SINE: Cache-Friendly Integrity for the Web

Camille Gaspard [1] Sharon Goldberg [2] Wassim Itani[1]
Elisa Bertino [1] Cristina Nita-Rotaru [1]
cgaspard@cs.purdue.edu, goldbe@princeton.edu, witani@cs.purdue.edu
bertino@cs.purdue.edu, crisn@cs.purdue.edu
[1] Purdue University, West Lafayette, IN 47907 USA
[2] Princeton University, Princeton, NJ 08544 USA

Abstract

In this paper we present SINE, a cache-friendly protocol for integrity-enforced web documents. SINE operates by decoupling integrity from confidentiality and provides web documents with an integrity-enforcement authentication tag that can be incrementally verified by multiple parties. We developed a prototype implementation of SINE with minimal changes to the standard web client/server architecture and conducted experiments using the standard Squid web proxy. Our experimental results show that SINE provides the required integrity services to web pages while maintaining the standard caching mechanisms. Moreover, by taking advantage of caching, SINE shows a performance gain that reached a factor of 5 over SSL/TLS.

I. Introduction

With the increased use of data-rich web 2.0 applications and the amount of web-accessible information, a major concern for users and content providers is the integrity of these web documents. However, Reis et. al show in [8] that up to 1% of all HTTP traffic may be tampered with while it traverses the network from a web-server to a client (usually a web browser). The standard way to prevent such attacks is to require web clients to use the Hypertext Transfer Protocol Secure (HTTPS) [3]. HTTPS refers to the use of HTTP over an encrypted Secure Sockets Layer (SSL) or Transport Layer Security (TLS) [5] connection. With TLS, the server's public key is used to negotiate a pairwise shared secret key between the client and server; the client and server then use symmetric-cryptographic operations (keyed with their pairwise-secret key) to encrypt and authenticate each transport-layer segment that they send. While SSL/TLS protects both the integrity and confidentiality of web documents, this paper focuses only on *integrity protection*.

Unfortunately, HTTPS significantly degrades performance because it interferes with a very popular web technology called *web document caching*. With web document caching, copies of frequently requested web documents are stored on client machines or on specialized proxy servers (such as Squid [2]) on the clients' local network. Web caching allows large organizations to significantly reduce their upstream bandwidth usage and cost, while significantly improving application responsiveness for clients and reducing load on web-servers. Because HTTPS encrypts each document the client wants to download using a pairwise key between the client and the server, other clients can no longer take advantage of the frequently-requested documents stored on the proxy. One way to get around this problem is to allow the server to delegate its symmetric keys to the proxy and have the proxy use these symmetric keys to simulate an HTTPS session to the client. However, this solution has the major disadvantage that it requires the client to *trust* the proxy.

If the client does not trust the proxy, the server could instead append a public-key digital signature to each web document. Then, if the proxy stores the document and its signature, the web client can verify the integrity of the document using the server's public key, even if it does not trust the proxy. While it initially seems very appealing, we believe this solution is completely impractical. On one hand, suppose that a single digital signature is computed over the entire document. Then there is an unacceptable latency at the client, since it must wait to download the entire document before it verifies the digital signature. On the other hand, suppose that the document is broken into small blocks that are individually authenticated using a public-key cryptography (as in HTTPS, where documents are broken into transport-layer segments that are individually authenticated using a symmetric-key cryptography). Since public-key cryptography is known to be significantly (> 100 times) slower than symmetric-key cryptography, this solution incurs an unacceptable computational cost at the client.

