# HTTPI: An HTTP with Integrity

Taehwan Choi
Department of Computer Science,
The University of Texas at Austin
ctlight@cs.utexas.edu

Mohamed G. Gouda
National Science Foubdation
The University of Texas at Austin
mgouda@nsf.gov

***Abstract*—The World Wide Web famously supports two transport protocols: HTTP and HTTPS. These two protocols are at the opposite ends of three dimensions: security guarantees, cost of use, and compatibility with middle boxes (e.g. cache proxies) in the Internet. At one end, HTTP provides no security guarantees, but it is inexpensive to use, and is compatible with middle boxes in the Internet. At the other end, HTTPS provides three security guarantees, but it is expensive to use and is not compatible with middle boxes. Although the three security guarantees provided by HTTPS, namely server authentication, message integrity, and message confidentiality, are important in general, many web servers (e.g. news) do not need the message confidentiality guarantee. In this paper, we present a new transport protocol for the Web, named HTTPI. This protocol provides both server authentication and message integrity, but not message confidentiality. Like HTTP, HTTPI is inexpensive to use and is compatible with middle boxes, and like HTTPS, it defends against many cyber attacks (e.g. Pharming attacks) that HTTP cannot defend against. We developed a preliminary implementation of HTTPI and showed through experimentation that the throughput of HTTPI is within 1.2% from that of HTTP and is 37% better than that of HTTPS.**

## I. Introduction

The World Wide Web supports two well-known transport protocols: HTTP [1] and HTTPS [2]. These two protocols have different costs and provide different security guarantees for the web applications deployed on top of them. At one end, HTTP is inexpensive to use but provides no security guarantees for any web application deployed on top of it. At the other end, HTTPS is expensive to use but provides three important security guarantees for any web application deployed on top of it. These three security guarantees are (1) server authentication, (2) message integrity, and (3) message confidentiality [3].

In most cases, HTTPS is also augmented with a password protocol in order to provide the added guarantee of client authentication. (Note that HTTP cannot be easily augmented with a password protocol to provide client authentication.)

A third transport protocol for the web, namely HTTP Authentication or HTTPA for short, has also been proposed [4]. The cost of using HTTPA is somewhere in the middle between the two costs of using HTTP and of using HTTPS. HTTPA provides only two security guarantees: client authentication and message integrity [4]. The message integrity in HTTPA is optional and HTTPA does not provide the message integrity of HTTP header.

Therefore depending on the security requirements of some web application, the application designers can choose to deploy their application on top of HTTP, HTTPS, or HTTPA. For example, the designers of some search engine, which requires no security, can deploy their engine on top of HTTP. On the other hand, the designers of some on-line banking application, which requires maximal security, can deploy their application on top of HTTPS after it is augmented with a password protocol.

Recently, however, a new class of web applications, which we refer to as "open applications", has emerged. And it turns out that the security requirements of open applications do not quite match the security guarantees provided by HTTP, HTTPS, or HTTPA. Examples of these open applications are news, social networking, and web blogging. The security requirements of these open applications are (1) server authentication, (2) message integrity, and in some cases (3) client authentication. These open applications, however, do not usually require message confidentiality. For example, users are vulnerable to many attacks by surfing only news web sites [5] with HTTP; Attackers can actively inject malicious JavaScripts [6], steal cookies by impersonation [7], and replay cookies [8].

Because neither HTTP nor HTTPA provides all the security requirements of open applications, these applications cannot be deployed on top of HTTP or HTTPA. Also, because open applications do not usually require the expensive requirement of message confidentiality, which is provided by HTTPS, deploying these applications on top of HTTPS is both an expensive and overkill proposition. Moreover, as explained below, because HTTPS is not compatible with middle boxes (such as cache proxies) in the Internet, deploying open applications on top of HTTPS prevents these applications from taking advantage of the middle boxes in the Internet.

Thus, in order to support the secure and efficient deployment of open applications in the web, we present in this paper a new transport protocol for the web, which we refer to as HTTP Integrity or HTTPI for short. HTTPI provides the two security guarantees of server authentication and message integrity. But it does not provide message confidentiality. Moreover, HTTPI can be augmented by a password protocol in order to provide the added guarantee of client authentication.

