*s*Monitor: A Non-Intrusive Client-Perceived End-to-End Performance Monitor of Secured Internet Services

Jianbin Wei and Cheng-Zhong Xu
Dept. of Elec. and Comp. Engg., Wayne State University, Detroit, MI, 48202
Email: {jbwei, czxu}@wayne.edu

Abstract

End-to-end performance measurement is fundamental to building high-performance Internet services. While many Internet services often operate using HTTP over SSL/TLS, current monitors are limited to plaintext HTTP services. This paper presents sMonitor, a non-intrusive server-side end-to-end performance monitor that can monitor HTTPS services. The monitor passively collects live packet traces from a server site. It then uses a size-based analysis method on HTTP requests to infer characteristics of client accesses and measures client-perceived pageview response time in real time. We designed and implemented a prototype of sMonitor. Preliminary evaluations show measurement error of less than 5%.

1 Introduction

The Internet is increasingly being used as a platform to deliver information to remote clients in a secure and timely manner. End-to-end performance monitoring is fundamental to continuously supporting and developing such Internet services. For example, performance monitoring is critical for provisioning of quality of service guarantees. Without accurate measurement of client-perceived service quality, it will be impossible to allocate server resources between different clients.

To effectively support the developing of Internet services, there are three requirements in performance monitoring. First, a performance monitor needs to be non-intrusive so as to minimize its interference with operations of monitored systems. While Web server and Web page instrumentations, as proposed in [2, 11, 19], are easy to be deployed in small Web sites, they need to modify Web servers and Web pages. Thus they are not suitable for large-scale Internet services and are limited to certain Web servers. Second, the monitor should be able to characterize service performance perceived by all clients without biases. Active sampling of Internet services, such as that provided by Keynote [13], can obtain detailed response time characteristics from particular network locations. These characteristics, however, are not representative since they use customized browsers, which are different from that are used by real clients. The browser-instrumentation approaches, such as Page Detailer from IBM [12], also have the same limitations as active sampling and are mainly for service testing and debugging. Therefore, a performance monitor should be close to servers to capture all traffic in and out the servers. Third, to accommodate the increasing deployment of HTTPS services, such as e-commerce service, the monitor should be able to measure performance of HTTPS services as well. Although recently proposed EtE [6] and ksniffer [17] can meet the first two requirements, they fail the last one since they need to parse the HTTP headers, which are unavailable for encrypted HTTPS traffic.

As a remedy, we present *s*Monitor, a non-intrusive server-side performance monitor that is capable of monitoring client-perceived end-to-end response time of HTTPS services in real time. *s*Monitor resides near monitored servers and passively collects traffic in and out of the servers. It then applies a size-based analysis method on HTTP requests to infer characteristics of client accesses. The size analysis is based on our observations that the first HTTP request to retrieve the base object, which normally is an HTML file, of a Web page is normally significantly larger than the following requests for the page's embedded objects. The size difference is because they have different `Accept` headers. Based on the inferred characteristics, *s*Monitor measures client-perceived pageview response time. Furthermore, *s*Monitor calculates the size of an HTTP request solely based on plaintext IP, TCP, and SSL/TLS [1, 9] packet headers. Thus there is no need to parse HTTP headers and *s*Monitor supports performance monitoring of HTTPS services.

This paper describes the design and implementation of *s*Monitor, and presents its preliminary evaluation. To conduct this evaluation, we used *s*Monitor to monitor service performance of HTTP and HTTPS Web servers. The evaluation results showed that the measurement error was less than 5%.

The structure of the paper is as follows. Section 2 discusses the size relationship between HTTP and HTTPS messages. Section 3 presents the design of *s*Monitor and

its implementation. Section 4 presents preliminary evaluations and Section 5 concludes the paper.

2 HTTP over SSL/TLS

In general, secured Internet services use either a version of Secure Sockets Layer (SSL) protocol [9] or Transport Layer Security (TLS) protocol [1] to encrypt HTTP messages before transmitting them over TCP connections. Since SSL and TLS are very similar, our discussion of SSL also applies to TLS unless specified explicitly.