To get around these issues and address the tension between performance and security, this paper presents SINE, a cache-friendly protocol for integrity-protected web documents. SINE is a high-performance web protocol that allows clients to take advantage of caching opportunities without requiring them to trust the proxy or each other. With SINE, clients can incrementally verify the integrity of web documents using one (or fewer) public-key cryptographic operations and a small number of fast cryptographic hash computations. The contributions of this paper are as follows:

- We show how to obtain a practical integrity-only security service for web documents that requires minimal modification to the current web client/server model. We base our work on the cryptographic protocols proposed by Gennaro and Rohatgi in [6], and present three protocol variants, SINE, SINEB, and SINEX, that trade-off computational cost and latency. We compare the three variants and discuss appropriate application scenarios.
- We develop a prototype implementation of SINE, SINEB, and SINEX with minimal changes to the standard web client/server architecture and evaluate all three protocols using the standard Squid web proxy. Our experimental results show that SINE provides the required integrity services while maintaining the standard web caching mechanisms. We also find that SINE can improve performance over SSL/TLS (by a factor of up to 5 when the file size is large), because it allows clients to take advantage of web caching.

This paper is organized as follows. We start with the system model and requirements in Section II and then describe the different flavors of SINE in Section III. Section IV evaluates SINE in different scenarios. We discuss related work in Section V and conclude in Section VI.

II. Model and Requirements

We present the client-server model considered in this work, as well as our design requirements.

A. Client Server Model

The system model complies with the traditional web client/ server architecture and is functionally represented by three main components: the client, the server, and the proxy.

When a client sends an HTTP request to the web server, the proxy intercepts client requests and checks if the requested content is available locally. If it is, the proxy replies with the requested content; otherwise, it forwards the HTTP request to the server. On the way back, the proxy intercepts the HTTP response content and stores it locally for servicing clients requesting the same content in the future. In HTTPS on the other hand, the client must establish a direct connection to the server. All communication between client and server is encrypted and authenticated under a secret pairwise key. Because this key is kept secret from the untrusted proxy, all HTTPS communication must bypass the proxy entirely.

B. Design Requirements

We have identified a number of requirements that we believe are crucial for any integrity solution for web content:

H(x)	a collision-resistant hash function
PK	a public key from a public-key scheme
SK	a secret key from a public-key scheme
$Sign_{SK}(m)$	a digital signature of m using SK
$Ver_{PK}(m, tag)$	a digital signature verification function of m and tag using PK
$B_1, B_2, ..., B_n$	an equally-sized block partitions of an object. The blocks equal size is guaranteed by padding the last block
Y_i	a data structure containing file block B_i and its respective hash authenticator
,	the string concatenation operator

TABLE I
Notation

End-to-end integrity protection. is required from server to client. The client only trusts the web server that originates the web content. No trust should be placed in the proxy.We only require the proxy to cache web content; because the proxy is untrusted, it is not responsible for verifying or guaranteeing the integrity of the content it receives from the server. The server does not assume any trust relationship with either the client or proxy.

Incremental verification. Standard web browsers progressively render content in order to provide increased application responsiveness. A practical solution for web content integrity has to allow the client to incrementally verify the integrity of every part of the document as it is downloaded.

Support for caching. We would like our protocols to be backward-compatible; thus, we design protocols that require no change to the web proxy architecture.

Leveraging SSL/TLS. Given that the SSL/TLS protocol has been under scrutiny for many years, it is desirable (when possible) to leverage existing components of SSL/TLS.

Communication/computation overhead. As resource-constrained devices (e.g., smart phones, PDAs) are currently being used to browse the web, we require solutions to have small communication and computation overhead for the client.

III. SINE Protocols Description

We start with the basic SINE protocol based on [6]. We then show two protocol flavors, SINEB and SINEX, that leverage existing web architecture and web access scenarios to provide better performance. The notation used in the rest of the section is presented in Table I.

A. SINE Overview

At a high level, SINE, works as follows: whenever a client requests a web page from a web server, the server answers with the requested page and a corresponding *Authentication Tag* (AT). This AT is a separate file or a header added to the regular HTTP response of the requested object, and is generated when the page is updated on the server. As proposed by Gennaro and Rohatgi [6], the AT is a public-key authenticated hash chain on a set of small (typically 1 KB) equally-sized blocks of the exchanged web document. The client can incrementally verify the integrity of each block of the web page by verifying the interdependent collision-resistant hash values, while the authenticity of the entire document is guaranteed by a (public-key) digital signature on the first entry of hash chain.