For convenience, Table I lists the security guarantees that are provided by each member of the HTTP family of protocols. Because the security requirements of open applications are server authentication, message integrity, and in some cases client authentication, it is clear from Table I that these applications are better deployed on top of HTTPI (with or without

978-1-4577-0638-7 /11/$26.00 ©2011 IEEE

TABLE I
SECURITY GUARANTEES OF HTTP, HTTPS, HTTPA, HTTPI

| | [1]SA | [2]CA | [3]MI | [4]MC |
|---|---|---|---|---|
| HTTP | | | | |
| HTTPS | ✓ | | ✓ | ✓ |
| HTTPS with Password | ✓ | ✓ | ✓ | ✓ |
| HTTPA | | ✓ | ✓ | |
| HTTPI | ✓ | | ✓ | |
| HTTPI with Password | ✓ | ✓ | ✓ | |

[1]SA: Server Authentication, [2]CA: Client Authentication
[3]MI: Message Integrity, [4]MC: Message Confidentiality

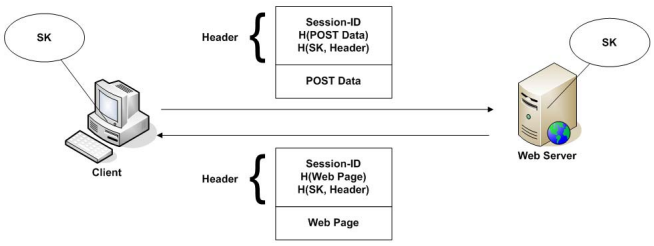



Fig. 1. Progression Phase in HTTPI

password protocol).

HTTPI is similar to work done independently and concurrently by Gaspard et al. as SINE [9][1]. SINE, however, does not provide comprehensive integrity like HTTPI. SINE does not provide the integrity for HTTP request messages, and HTTP header, but considers only the integrity of the contents in HTTP response messages. In doing so, SINE uses incremental signing by private key and hashing whereas HTTPI uses keyed-hashing with session key.

## II. DESIGN OF HTTPI

In order for a web client $C$ (i.e. a browser) to communicate with a web server $S$ using HTTPI, $C$ needs first to establish an HTTPI session with $S$. The established session is uniquely identified by two parameters:

i. A session id that is computed by $S$ and communicated to $C$ during the session establishment phase.
ii. A session (symmetric) key that is computed by both $C$ and $S$ during the session establishment phase.

The session id will be sent in the clear in all the HTTPI request and response messages that are exchanged between $C$ and $S$ in the established HTTPI session. The session key will remain private, known only to $C$ and $S$.

The HTTPI session is established over a TCP connection between a high-numbered TCP port in $C$ and the TCP port 80 in $S$.

In this section, we describe the three phases of an HTTPI session between $C$ and $S$: establishment, progression, and termination. Then, we describe the verifiable cookies which can be exchanged in each such session.

[1]The seminal work for HTTPI was done in 2009 and the initial version describes HTTPI protocol with more technical details [10].

*1) Session Establishment:* The web client $C$ and the web server $S$ use a TLS-like protocol to establish their HTTPI session. This protocol allows $C$ and $S$ to perform three tasks.

1. The web client $C$ becomes certain that it is indeed communicating with the right web server $S$. (Moreover, if a password protocol is added to the TLS-like protocol, then $S$ also can become certain that it is indeed communicating with the right web client $C$.)
2. The web server $S$ chooses a unique id for the session and communicates it to $C$. Server $S$ also chooses a future expiration time for the session.
3. Both $C$ and $S$ agree on a symmetric session key known only to $C$ and $S$.