In the SSL protocol, HTTP request messages are first fragmented into blocks of 2^{14} bytes or less. Next, compression is optionally applied. A message authentication code (MAC) over the compressed data is then calculated using HMAC MD5 or HMAC SHA-1 algorithms [14] and is appended to the fragment. After that, the compressed message plus the MAC are encrypted using symmetric encryption. The final step of the SSL record protocol is to attach a 5-byte plaintext header to the beginning of the encrypted fragment. The SSL record data are then passed to lower protocols, such as TCP, for transmission. In this way, with the support of SSL, all HTTP messages are transmitted with a guarantee of their secrecy, integrity, and authenticity.

As is known, it is extremely hard to hide information such as the size or the timing of messages [7]. We conducted experiments to determine the size relationship between HTTP and HTTPS messages. In the experiments, we sent HTTP requests with known sizes over SSL to an HTTPS Web server using Microsoft's Winiest library. The corresponding HTTPS messages were captured using WinPcap and their sizes were measured. There are three issues worth noting in the design of the experiments.

- In our experiments, to control the size of an HTTP request, we set the `Accept` header to meaningless characters. In HTTP/1.1 protocol [8], it specifies that a Web server should send a 406 ("not acceptable") response if the acceptable content type set in a request's `Accept` header cannot be satisfied directly. We found that, however, most Web servers, including Microsoft Internet Information Services (IIS) and Apache Web server, just ignore the unrecognized `Accept` header and send the default response. Thus the meaningless `Accept` header we set in an HTTP request does not bias our experimental results.
- In order to evaluate different protocols, encryption algorithms, and MAC algorithms, we modified the settings of Windows according to [15]. In practice, in IE 6.0 TLS is disabled while in IE 7.0 Beta 1 it is enabled by default. In addition, RC4 and MD5 are always used as the first option for encryption algorithm and MAC algorithm, respectively.

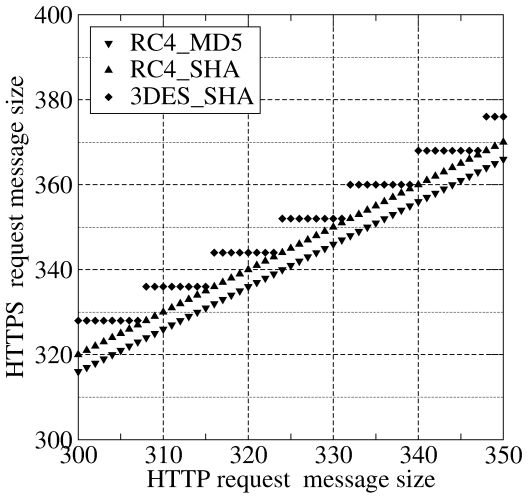


Figure 1: Size relationship between HTTP and HTTPS messages.

- To ensure the HTTPS requests are sent to the server instead of the local cache, we passed flags `INTERNET_FLAG_NO_CACHE_WRITE` and `INTERNET_FLAG_RELOAD` to function `HttpOpenRequest()` to disable caching.

We conducted experiments in which the size of an HTTP request ranges from 100 through 2000 bytes. The size of corresponding HTTPS message is obtained by checking the record header of the first SSL/TLS record fragment with content type as application data. To clearly show the size relationship between HTTP and HTTPS messages, Fig. 1 only depicts the results where the HTTP message size ranges from 300 through 350 bytes. In this figure, RC4_MD5 denotes that RC4 encryption algorithm and MD5 MAC algorithm are used. Legend RC4_SHA and 3DES_SHA follow similar format. We obtained the cipher suite used between the client and the server by checking their handshake messages for session establishment, which are not encrypted.

From Fig. 1 we observe that the size difference between HTTPS and HTTP request messages is always 16 bytes and 20 bytes for RC4_MD5 and RC4_SHA, respectively. It is because RC4 is a stream cipher, in which the ciphertext has the same size as plaintext, and MD5 and SHA calculate a 16-byte and 20-byte MAC, respectively.