Authentication Tag Generation. Figure 1 shows the authentication tag AT for a file. First, the server divides the file into equally-sized blocks $(B_1, B_2, ..., B_n)$ (with last block padded). Then, the server takes a backwards pass over the file using the blocks to generate a $(n+1)$-elements hash chain, where n is the resulting number of blocks in the file. Specifically, Y_i is computed as the hash function applied to the previous authenticator Y_{i-1}. The first authenticator, Y_0, is computed as the server's digital signature over Y_1, n the number of blocks in file, a timestamp T, and an expiration date E. T and E protect against replay attacks and allow clients to check the freshness of server responses. n prevents an attacker from delivering only the first part of the file (and withholding the rest of the file without the client's knowledge).

Storing the Document and AT. After the first request for an object, the proxy will store the page and the corresponding AT file. For subsequent client requests, the proxy will thus be able to deliver the requested web page along with its corresponding AT file without having to contact the original web server. If the cache expires, the proxy requests an updated version of the page and its corresponding AT from the server.

Integrity Verification. In SINE the proxy (and server) each delivers the document to the client as blocks Y_i for $i = 0...n$. When the client receives the first block, Y_0, it verifies this block using the public key of the server (and verifies freshness by checking T and E). As with TLS/SSL, the client first needs to cross-check the certificate of the server it is communicating with against the chain of trusted CAs it has in its database to validate the claimed public key of the server. The authenticator Y_i is used to authenticate the subsequent block B_{i+1}. Authentication of blocks Y_i for $i = 1...n$ is performed using fast collision-resistant hashing. Finally, the client verifies that it has received the entire document by comparing the number of blocks received with the authenticated n from Y_0.

Security. In [6], the authors prove that if the server correctly formed the AT and if the client uses the correct public-key for verification, then it is infeasible for an adversary to generate an AT for a document that was not created by the server. It follows that the integrity of the web document is guaranteed even if the document and AT are stored on the untrusted proxy.

Overhead. In SINE, the computation cost of the authentication tag AT at the server (resp. client) consists of one 'slow' public-key digital signature (resp. verification)

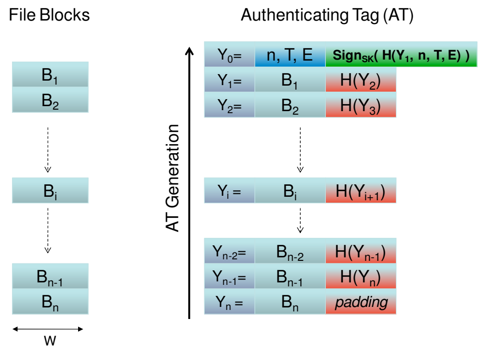

File Blocks
Authenticating Tag (AT)
B_1
B_2
B_i
B_{n-1}
B_n
w
AT Generation
$Y_0=$ | n, T, E | $Sign_{SK}(H(Y_1, n, T, E))$
$Y_1=$ | B_1 | $H(Y_2)$
$Y_2=$ | B_2 | $H(Y_3)$
$Y_i=$ | B_i | $H(Y_{i+1})$
$Y_{n-2}=$ | B_{n-2} | $H(Y_{n-1})$
$Y_{n-1}=$ | B_{n-1} | $H(Y_n)$
$Y_n=$ | B_n | padding

Fig. 1. SINE Authentication Tag

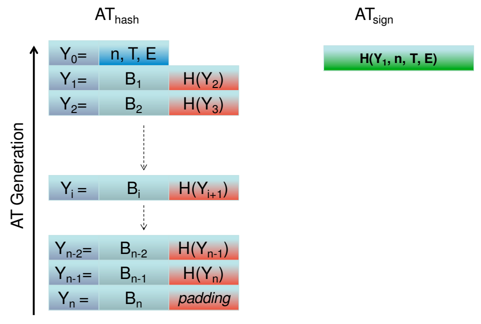

AT_{hash}
AT_{sign}
$H(Y_1, n, T, E)$
AT Generation
$Y_0=$ | n, T, E
$Y_1=$ | B_1 | $H(Y_2)$
$Y_2=$ | B_2 | $H(Y_3)$
$Y_i=$ | B_i | $H(Y_{i+1})$
$Y_{n-2}=$ | B_{n-2} | $H(Y_{n-1})$
$Y_{n-1}=$ | B_{n-1} | $H(Y_n)$
$Y_n=$ | B_n | padding

Fig. 2. SINEB Authentication Tag

and n 'fast' collision-resistent hash computations.