At the end of the session establishment phase, the web client $C$ associates the following session entry with its TCP port for this session: (Session Id, Session Key, IP address of $S$, Session Cookies)
The last field "Session Cookies" in this entry is a pointer to all the cookies that are accumulated in $C$ during this session. (See Section II-4 below.) Also at the end of the session establishment phase, the web server $S$ adds the following session entry to its session table: (Session Id, Session Key, IP address of $C$, TCP port number in $C$, Session Expiration Time)

*2) Session Progression:* Once an HTTPI session between $C$ and $S$ is established, $C$ can start to send HTTPI request messages to $S$ and receive HTTPI response messages from $S$. Each HTTPI message, whether a request or response, is similar to an HTTP message with three exceptions:

a. The header of each HTTPI message has a new field, called Session Id, whose value is the id of the session in which this message is sent.
b. The header of each HTTPI message has another new field, called Header Hash. The value of this field is the result of applying the secure hash function SHA-1 to all the immutable fields in the message header with the session key [11].
c. If an HTTPI message has some "content", then the header of the message has the Content-MD5 [12] header field. The value of this field is the result of applying the secure hash function MD5 to message content. (Note that this field is optional in HTTP, but mandatory in HTTPI when the message has some content.)

An illustration of the message exchange between web client $C$ and web server $S$ during the session progression phase is shown in Fig. 1. If HTTPI request is GET request compared to POST request in Fig. 1, the GET request does not have POST Data and does not need to have the Content-MD5 header field, H(POST Data) in Header. We adopt Content-MD5 to hash contents because we want to make HTTPI compatible with HTTP standards and MD5 is sufficient for our purpose. However, MD5 can be replaced with SHA-1 or much secure hashing functions if MD5 can be easily broken in the future.

We apply keyed-hashing only to HTTPI headers for two reasons. First, web servers can precompute content hashing.

Web servers need to compute only header hashing on the fly by not causing significant overheads due to the small size of HTTPI headers if the requested contents have not been changed after the content hashing is computed. Second, middle boxes can cache contents. Content hashing can be cached with the contents in middle boxes if keyed-hashing is not used. Caching with middle boxes is explained in detail in section IV.

There are two kinds of header fields depending on the behavior of caching: (1) end-to-end header fields (2) hop-by-hop header fields [1]. End-to-end header fields are immutable header fields, but proxies might omit or modify them and HTTPI requires web clients and web servers to enumerate the hashed header fields as in OAuth [13].

The progression phase of an HTTPI connection between web client $C$ and web server $S$ can last for a long time, for example days or weeks, even if $C$ stops sending HTTPI request messages to $S$. This is in complete contrast with HTTPS connections whose progression phase is usually short, lasting only for minutes, and requires that $S$ continuously sends HTTPS request messages to $S$. This difference between HTTPI and HTTPS is due to the fact that HTTPI, unlike HTTPS, does not provide any confidentiality guarantees that can be threatened during the long progression phase even when web client $C$ leaves the established connection unattended.

In HTTPS, if a web client is disconnected from a web server, the web client must resume its session with fast handshake because the session id of TLS session can be sent only using TLS client hello message. On the other hand, even though a web client is disconnected from a web server in HTTPI, the web client can use its session instantly because HTTPI sends session id in HTTP header. Thus, the maximum number of sessions in HTTPI is decided only by the maximum size of sessions stored in the server whereas the maximum number of sessions in HTTPS is limited by the maximum number of connections. Thus, HTTPI is more scalable than HTTPS.

*3) Session Termination:* At the end of the progression phase of an established HTTPI session between client $C$ and server $S$, server $S$ proceeds to terminate the established session. It is also possible that server $S$ may decide to terminate the oldest established HTTPI session in its session table, before the expiration time of this session, when $S$ notices that the session table has become full or near full.

Server $S$ terminates an established HTTPI session with client $C$ by simply removing the session entry from the session table (in $S$) and by tearing down the TCP connection, between $C$ and $S$, over which this HTTPI session was established. Later, if $C$ sends to $S$ a request message in this HTTPI session, C will receive back a TCP reset message informing it that the TCP connection between $C$ and $S$ has been torn down. This will cause $C$ to tear down the TCP connection and remove the session entry associated with its TCP port. Note that in this design of HTTPI session termination, only servers can initiate session terminations. But this simple design can be easily extended to allow clients to initiate session terminations as well. The extension consists of introducing new HTTPI request-termination messages.