In the case of 3DES_SHA, we can infer the size of an HTTP message within 8 bytes from the size of the corresponding HTTPS message. Assuming a 360-byte HTTPS message, since there exists a padding size byte, only another zero or seven bytes need to be padded to make the total an integer multiple of 8 in 3DES. Considering the 20-byte MAC calculated by SHA, the HTTP message therefore ranges from 332 through 339 bytes. It can be observed from Fig. 1.

In SSL 3.0, the padding added prior to encryption of

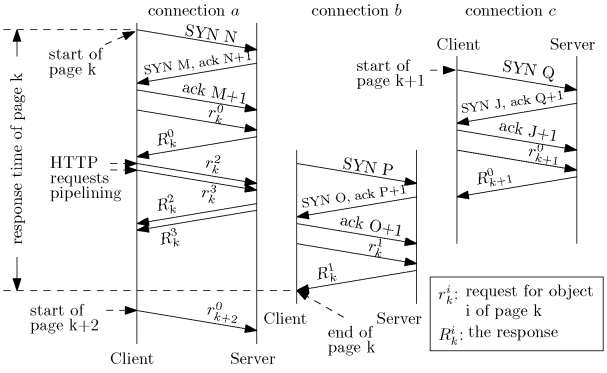


Figure 2: Retrievals of multiple pages and embedded objects using pipelined requests over multiple connections.

an HTTP message is the *minimum* amount required so that the total size of the data to be encrypted is a multiple of the cipher's block size. In contrast, TLS 1.0 and 1.1 (draft) define *random* padding in which the padding can be any amount that results in a total that is a multiple of the cipher's block size, up to a maximum of 255 bytes. It is to frustrate attacks based on size analysis of exchanged messages. From Fig. 1 we observe that random padding is not implemented in IE 6.0 and 7.0 Beta 1. We also verified this observation for TLS used in Firefox.

If compression is performed, the size of HTTP requests will change, which might interfere the size-based analysis method presented in Section 3. However, from Fig. 1 we observe that the optional compression step in SSL/TLS is not performed. It is because no default compression algorithm is defined in SSL and TLS protocols.

In summary, we can obtain the size of an HTTP request from the corresponding HTTPS message in case of stream ciphers, such as RC4, or an estimation within 8 bytes difference in case of block ciphers, such as DES and 3DES. This agrees with the observations made in [21]. Therefore, we can conduct size analysis on HTTPS traffic to measure the response time of HTTPS services.

3 Design and Implementation of *s*Monitor

In general, a Web page consists of a base object and multiple embedded objects. A browser retrieves a Web page by issuing a series of HTTP requests for all objects. The key challenge in end-to-end performance monitoring is to determine the beginning and the end of the retrieval of a Web page. The determination needs to be achieved even when multiple Web pages are retrieved by clients using pipelined HTTP requests over multiple TCP connections. The scenario is depicted in Fig. 2. More important, in HTTPS services, such determination must be achieved without parsing HTTP headers, which are encrypted.

Let r_k^0 and r_k^i denote the HTTP requests for the base object and embedded object i of some Web page k, respectively. Essential to the inference of client-perceived response time is the identification of request r_k^0, which delimits Web page retrievals. In this section we present the size-based analysis method to identify r_k^0. As discussed in Section 2, the size-based analysis method also works for HTTPS services.

3.1 Analysis of HTTP `Accept` header

Our size-based analysis method is based on the HTTP `Accept` header, which is designed as part of the effort for content negotiation. However, HTTP/1.0 and HTTP/1.1 do not specify how browsers should implement the `Accept` header. In this work, we focus on two most popular Web browsers, Microsoft Internet Explorer (IE 6.0 and 7.0 Beta 1) and Mozilla Firefox 1.0 and 1.5 in Windows, which have more than 95% market share [16]. Notice that there is no need for the *s*Monitor to know the browsers used by clients in measuring HTTPS service performance.

In IE, the default `Accept` header is specified in the registry key *Accepted Documents* as part of local machine's Internet settings with size of 56 bytes. When certain applications are installed, their identifications may be added to the registry key. In Firefox, the `Accept` header can be accessed through URL `about:config` and its default value is 99 bytes. There are also several commonly used `Accept` headers in Firefox. Their sizes range from 3 through 56 bytes.