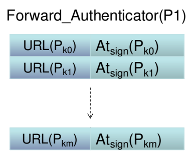
B. SINEB Overview

SINEB is a variant of the SINE protocol that amortizes the cost of computing the digital signature across multiple web pages. Consider a user who uses SINE to browse through a number of webpages on a certain website (e.g. CNN.com) over some time period or *session*. With SINE, the client needs to perform one public-key verification per web object requested. We now propose SINEB, which reduces the number of public-key operations by exploiting the fact that these webpages are all generated by the same web server. With SINEB, during a *session* with a particular webserver, the client preforms only a single (slow) public-key operation and thereafter preforms only fast symmetric-key cryptographic operations.

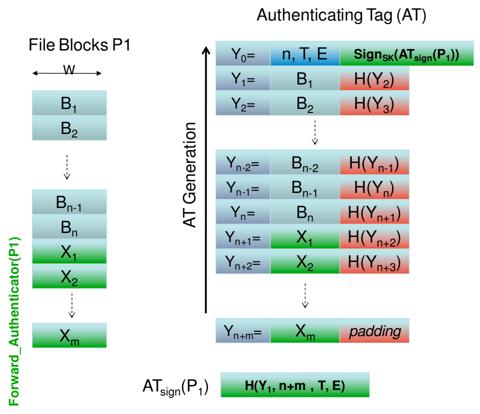
Authentication Tag Generation. In SINEB, the signature-authenticated component of the authentication tag AT is separated from the non-authenticated component, as shown in Figure 2. The first authenticator is divided into two parts: AT_{hash} containing the authenticators $Y_1....Y_n$ as in SINE (but not including Y_0), and AT_{sign}, containing collision-resistant hash of Y_1, n, T, E.

Storing the AT. This time, AT_{hash} is cached along with the web document on the proxy server, but the web server will store and secure the AT_{sign} for every web document it serves.

Integrity Verification. In a simplified version of SINEB, when a client wants to communicate with a web site, he starts by establishing an SSL session with the server. After the SSL connection is established, the client



(a) $Forward\_Authenticator(P1)$



(b) Authentication Tag

Fig. 3. SINEX Authentication Tag

initiates two requests per web object: one using HTTP for the object itself and another SSL request for the AT_{sign} of the same object. The client can then verify the integrity of the web document by combining AT_{hash} and AT_{sign} and proceeding as in the SINE protocol. Note that SSL performs only symmetric-key cryptographic operations once the handshake phase is complete, so that verifying the AT_{sign} tag requires only symmetric-key cryptographic operations.

We note that requiring an extra SSL request per web document adds latency to the client because of the extra RTT with the server, and burdens the server with additional SSL requests. A possible solution is to batch the AT_{sign} for all the web objects that are expected to be requested in one session together in one large authenticator $AT_{collect}$.

C. SINEX Overview

We describe SINEX, another variant of the protocol proposed in Section III-A that combines the attractive properties of SINE and SINEB. SINEX requires only a single public-key verification per session (i.e. multiple pages served from the same website), and does not incur any communication with the web server on a per-request basis. SINEX exploits the fact that the webpages at a website are interconnected in manner that (often) allows the server to predict the sequence in which the client accesses the webpages on a website.

The Expected Set. Consider a single website with a page P_1 that links to a set of other pages at the same website. We refer to this set of linked pages as $Expected\_Set(P_1)$. When content is static, the $Expected\_Set(P_1)$ is deterministic and the same for each user. In the case of dynamic content that is the same for each client (e.g., a stock update on CNN.com), the $Expected\_Set(P_1)$ can be computed each time the content changes. For these cases, we assume a server function $Compute\_Expected\_Set(P_1)$ that returns $Expected\_Set(P_1)$, and show how to protect these pages with SINEX. However, we do not advocate using SINEX to protect user-specific dynamic content (e.g. an webmail inbox) where content is tailored to each user; this content should be protected using SINE or SINEB.