*4) Verifiable Cookies:* All the exchanged cookies between a web client $C$ and a web server $S$ in an established HTTPI session are "verifiable cookies". A verifiable cookie is an ordinary cookie with one additional field, called the Verifier. The value of this Verifier field is computed by $S$ as follows:

Verifier := H(Server Key, Session Key, Cookie)

where H is a secure hash function, say SHA-1, applied to the concatenation of three components:

i. Server Key: is a symmetric key that is known only to server $S$.
ii. Session Key: is the shared key between $C$ and $S$ for the established HTTPI session in which this verifiable cookie is to be sent.
iii. Cookie: is the content of the cookie to which this verifier is to be attached.

When server $S$ receives a verifiable cookie (part of an HTTPI request message) that is supposedly sent in some established HTTPI session, then server $S$ examines the verifier field in the cookie and verifies whether: (1) $S$ itself has generated this cookie earlier (since the verifier field is computed using the server key of $S$, which is known only to $S$) and (2) $S$ itself has sent this cookie earlier in the established HTTPI session (since the verifier field is computed using the session key of the established session).

## III. Defending against Cyber Attacks

If a web client $C$ uses HTTP to communicate with a web server $S$, then the communication between $C$ and $S$ can be interfered with or disrupted using any of the following four attacks: (1) Server Impersonation (2) Message Modification (3) Cookie Theft (4) Cookie Injection.

In this section, we argue that if client $C$ uses HTTPI (instead of HTTP) to communicate with server $S$ then none of these attacks can succeed in interfering with or disrupting the communication between $C$ and $S$. For convenience, Table II lists the cyber attacks that HTTPI can defend against and the security guarantees of HTTPI that can be used to defend against these attacks. Interested readers can find more details about the security guarantees of HTTPI and these four attacks in some detail in [10].

*1) Server Impersonation:* Assume that $C$ ends up with a wrong IP address, that belongs to a server $S'$ different from the intended server $S$. In this case, execution of the TLS protocol between $C$ and $S'$, in the establishment phase of the HTTPI session, will fail because $S'$ does not know the private key of $S$. And so the server impersonation attack will not succeed.

*2) Message Modification:* If any computer on the communication path between $C$ and $S$ modifies any message that is sent between $C$ and $S$, then the header hash field and the Content-MD5 field in this message will no longer be consistent with the rest of the message and the message ends up being discarded before it is delivered. (Note that the computer, that modified the message, cannot modify the header hash field and the Content-MD5 field in the message to make them consistent with the rest of the message. This is because this computer

TABLE II
CYBER ATTACKS THAT HTTPI CAN DEFEND AGAINST

| Cyber Attacks | Examples of Attacks | Security Guarantees to Defend Against Attacks |
|---|---|---|
| Server Impersonation | Drive-By Pharming [14]–[16], DNS rebinding [17], DNS cache poisoning [18] | Server Authentication |
| Message Modification | In-flight Page Change [19], ARP poisoning [20], [21] | Message Integrity from $C$ to $S$, and $S$ to $C$ |
| Cookie Theft | Side Jacking [8], Surf Jacking [7] | Cookie Integrity from $C$ to $S$ |
| Cookie Injection | Session Fixation [22] | Cookie Integrity from $S$ to $C$ |

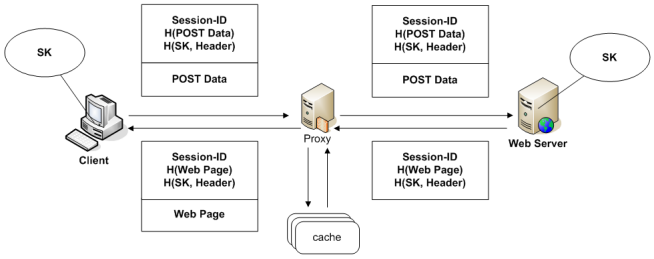



Fig. 2. Compatibility with Cache Proxies

does not know the session key for the current HTTPI session between $C$ and $S$.)