Based on our experiments, we have following important observations regarding the use of `Accept` header in HTTP requests.

- *In an HTTP request, the default* `Accept` *header is normally used in IE if the requested object is to be loaded into a known frame or in Firefox if the object is to be loaded into any frame as the first object, regardless of the content type of the object. Otherwise, in IE the short header "*$\star/\star$*" is normally used or in Firefox the* `Accept` *header is set according to the object's content type.*

A known frame means it is specified using reserved HTML target names "_self", "_parent", or "_top"; or the target name can be found in the window that contains the link or every other window. Otherwise, the frame is unknown. A frame created using target name "_blank" is always an unknown one.

3.2 Size analysis for request identification

Based on the observations, we found that request r_k^0 normally was significantly larger than request r_k^i while the difference between r_k^i and r_k^{i+1} is small. An HTTP request message begins with a request line and is followed by a set of optional headers and optional message body.

The large size difference between r_k^0 and r_k^i is mainly because the following reasons.

- r_k^0 has larger `Accept` header than r_k^i. For example, in IE, the size difference of `Accept` header between default one and the short one is at least 53 bytes and can become even larger than 161 bytes if more software is installed in the client system. In Firefox, the size difference is at least 43 bytes and can be as large as 93 bytes.
- For static Web objects, the request lines in r_k^0 and r_k^i normally have limited size for the ease of Web site administration. For dynamically generated objects, their addresses normally follow the same pattern and the sizes are very close to each other.
- The only difference between requests r_k^0 and r_k^i normally is the `Accept` headers if cookie is not used. In the case when cookie is used, a `Cookie` header will be used in all requests except the first one from a client. Since normally the first request is for base object of a Web page, we also identify it as r_k^0. Thus, the size difference is a good indicator for different requests.
- Across a wide variety of measurement studies, the overwhelming majority of Web requests use the GET method to retrieve Web pages and invoke scripts [10, 18]. They do not have message bodies.

Based on the observation of size difference, we propose a size-based analysis method to identify request r_k^0. Let x_n denote the size of the nth HTTP request message. Let $f(x) = x$ denote the HTTP request size function. Its second derivative can be approximated as

$$\begin{aligned} f''(x_n) &\cong (f'(x_n + h/2) - f'(x_n - h/2))/h \\ &= (f(x_{n+1}) - 2f(x_n) + f(x_{n-1}))/h^2 + O(h^2). \end{aligned}$$

Since the function is discrete, let h be 1, we have

$$f''(x_n) \cong x_{n+1} - 2x_n + x_{n-1}.$$

If $f''(x_n) < 0$, then $f(x)$ is concave down ($\bigcap$) and it has a relative maximum at x_n. Let t denote a configurable threshold of the size difference between r_k^0 and r_k^i. We have

- request n is identified as r_k^0 if its second derivative $f''(x_n)$ is less than t.

The selection of t should maximize the number of correctly detected r_k^0 and limit the number of false positive ones that would otherwise damage the accuracy of *s*Monitor. We found that the `Accept` headers issued by Firefox have smaller size differences than those in IE. Based on analysis of `Accept` header and captured HTTP requests, we set t as -60 bytes in *s*Monitor. We find that it is a good setting through our experiments.

To measure client-perceived response time, we need to consider following issues. First, we must consider the time difference between sending segments from clients and receiving them in servers and vice versa. Normally, when a client issues request r_k^0 and there is no connection between the client and the server, the client must first contact the server for connection establishment using an SYN segment. Request r_k^0 is normally issued right after the last step of the three-way handshake. Therefore, there is at least 1.5 round-trip time (RTT) gap before the server receives the first r_k^0 segment. In the case that request r_k^0 is transmitted over an established TCP connection, the time gap then is 0.5RTT. There is also a 0.5RTT time gap between the server sending the last packet of a response and the client receiving the packet.