Authentication Tag Generation. As shown in Figure 3(a) the AT_{sign} elements of the $Expected\_Set(P_1)$ are stored in a new data structure, referred to as $Forward\_Authenticator(P_1)$. As shown in Figure 3 the $Forward\_Authenticator(P_1)$ is then appended to the end of the page P_1 (shown as blocks $X_1, ..., X_m$). Finally, the page P_1 and $Forward\_Authenticator(P_1)$ are authenticated (using a digital signature on the hash chain) as in SINE.

Integrity Verification. SINEX amortizes the cost of public-key verification across multiple pages, without requiring the client to establish an SSL connection with the server. To see how, notice that in SINEX, the first authenticator Y_0 is the root of the hash chain that guarantees the integrity of P_1 *as well as* $Forward\_Authenticator(P_1)$. Suppose the client would like to access a page P_1 followed by a linked page P_{ki} that is stored in $Forward\_Authenticator(P_1)$. To do this, the client first processes the AT of page P_1, and verifies the integrity of a block $X_i = AT_{sign}(P_{ki})$ using a fast collision-resistant hash computation. Next, when the client begins processing the AT of page P_{ki}, it no longer needs to perform a public-key verification of page P_{ki}'s first authenticator, (as it already did this when it verified the integrity of $X_i = AT_{sign}(P_{ki})$ in page P_1's AT).

D. Discussion

We compare our protocols below and in Table II.

SINE. With SINE, one public-key verification per web object is needed, while no communication is established with the original web server when the cache is not stale. SINE in its original form is very suitable for scenarios where a user accesses only one particular website and downloads one particular object.

TABLE II
COMPARISON FOR N REQUESTS IN ONE SESSION

	SINE	*SINEB*	*SINEX*
Public-key verification	N	1	*1 (expected)
RTTs with server	0	N	0

Protocol	*Description*
http	HTTP without any caching between client and server.
http-proxy	Same as *http* but using a proxy with a variable connection bandwidth between client and server.
https	HTTPS without any caching between client and server.
SINE	SINE without any caching between client and server.
SINE-proxy	Same as *SINE* but using a proxy with a variable connection bandwidth between client and server.
SINEB	SINEB without any caching between client and server.
SINEB-proxy	Same as *SINEB* but using a proxy with a variable connection bandwidth between client and server.
SINEX	SINEX without any caching between client and server.
SINEX-proxy	Same as *SINEX* but using a proxy with a variable connection bandwidth between client and server.

TABLE III
PROTOCOLS USED IN EXPERIMENTAL EVALUATION

SINEB. SINEB has the advantage that it amortizes the cost of both the public-key operations and the download of the AT_{sign} over consecutive requests in one session. As a result, SINEB is more appropriate for scenarios where the user is interested in browsing one specific website for a considerable amount of time. However, SINEB incurs the cost of a RTT with the server for each fetched AT (or collected set of ATs).

SINEX. By anticipating the pages a client will visit after reaching a certain web page in a web site, SINEX can achieve the same security guarantees of SINE with a single public-key verification and without paying the cost of RTTs with the web server. We believe that SINEX is the most appropriate choice for integrity protecting web documents in common web scenarios.

IV. EVALUATION

Experimental Setup. We evaluated SINE on a real network using two local Linux machines (located on a campus network within the US), to implement the client and proxy, and one geographically-separated PlanetLab [1] machine (located in France) to implement the server. The machines were each equipped with a 3.4 GHz uniprocessor Pentium CPU and 1 GB of physical memory running Linux with a 2.6 kernel. The web server used was Apache 2.2.10 and the proxy server was Squid 3.0. The time required to compute the $Expected\_Set$ is *not* included in our evaluation. The evaluated protocols are summarized in Table III.

