ietf.org/rfc/.

[7] S. Garfinkel and G. Spafford. *Web Security, Privacy, and Commerce*. O'Reilly & Associates, second edition, 2002.

[8] G. Hunt, G. Goldszmidt, R. King, and R. Mukherjee. Network Dispatcher: A Connection Router for Scalable Internet Services. In *Proceedings of the 7th International World Wide Web Conference*, April 1998.

[9] IBM Corporation. IBM 4197 Cryptographic Accelerator. http://www.ibm.com/, 2000.

[10] A. Iyengar and J. Challenger. Improving Web Server Performance by Caching Dynamic Data. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

[11] C. Kaufman, R. Perlman, and M. Speciner. *Network Security, Private Communication in a Public World*. Prentice-Hall, 2002.

[12] T. T. Kwan, R. E. McGrath, and D. A. Reed. NCSA's World Wide Web Server: Design and Performance. *IEEE Computer*, 28(11):68–74, November 1995.

[13] J. Mogul. The case for persistent-connection HTTP. In *Proceedings of SIGCOMM '95*, pages 299–313, 1995.

[14] R. Mraz, K. Witting, and P. Dantzig. Using SSL Session ID Reuse for Characterization of Scalable Secure Web Servers. Technical Report RC 22323(Revised May 5, 2002), IBM Research Division, Yorktown Heights, NY, September 2002.

[15] Ronald Mraz. Secure Blue: An Architecture for a High Volume SSL Interent Server. In *17th Annual Computer Security Applications Conference, December 2001, New Orleans, Louisiana*, 2001.

[16] G. Necula and P. Lee. Safe Kernel Extensions Without Run-Time Checking. In *Proceedings of OSDI '96*, October 1996.

[17] E. Resorla. HTTP Over TLS (RFC 2818). http://www.ietf.org/rfc/.

[18] B. Schneier. *Applied Cryptography*. John Wiley & Sons, Inc., New York, NY, 1996.

[19] A. Shaikh, R. Tewari, and M. Agrawal. On the Effectiveness of DNS-based Server Selection. In *Proceedings of IEEE INFOCOM 2001*, 2001.