Second, in *s*Monitor, the end of a Web page is identified as the last non-empty server segment before another request r_k^0 over the same or other connections between the client and the server. This identification method may cause delayed identification because there exists a time gap between the end of a Web page and the arrival of requests for next Web page. We address this issue using a configurable time-out scheme. If there is no any new segment from either the client or the server, *s*Monitor determines that the Web page is ended. In *s*Monitor, we adopt 5 seconds by default as it is treated as good response time of a Web page [4].

In [5], the authors marked the first packet with size smaller than the MTU as the end of server response. They argued that a server response is packed into a series of packets with size as the MTU except the last one because there is not enough content to fill it. The method, however, is problematic because the actual size of a packet is affected by multiple factors, including the TCP implementation in the server and network conditions between the client and the server. It becomes even worse in the case of packet retransmission and reordering. For example, assuming a server response is packed into an MTU-size packet and a non-MTU size packet, and they are sent to the client together without receiving any acknowledgments. The first packet can be lost and retransmitted after the second one. Their method then determines false end of the server response.

Third, in *s*Monitor we also need to consider the effects of HTTP pipelining. In HTTP pipelining, multiple requests can be issued to the server before the response of the first request is sent out. We found that HTTP/1.1 pipelining was becoming widely used. We examined three-hour traffic in and out the Web server of the college of engineering at wayne state university. The results showed that 19.3% of HTTP requests were pipelined. Also, pipelining is implemented in Firefox as an experimental feature. As observed in [20], more than 90% requests are less than 1000 bytes and most large requests have non-GET methods for services such as web-mail. Thus, if the segment size is the same as the

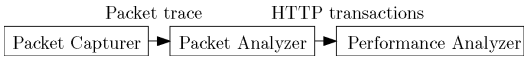


Figure 3: The architecture of *s*Monitor.

MTU, *s*Monitor assumes that this is the first segment of a request and more are coming. Otherwise, the monitor marks the segment as the end of a request.

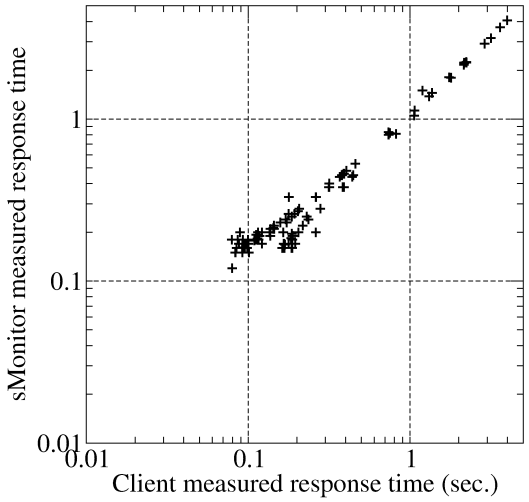
Fourth, we also need to deal with parallel downloading, which means a user retrieves multiple Web pages from one server at the same time. During parallel downloading, the requests for these pages can intertwine each other, which might cause false identification of the end of a Web page. As an example, assuming that the user accesses Web pages k and $k+1$ simultaneously. In *s*Monitor, all requests issued after request r^0_{k+1} are identified as the requests for embedded objects in page $k+1$ while request r^i_k may come after r^0_{k+1}. Consequently, the measured response time of page k is smaller than that perceived by the user while it is opposite for page $k+1$. However, *s*Monitor still measures average response time perceived by the client accurately. The parallel downloading of embedded objects within a Web page does not affect the performance of *s*Monitor because it does not affect the identification of the beginning of r^0_k for the Web page.