*3) Cookie Theft:* All the (verifiable) cookies, that are exchanged in the established HTTPI session between $C$ and $S$, are sent in the clear. Assume that a computer $C'$, located on the communication path between $C$ and $S$, copies all cookies that occur in the request messages from $C$ to $S$. Now if computer $C'$ later establishes an HTTPI session with server $S$ and pretends to be $C$ by using any of the cookies that it has copied from the former HTTPI session in the later HTTPI session, then the verifier field of this cookie will not be consistent with the session key of the current HTTPI session between $C'$ and $S$ and the cookie will be rejected. (This is because the session key of the later HTTPI session between $C'$ and $S$ is different from the session key of the former HTTPI session between $C$ and $S$.) In other words, cookies that are stolen from one HTTPI session cannot be replayed in a later HTTPI session and the cookie theft attack will fail. Note that the Date header field in HTTP requests ensures that HTTP requests can not be replayed. Furthermore, the Expires header field in HTTP responses can limit the time for HTTP responses to be replayed.

*4) Cookie Injection:* Note that each cookie injection attack starts with a successful server impersonation attack. But HTTPI can defend against server impersonation attacks, as discussed in Section III-1. Thus, if a client $C$ uses HTTPI to communicate with a server $S$, then client $C$ cannot be subjected to any cookie injection attack.

## IV. COMPATIBILITY WITH MIDDLE BOXES

We argued in [10] that HTTPI has several security advantages over HTTP: first HTTPI provides some security guarantees that cannot be provided by HTTP, and second HTTPI can defend against some cyber attacks that cannot be defended against by HTTP. This means that the World Wide Web needs to support HTTPI beside, or instead of, HTTP especially since we show in the next section that the performance of HTTPI is very close to that of HTTP.

On the other hand, one may argue that the web may not need to support HTTPI beside HTTPS since HTTPS can be used, instead of HTTPI, in those applications that only need HTTPI. But we refute this argument in two ways. First, we argue in this section that HTTPI is compatible with middle boxes, such as cache proxies and application firewalls, in the Internet whereas HTTPS is not. One might think that HTTPS can support cache proxies with dummy encryption. In fact, TLS supports a null cipher feature such as TLS_RSA_WITH_NULL_SHA or TLS_RSA_WITH_NULL_MD5 [3] though they are not used in practice. But, it requires web servers to compute the hashing for the entire HTTP response messages to every request because TLS hashing does not decouple HTTP headers and contents. On the other hand, HTTPI can precompute contents hashing and web servers do not have to compute HTTPI contents always, but web servers need to compute HTTPI header hash on the fly if the requested contents have not been changed after they are cached. Second, we show in the next section experiment results which demonstrate that the throughput of HTTPI is 37% higher than that of HTTPS and the CPU execution time of HTTPI is 23% lower than that of HTTPS. Therefore, there are significant performance gains to be had when HTTPI is used in place of HTTPS in those applications that do not require message confidentiality.

In the remainder of this section, we discuss how HTTPI is compatible with two important types of middle boxes: cache proxies and application firewalls.

*1) Compatibility with Cache Proxies:* Contents are most likely user-independent and header fields are user-dependent. If the Content-MD5 header field is precomputed, it can be used for every user who accesses the same page. On the other hand, header fields like Date, User-Agent, and Cookie are user specifics. Thus, the computation is minimized by decoupling header fields from contents. Moreover, [23] proposes to use the Content-MD5 header field for a strong cache validation and [24] uses the Content-MD5 header field to detect duplicate transfer.

Consider the case where a client $C$ uses HTTPI to communicate with a web server $S$, and assume that all the exchanged (request and response) messages between $C$ and $S$ reach a cache proxy $PR$ after they are sent and before they reach their ultimate destinations. Note that $PR$ does not know the session key for the current HTTPI session between $C$ and $S$. Yet, $PR$ can still read each request message from $C$ to $S$ (since none of the messages is encrypted) and determine whether or not the requested web page in the request message is already stored in $PR$.

There are two possible scenarios in this case:

1. If the requested web page is not in $PR$, then $PR$

forwards the request message to $S$. Later, when $S$ sends back the requested page in a response message, $PR$ stores a copy of the requested page in its memory before forwarding the response message to $C$.