Throughput. We conducted experiments in two scenarios: direct connection between the client and the server, and connecting using a local proxy.

To simulate a client connected to a local proxy via a typical Ethernet or WiFi connection, we connect our two local Linux machines (client and proxy) using an effective 100 Mbps connection that is routed through the local campus network. The PlanetLab machine acts as the

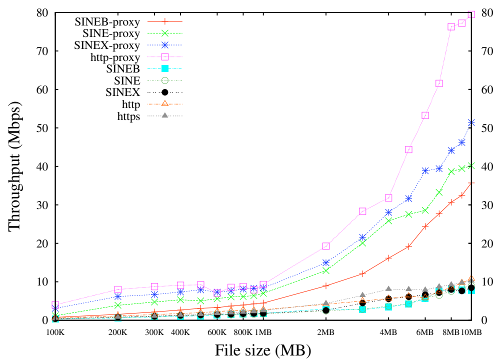


Fig. 4. Throughput per file size

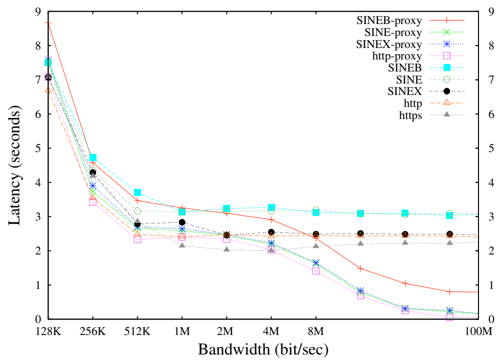


Fig. 5. Latency for one 100 KB file

server. We measured the *average throughput* (the average number of bits transmitted in a second) for different file sizes.

As depicted in the Figure 4, all three SINE-flavored protocols used with a proxy have a significant performance gain over HTTPS, particularly for larger file sizes. Notice that the throughput of SINE is comparable to that of regular unprotected HTTP. Figure 4 also confirms that SINEX achieves the best throughput of all the SINE flavors; indeed, when the file is large, SINEX provides a fivefold improvement over HTTPS.

Interestingly, one anomaly we observed is that HTTPS slightly outperformed HTTP without proxy. We suspect this behavior is due to the compression performed by SSL.

Latency. To assess the latency perceived by the user while browsing the web using SINE we have conducted experiments in two scenarios: single fixed-size file and a web portal.

Fixed-size file. We first measured the latency for a single fixed 100 KB file. The server-proxy measured bandwidth was around 4 Mbps, and we increased the bandwidth of the client's connection from 128 Kbps to 100 Mbps using the Linux Traffic Control (i.e., the *tc* command). We ran the experiments 30 times each and present the average over all runs. When a proxy is used, Figure 5 shows that SINE and SINEX perform as well as HTTP, while SINEB incurs an additional latency due to a round trip to the server (see Table III).

Static content. We instrumented the main page of a

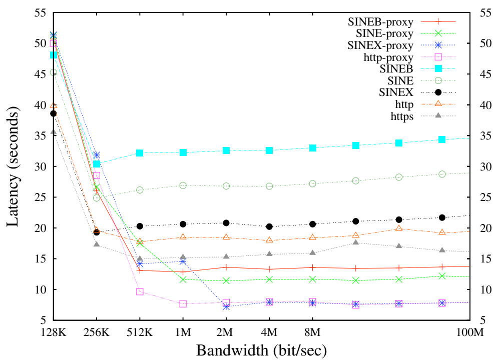


Fig. 6. Latency for a portal main page of 330 KB

portal web site (www.msn.com) and simulated the browser rendering of such a page. Our analysis showed that in order to render www.msn.com, the browser needs to connect to 22 hosts on the msn.com subnet, getting around 15KB of content in one TCP connection from each host. The total size of the web page was 330 KB.