3.3 Implementation of *s*Monitor

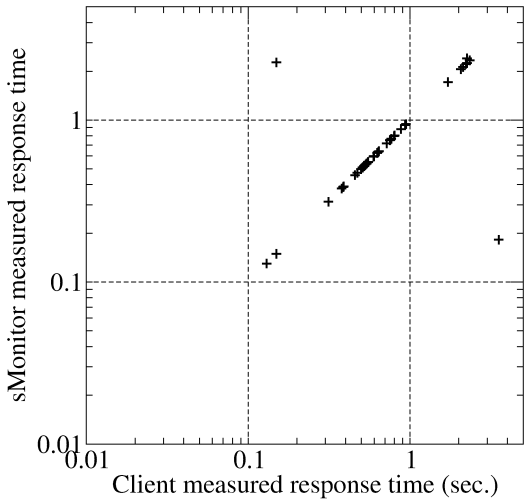
A prototype of *s*Monitor has been implemented at user-level as a stand-alone application in C. It captures packets in and out of the monitored servers, analyzes packet headers to extract packet information, derives page-related data from packet headers, and conducts performance analysis. Fig. 3 presents the architecture of *s*Monitor. The *packet capturer* collects live network packets using pcap [22] (libpcap on Unix-like systems or WinPcap on Windows). In the *packet analyzer*, *s*Monitor parses the packet headers to extract HTTP transaction information, such as HTTP request sizes, and passes them to the *performance analyzer*. Such information can be obtained by parsing TCP/IP and SSL/TLS headers. The *performance analyzer* derives client-perceived response time of the monitored services.

4 Preliminary Evaluation

To evaluate the accuracy of *s*Monitor, we conducted two experiments and compared the response time measured by *s*Monitor with those measured on client side. In the first experiment, we modified Surge [3] so that the first requests for Web pages have long `Accept` headers. We also modified Surge so that it records the pageview response time. We then set up a Web server in Detroit, MI. Surge was deployed in a node of PlanetLab in UCSD to access the Web server. In the experiment, around 140 Web pages were retrieved from the server in 10 minutes.



(a) PlanetLab evaluation.



(b) Evaluation with real services.

Figure 4: Accuracy evaluation of *s*Monitor.

*s*Monitor was deployed on the Web server to capture the network traffic and measure the response time.

Fig. 4(a) shows the comparison between the client-side measurement and the *s*Monitor measurement. From this figure we can clearly see that these two measurements have strong linear relationship since the *s*Monitor measured response times are very close to those recorded by Surge directly. On the other hand, when the response time is smaller than 1 second, we can see that the measurement difference is relatively large. This is mainly caused by the variance in the estimated RTT. A numerical analysis shows that the average response times measured by Surge and *s*Monitor are 1.00 and 0.95 seconds, respectively. The difference is 5%.

In the second experiment, we used IE to access 44 Web pages in several real Web services, including several banks' Web sites. We also recorded the correspond-

ing response time manually. The *s*Monitor was placed on the client to measure the response time.

The results are plotted in Fig. 4(b). Comparing with Fig. 4(a) we can see that the measured response times are very accurate even when the response times are smaller than 1 second. This is because the client-side placement of *s*Monitor removes the variance in RTT estimations. From this figure we can also see several falsely identified Web pages. For example, when the client-perceived response time is 3.5 seconds, *s*Monitor measures the response time as 0.18 seconds since it incorrectly delimits the beginning and ending of the Web page. Similarly, when the client-perceived response time is 0.15 seconds, *s*Monitor gives the response time as 2.27 seconds. However, from the figure we can see that in most cases the Web pages are correctly delimited. The average response times recorded manually and those by *s*Monitor are 1.02 and 0.99 seconds, respectively. The difference is 3%. Therefore, from these two experiments we conclude that *s*Monitor is very accurate in measuring the client-perceived pageview response time.

5 Conclusions

In this paper we presented *s*Monitor to monitor end-to-end performance of HTTPS services. We designed a size-based analysis method on HTTP requests to characterize the client access behaviors. Based on the characteristics, we designed and implemented a prototype of *s*Monitor. We investigated its accuracy in measuring client-perceived response time using HTTPS and HTTP services and found that the difference was less than 5%.

We note that the accuracy evaluation is preliminary. The environments of real Web services are very complicated. In the future, we plan to conduct comprehensive evaluation of our approach using live Web services. The size-based analysis method may also be improved. For example, we used simple heuristic methods to address the issues of parallel downloading and HTTP pipelining. In the future, we will work on systematic solutions for these issues.

Acknowledgements

We would like to thank our shepherd, Erich Nahum, for helping us to significantly improve this work and the anonymous reviewers for their numerous helpful comments. This work was supported in part by US NSF grants ACI-0203592, CCF-0611750, and NASA 03-OBPR-01-0049.

