HTTPi for Practical End-to-End Web Content Integrity

Kapil Singh
IBM Research

Helen J. Wang
Microsoft Research

Alexander Moshchuk
Microsoft Research

Collin Jackson
Carnegie Mellon University

Wenke Lee
Georgia Institute of Technology

Abstract

Widespread growth of open wireless hotspots has made it easy to carry out man-in-the-middle attacks and impersonate web sites. Although HTTPS can be used to prevent such attacks, its universal adoption is hindered by its performance cost and its inability to leverage caching at intermediate servers (such as CDN servers and caching proxies) while maintaining end-to-end security.

To complement HTTPS, we revive an old idea from SHTTP, a protocol that offers end-to-end web integrity without confidentiality. We name the protocol HTTPi and give it an efficient design that is easy to deploy for today's web. In particular, we tackle several previously-unidentified challenges, such as supporting progressive page loading on the client's browser, handling mixed content, and defining access control policies among HTTP, HTTPi, and HTTPS content from the same domain. Our prototyping and evaluation experience show that HTTPi incurs negligible performance overhead over HTTP, can leverage existing web infrastructure such as CDNs or caching proxies *without* any modifications to them, and can make many of the mixed-content problems in existing HTTPS web sites easily go away. Based on this experience, we advocate browser and web server vendors to adopt HTTPi.

1 Introduction

The same-origin policy [31] (SOP) is the key access control policy for the web and browsers. SOP defines a principal model where web sites are mutually distrusting principals [36, 37], and where one site's script cannot access another site's content. However, the authenticity of the principal and the integrity of its content are often at question since much of the web is delivered over HTTP rather than HTTPS. Consequently, network attackers can carry out man-in-the-middle attacks and undermine browsers' access control, even if browsers flawlessly implement the enforcement of the same-origin policy. Such attacks are highly practical today with the prevalence of wireless hotspots and insecurity in the DNS infrastructure [17]. The web requires *end-to-end security* to allow meaningful SOP enforcement in browsers.

HTTPS [29] can prevent network attacks. However, its universal adoption is hindered by two deficiencies of HTTPS: its performance cost and its uncacheability at intermediate servers. Regarding performance, although GMail has recently demonstrated the ability of serving HTTPS content with low overhead using commodity hardware [24], a general applicability of their solution to other SSL setups is still not clear [3]. Web caching offers significant benefits to web sites and users. It enables web sites to save bandwidth costs and to reduce latency for users by outsourcing infrastructure to CDNs and offloading requests to CDN servers. Although CDNs do offer services for HTTPS content [6], this carries the cost of trusting CDN servers to be man-in-the-middle and losing end-to-end security. Furthermore, such services come with a hefty charge of up to $3,000 per month plus bandwidth costs [14]. Caching proxies can also deliver web content significantly faster to large user communities behind gateways or firewalls, such as mobile users. HTTPS content cannot take advantage of these proxies at all today. We observe that much of the web is

cacheable (Section 7.1), and we expect significant growth in cacheable web content as rich media proliferates [2]. To achieve an end-to-end secure web, HTTPS is definitely not the complete answer.

Fortunately, end-to-end security, cacheability, and performance are not at conflict inherently. End-to-end security encompasses (1) end-to-end authentication (i.e., content comes from the right origin[1]) (2) end-to-end content integrity (i.e., content is not tampered with), and (3) end-to-end content confidentiality (i.e., content is kept private). For the browser platform to meaningfully enforce its access control policy, only authentication and integrity are needed, and confidentiality is *not* required. Without confidentiality, the content can be cached at intermediary web servers. HTTPS provides all three properties simultaneously and is hence not cacheable.

To complement HTTPS and address its deficiencies, we advocate a new protocol, called *HTTPi*, which offers end-to-end web integrity without confidentiality. While this is an old idea which first appeared in the signature mode of operation in SHTTP [30] (which was a proposal that unsuccessfully competed with SSL and HTTPS in 1999), the development of this idea has stayed at the algorithmic level. In this paper, we give it an efficient and practical system design and implementation that can be easily deployed in today's web. In particular, we tackle several *previously-unidentified challenges*, such as supporting progressive page loading on the client's browser, designing browsers' policies for handling of mixed end-to-end secure and insecure content, and defining access control policies among HTTP, HTTPi, and HTTPS content from the same domain.

We have built an end-to-end prototype to evaluate HTTPi. On the browser side, we implemented the HTTPi protocol for Internet Explorer (IE) using IE's Asynchronous Pluggable Protocol extension mechanism. On the server side, we implemented HTTPi support in IIS 7 as well as in an HTTP proxy that can be placed in front of origin web servers. While HTTPi requires both browser and server-side modifications, our design does not require changes at intermediate nodes, such as CDN servers or caching proxies.

Our measurements indicate that HTTPi incurs an acceptable verification and one-time signing overhead with our unoptimized implementation. This cost is quickly amortized over many requests; for example, a typical web server deployed on Amazon EC2 achieved a 4.06x higher throughput for static content served over HTTPi (and signed offline) than over HTTPS, and HTTPi's throughput is negligibly lower than that of HTTP.