The results depicted in Figure 6 show that SINEX and HTTP both operating with a proxy provide almost similar perceived latency to the user when browsing a portal web site. This means that the client gets the integrity guarantees of SINE almost without any noticeable loss in responsiveness. On the other hand, the same figure shows that the latency perceived by clients when using HTTPS is almost three times worse than what SINEX can guarantee with a proxy. In the case in which all the elements of the portal page were a “miss” in the cache (these elements are not found in the cache store), the experimental results show that SINEX is only slightly worse than pure HTTP in terms of responsiveness. Furthermore, we can see that SINEX is only slightly worse than HTTPS in this case of a “cold cache”.

V. Related Work

SSL/TLS. SSL is currently the predominant security protocol providing end-to-end confidentiality and integrity to web content. SSL interferes with performance-oriented network services such as web document caching (as discussed in Section II). However, SSL provides confidentiality services that are *not* provided by our (integrity-only) SINE protocol.

SSL Splitting [7]. SSL Splitting allows a web proxy to serve cached documents directly to a client while requiring the server to forward only the symmetric-key cryptographic *authentication tags* for these documents to the clients via the proxy. Both SSL Splitting and SINE provide incrementally-verifiable integrity checking using mostly symmetric-key cryptographic operations. However, SINE allows the client to verify the integrity of cached web documents *without* communicating with the server. Thus, as compared to SSL Splitting, SINE reduces the latency at the client and the load at the server. We believe that the only cost that SINE incurs over SSL Splitting is a small number of public-key operations, which can be amortized over many webpages.

HTTP Merkle Trees [4]. In [4], web documents are represented in a Merkle tree that authenticates both the existence and non-existence of web resources. While this solution does not interfere with caching, it is highly inefficient in dealing with dynamic content. Changing one bit in any of the web pages requires a secure redistribution of the secured root hash, and the recalculation and redistribution of parts of the tree. SINE (and SINEX), on the other hand, recalculates only the pages *linking* to the webpage being updated.

Javascript-based Integrity Checking [8]. Recently, [8] presented a non-SSL integrity-checking solution for web content exchanged using HTTP. In [8], integrity-checking Javascript code is embedded in the webpage provided by the server. Once the webpage is rendered by the client browser, the Javascript code executes and verifies the webpage’s integrity. In [8], the goal is to develop a HTTP-based solution that does not require changing current browsers, so they do not require cryptographic authentication of the webpage or the embedded Javascript. Their solution is vulnerable to adversarial modifications of webpage and/or Javascript code. We believe that the ideas in [8] are more appropriate for detecting errors that occur *randomly* (rather than adversarially) as a webpage traverses the network.

VI. Conclusion

In this paper we presented SINE, a family of security protocols based on [6] that ensures the integrity of web documents while supporting standard web caching mechanisms. SINE allows web clients to verify a web document incrementally, as it is downloaded from the network. The paper presented the protocol design and supported it with a prototype implementation using standard web architecture components. The experiments conducted showed that SINE provides the required integrity services to web pages while maintaining the standard caching mechanisms. We believe that in settings where confidentiality is not required, SINE has a significant advantage over SSL/TLS; indeed, when files are large, SINE showed a fivefold improvement in throughput over SSL/TLS.

References

[1] PlanetLab. http://www.planet-lab.org.
[2] Squid: Optimising Web Delivery. http://www.squid-cache.org/.
[3] HTTP over TLS, 2000. RFC2818.
[4] R. J. Bayardo and J. Sorensen. Merkle tree authentication of http responses. In *WWW ’05*, pages 1182–1183, 2005.
[5] T. Dierks and C. Allen. RFC 2246: The TLS protocol version 1, January 1999. Status: PROPOSED STANDARD.
[6] R. Gennaro and P. Rohatgi. How to sign digital streams. In *Proceedings of CRYPTO 97*, pages 180–197. ACM, 1997.
[7] C. Lesniewski-Laas and M. F. Kaashoek. SSL splitting: securely serving data from untrusted caches. In *SSYM’03*, pages 13–13, 2003.
[8] C. Reis, S. D. Gribble, T. Kohno, and N. C. Weaver. Detecting in-flight page changes with web tripwires. In *NSDI’08*, pages 31–44, 2008.