References

[1] ALLEN, C., AND DIERKS, T. *The TLS Protocol Version 1.0*. RFC 2246, January 1999.

[2] ALMEIDA, J., DABU, M., MANIKUTTY, A., AND CAO, P. Providing differentiated levels of service in web content hosting. In *Proc. ACM SIGMETRICS Workshop on Internet Server Performance* (June 1998), pp. 91–102.

[3] BARFORD, P., AND CROVELLA, M. Generating representative Web workloads for network and server performance evaluation. In *Proc. ACM SIGMETRICS conference* (June 1998), pp. 151–160.

[4] BHATTI, N., BOUCH, A., AND KUCHINSKY, A. Integrating user-perceived quality into Web server design. In *Proc. Int'l World Wide Web Conference* (2000), pp. 1–16.

[5] CHENG, H., AND AVNUR, R. Traffic analysis of SSL encrypted Web browsing. Available at: `http://www.cs.berkeley.edu/~daw/teaching/cs261-f98/projects/final-repor%ts/ronathan-heyning.ps`.

[6] CHERKASOVA, L., FU, Y., TANG, W., AND VAHDAT, A. Measuring and characterizing end-to-end internet service performance. *ACM Trans. on Internet Technology*, 3(4) (2003), 347–391.

[7] FERGUSON, N., AND SCHNEIER, B. *Practical Cryptography*. John Wiley & Sons, 2003. p114.

[8] FIELDING, R. T., GETTYS, J., MOGUL, J. C., NIELSEN, H. F., MASINTER, L., LEACH, P. J., AND BERNERS-LEE, T. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, June 1999.

[9] FREIER, A. O., KARLTON, P., AND KOCHER, P. C. The SSL protocol version 3.0, November 1996. `http://wp.netscape.com/eng/ssl3/ssl-toc.html`.

[10] HERNANDEZ-CAMPOS, F., JEFFAY, K., AND SMITH, F. D. Tracking the evolution of web traffic: 1995-2003. In *Proc. Int'l Symp. on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems* (2003), pp. 16–25.

[11] HEWLETT-PACKARD DEVELOPMENT COMPANY, L.P. Openview transaction analyzer. `http://openview.hp.com/`.

[12] IBM CORP.. Page detailer. `http://www.alphaworks.ibm.com/tech/pagedetailer`.

[13] KEYNOTE SYSTEMS, INC. www.keynote.com.

[14] KRAWCZYK, H., BELLARE, M., AND CANETTI, R. *HMAC: Keyed-Hashing for Message Authentication*. RFC 2104, February 1997.

[15] MICROSOFT CORPORATION. How to restrict the use of certain cryptographic algorithms and protocols in schannel.dll, December 2004. `http://support.microsoft.com/?kbid=245030`.

[16] NETAPPLICATIONS.COM. Firefox continues to erode microsoft dominance, June 2005. `http://www.netapplications.com/news.asp`.

[17] OLSHEFSKI, D. P., NIEH, J., AND NAHUM, E. ksniffer: Determining the remote client perceived response time from live packet streams. In *Proc. USENIX Operating Systems Design and Implementation* (2004).

[18] PADMANABHAN, V. N., AND QIU, L. The content and access dynamics of a busy Web site: Findings and implications. In *Proc. ACM SIGCOMM* (2000), pp. 111–123.

[19] RAJAMONY, R., AND ELNOZAHY, M. Measuring client-perceived response time on the WWW. In *Proc. USENIX Symp. on Internet Technologies and Systems* (March 2001).

[20] SMITH, F. D., HERNANDEZ-CAMPOS, F., JEFFAY, K., AND OTT, D. What TCP/IP protocol headers can tell us about the Web. In *Proc. ACM SIGMETRICS* (2001), pp. 245–256.

[21] SUN, Q., SIMON, D. R., WANG, Y.-M., RUSSELL, W., PADMANABHAN, V. N., AND QIU, L. Statistical identification of encrypted Web browsing traffic. In *Proc. IEEE Symp. on Security and Privacy* (May 2002).

[22] TCPDUMP. `http://www.tcpdump.org/`.