2. If the requested web page is already in $PR$, then $PR$ applies the secure hash function MD5 to the page and sends the result along with the session ID (for the HTTPI session between $C$ and $S$) to server $S$. Later server $S$ computes the header hash for the requested page and sends it to $PR$. Then $PR$ prepares the response message, that has the requested web page and the header hash, and sends it to $C$. The exchanged messages in this scenario are illustrated in Fig. 2.

Note that in Scenario 1, the requested web page was sent all the way from $S$ to $PR$ then to $C$, whereas in Scenario 2, the requested page is sent only from $PR$ to $C$. Thus, the saving in communication is achieved since in most cases Scenario 2 is much more likely to occur than Scenario 1.

*2) Compatibility with Application Firewalls:* Consider the case where a client $C$ uses HTTPI to communicate with a web server $S$. Assume that all the request messages from $C$ reach a server firewall $SF$ before reaching $S$, and all the response messages from $S$ reach a client firewall $CF$ before reaching $C$.

When $SF$ receives a request message, it checks whether the cookies in the message header are correct (i.e. could have been sent earlier by server $F$), and whether the JavaScript code in the POST data of the message, if any, is harmless. Based on these checks, $SF$ decides either to forward the request message to server $S$ or to discard the message.

When $CF$ receives a response message, it checks whether the JavaScript code in web page in the message, if any, is harmless. Based on these checks, $CF$ decides either to forward the response message to client $C$ or to discard the message.

Note that the two application firewalls $SF$ and $CF$ can perform their functions, even though they do not know the session key for the current HTTPI session between $C$ and $S$, because none of the exchanged messages between $C$ and $S$ is encrypted.

## V. Experimental Results

In this section, we describe an experiment that we carried out to compare the performance of HTTPI against the performance of HTTP and of HTTPS, when HTTP and HTTPS are used in place of HTTPI.

This experiment involves a client machine and a server machine with Ubuntu 8,04. The client machine is an Intel Core 2 Duo CPU @ 3.16 GHz with 2 GB RAM. The server machine is an Intel Core 2 Duo CPU @ 3.00 GHz with 2 GB RAM. The client and the server machine are connected using a 100 Mb/second Ethernet. The server machine hosts an Apache version 2.2.11 server which supports both HTTP and HTTPS. We augmented this Apache server with a new module that implements HTTPI.

We made the augmented Apache server host three web pages, which we obtained from the web: an Amazon page, a Facebook page, and a New York Times page. The characteristics of these three pages are as follows:

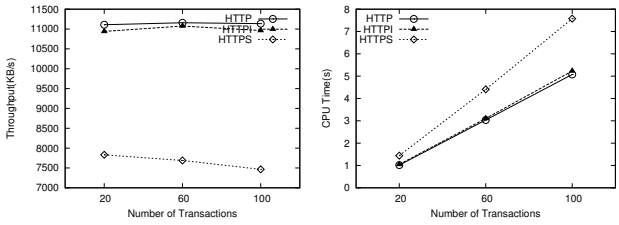
(1) The Amazon page, which belongs to the first author, consists of a container HTML page that has 172 KB, and 53 files of images, scripts, and style sheets totaling 484 KB.

(2) The Facebook page, which belongs to the first author, consists of a container HTML page that has 360 KB, and (2) 51 files of images, scripts, and style sheets totaling 1116 KB.

(3) The New York Times page, which is a recent front page, consists of a container HTML page that has 140 KB, and 94 files of images, scripts, and style sheets totaling 1040 KB.

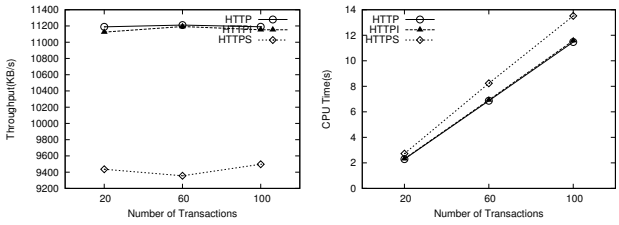
The experiment consists of three stages:

(1) In the first stage, the client machine communicates with the server machine using HTTPI and the two machines execute X transactions, where X = 20, 60, and 100 and a transaction consists of the client machine sending one request message and the server machine replying back with a response message that includes the requested web page (Note that the value of X is chosen to be relatively large since the lifetime of an HTTPI connection is intended to be relatively long as discussed in Section II-2 above).



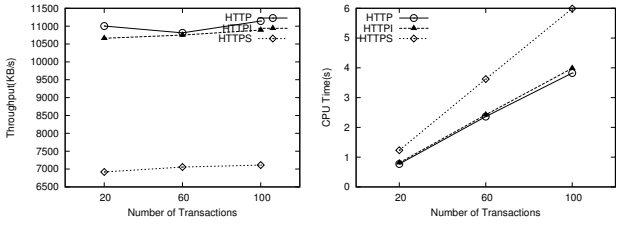

(a) Throughput (b) CPU Time

Fig. 3. Performance Comparison with Amazon Trace




(a) Throughput (b) CPU Time

Fig. 4. Performance Comparison with Facebook Trace




(a) Throughput (b) CPU Time

Fig. 5. Performance Comparison with NY Times Trace

(2) The second stage of the experiment is the same as the first stage except that the client machine and the server machine communicate using HTTP (instead of HTTPI).
(3) The third stage of the experiment is the same as the first stage except that the client machine and the server machine communicate using HTTPS (instead of HTTPI).

In each stage of the experiment we measured two parameters: network throughput (in KB/second) and CPU execution time (in seconds). We ran each stage 5 times and measured the average. The measured results of this experiment are shown in Fig. 3(a) to 5(b). Fig. 3(a), 4(a), and 5(a) show the network throughput when the requested web page is Amazon, Facebook, and New York Times, respectively. Fig. 3(b), 4(b), and 5(b) show the CPU execution time when the requested web page is Amazon, Facebook, and New York Times, respectively. From these figures, we conclude that the throughput of HTTPI is within 1.2% from that of HTTP and is about 37% better than that of HTTPS. We also conclude that the CPU time of HTTPI is within 1.9% from that of HTTP and is about 23% better than that of HTTPS.

From these results, we conclude that there is a significant performance gain that can be achieved by using HTTPI, instead of HTTPS, when message confidentiality is not required. Therefore, supporting HTTPI beside HTTPS in the web seems to be a reasonable design option.

For completeness, we repeated our experiment when the client machine and the server machine are connected using a 1 Gb/second fast Ethernet. The results show the same trends as those in Fig. 3, Fig. 4, and Fig. 5.

## VI. Concluding Remarks

There are two significant advantages of using HTTPI over using HTTPS: (1) Our experimental results, in Section V, showed that the throughput of HTTPI is almost 40% better than that of HTTPS (2) As discussed in Section IV, HTTPI is compatible with, and can take full advantage of, the middle boxes in the Internet. By contrast, HTTPS is not compatible with, and cannot utilize any of the middle boxes in the Internet.

Our experimental results in Section V also showed that the throughput of HTTPI is within 1.2% of that of HTTP. Therefore, if HTTPI happens to replace HTTP as the baseline transport protocol over the web, then the reduction in throughput can go unnoticed by most web users. On the other hand, the improvement of security (e.g. as discussed in Section III, HTTPI can defend against Pharming attacks but HTTP cannot) can be appreciated and cheered by all users.

Finally, we observe that the relationship between HTTPI and HTTPS is analogous to the relationship between the IP Authentication Header (AH) [25] and the IP Encapsulation Security Payload (ESP) [26] in IPsec [27]. Thus, just as both AH and ESP are supported by IPsec, both HTTPI and HTTPS should be supported by the web.

## References


[1] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1," RFC 2616 (Draft Standard), Jun. 1999, updated by RFC 2817. [Online]. Available: http://www.ietf.org/rfc/rfc2616.txt

[2] E. Rescorla, "HTTP Over TLS," RFC 2818 (Informational), May 2000. [Online]. Available: http://www.ietf.org/rfc/rfc2818.txt

[3] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," RFC 5246 (Proposed Standard), Aug. 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5246.txt

[4] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart, "HTTP Authentication: Basic and Digest Access Authentication," RFC 2617 (Draft Standard), Jun. 1999. [Online]. Available: http://www.ietf.org/rfc/rfc2617.txt

[5] R. Saltzman and A. Sharabani, "Active man in the middle attacks," OWASP AU 2009, Feburuary 2009.

[6] A. Madrigal, "The Inside Story of How Facebook Responded to Tunisian Hacks," the Atlantic, 2011.

[7] S. Gauci, "Surf jacking - "https will not save you"," http://enablesecurity.com/2008/08/11/surf-jack-https-will-not-save-you, August 2008.

[8] R. Graham, "Sidejacking with hamster," http://erratasec.blogspot.com/2007/08/sidejacking-with-hamster_05.html, August 2007.

[9] C. Gaspard, S. Goldberg, W. Itani, E. Bertino, and C. Nita-Rotaru, "SINE: Cache-Friendly Integrity for the Web," in *Proceedings of IEEE International Conference on Network Protocols (ICNP) Network Protocol Security (NPSec) Workshop*, Oct. 2009.

[10] T. Choi and M. G. Gouda, "HTTP Integrity: A Lite and Secure Web against World Wide Woes," Department of Computer Science, The University of Texas at Austin, Tech. Rep. TR09-41, December 2009.

[11] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication," RFC 2104 (Informational), Feb. 1997. [Online]. Available: http://www.ietf.org/rfc/rfc2104.txt

[12] J. Myers and M. Rose, "The Content-MD5 Header Field," RFC 1864 (Draft Standard), Oct. 1995. [Online]. Available: http://www.ietf.org/rfc/rfc1864.txt

[13] E. Hammer-Lahav, "The OAuth 1.0 Protocol," RFC 5849 (Informational), Internet Engineering Task Force, Apr. 2010. [Online]. Available: http://www.ietf.org/rfc/rfc5849.txt

[14] S. Stamm, Z. Ramzan, and M. Jakobsson, "Drive-by pharming," in *Proceedings of 9th International Conference on Information and Communications Security (ICICS)*, 2007, pp. 495–506.

[15] Z. Ramzan, "Drive-by pharming in the wild," http://www.symantec.com/connect/blogs/drive-pharming-wild, January 2008.

[16] ——, "DNS Pharming Attacks Using Rogue DHCP," http://www.symantec.com/connect/blogs/dns-pharming-attacks-using-rogue-dhcp, December 2008.

[17] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh, "Protecting Browsers from DNS Rebinding Attacks," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007. [Online]. Available: http:///crypto.stanford.edu/dns/dns-rebinding.pdf

[18] US-CERT, "Multiple dns implementations vulnerable to cache poisoning," http://www.kb.cert.org/vuls/id/800113.

[19] C. Reis, S. D. Gribble, T. Kohno, and N. C. Weaver, "Detecting in-flight page changes with web tripwires," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2008, pp. 31–44.

[20] K. Zhang, "ARP spoofing HTTP infection malware," http://securitylabs.websense.com/content/Blogs/2885.aspx, December 2007.

[21] B. Zdrnja, "Massive arp spoofing attacks on web sites," http://isc.sans.org/diary.html?storyid=6001, March 2009.

[22] M. Kolšek, "Session fixation vulnerability in web-based applications," www.acros.si/papers/session_fixation.pdf, December 2002.

[23] M. Nottingham, "Inherent http coherence," http://www.mnot.net/papers/coherence.html.

[24] J. C. Mogul, Y. M. Chan, and T. Kelly, "Design, implementation, and evaluation of duplicate transfer detection in http," in *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2004, pp. 4–4.

[25] S. Kent, "IP Authentication Header," RFC 4302 (Proposed Standard), Dec. 2005. [Online]. Available: http://www.ietf.org/rfc/rfc4302.txt

[26] ——, "IP Encapsulating Security Payload (ESP)," RFC 4303 (Proposed Standard), Dec. 2005. [Online]. Available: http://www.ietf.org/rfc/rfc4303.txt

[27] S. Kent and K. Seo, "Security Architecture for the Internet Protocol," RFC 4301 (Proposed Standard), Dec. 2005. [Online]. Available: http://www.ietf.org/rfc/rfc4301.txt