To evaluate the caching benefits that HTTPi brings to web sites, we measured cacheability on today's web. We found significant cacheable content from both HTTP and HTTPS, which makes HTTPi compelling. With much of existing HTTPS content being cacheable, that content can be safely turned into HTTPi content for better performance and for the ability to be offloaded to other servers without any loss of security. In fact, many existing HTTPS sites contain HTTP content including scripts and images. Such mixed-content pages often contradict a site's intent to defend against network attackers. This is precisely due to the cost of enabling HTTPS for such existing HTTP content. It would be much easier to turn HTTP content contained on HTTPS sites into HTTPi content, which will achieve the end-to-end security desired by these sites.

2 Design Overview

We set the following goals for the HTTPi design:

- *Guarantee of end-to-end integrity*: Our design ensures that the integrity of the rendered content is always maintained. For example, a network attacker will not be able to inject or remove content, or have adverse impact on browser-side rendering of content.
- *Easy adoption*: HTTPi should fit seamlessly into the current web infrastructure. In other words, the design should be transparent to the intermediate web servers (such as CDN servers and HTTP web

[1]The binding between the origin and its public key is provided by certificate authorities. Client authentication is at the discretion of web sites.

proxies) and should involve minimal changes to servers and browsers.
- *Negligible overhead over HTTP*: The design should incur negligible overhead over HTTP in computation, bandwidth, and user-experienced latency.

Note that intermediate servers could legitimately need to modify web content, such as for personalization or content filtering in enterprises. Transmitting content over HTTPi instead of HTTP would prevent such modifications. We argue that the guarantee of integrity must be end-to-end, and any intermediate modifications should be explicitly approved by the one of the endpoints (for example, by sharing the private and public key pair of an endpoint).

To guarantee end-to-end integrity and to minimize latency and overhead, we use a content-signature-based scheme that allows progressive content loading and at the same time is robust to any injection attacks, as described in Section 3. In Section 4, we present our design on how HTTPS, HTTPi and HTTP content can be mixed together and how web sites can express restrictions on mixed content to browsers. In Section 5, we design the access control policy that browsers should enforce across HTTPS, HTTPi and HTTP content.

To ease adoption, we implement HTTPi over the existing HTTP protocol so that intermediate web servers can cache HTTPi content seamlessly. Web browsers can show "httpi" in the address bar, but the messages on the wire speak HTTP. We use a new `Integrity` header to indicate the use of HTTPi as the protocol. The integrity header also carries the signature for HTTP headers (excluding the integrity header itself, of course). We use the existing `Strict-Transport-Security` header to prevent stripping attacks (Section 3.3) and the existing `X-Content-Security-Policy` header to allow web sites to configure mixed content policies (Section 4). Signatures for the HTTP response body are in-band in the body itself. Our server-side and client-side implementation of HTTPi uses public interfaces, requiring no modifications to core functionality of the server or the browser (Section 6).

3 Support for Progressive Content Loading

A protocol scheme that ensures message integrity needs to satisfy two requirements. First, the identity of the server sending the content needs to be authenticated and second, the content needs to verified for integrity. HTTPi uses a content-signature-based protocol scheme to satisfy these requirements.

In a strawman design, HTTPi could sign the hash of an *entire* HTTP response: The server first creates a cryptographic hash (e.g., SHA1) of the whole response and then signs the hash using the server's private key. The hash and its signature are then passed to the client along with the response. At the client side, the browser waits for the entire response to arrive, calculates its hash, and compares the value with the signed hash to authenticate the server and verify the response.

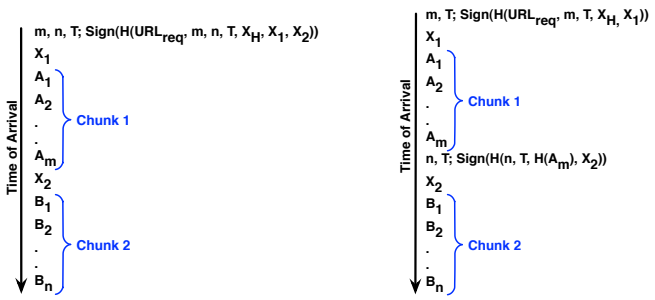
A key limitation of this design is that the browser would have to wait for the entire response to arrive before being able to verify the content integrity and dispatch the content for rendering. Consequently, this would disrupt the existing progressive content loading mechanisms in browsers, servers, and the HTTP protocol, and the user would experience longer delays before seeing any content.

We designed HTTPi to allow progressive content loading by using HTTPi segments. Before describing this design, we first provide some background.

3.1 Existing Progressive Content Loading Mechanisms

Current browsers support progressive loading of web content, attempting to render data as soon as it arrives from the network. The amount of data available at a time is determined by the underlying TCP congestion control and network conditions as well as server load. HTTPS content can also utilize progressive loading, especially a site uses a stream cipher.

Complementing browsers' progressive content loading, servers may start sending the response before completing the processing of a request, and therefore before knowing the entire response body. To this end,



(a) (b)

Figure 1: Protocol Scheme in HTTPi for (a) static content (b) dynamic content. $A_1, A_2, ..., A_m$ and $B_1, B_2, ..., B_n$ represent segments for Chunk 1 and 2, respectively. X_1 and X_2 represent concatenated hashes evaluated over the segments of Chunk 1 and 2, respectively. X_H represents concatenated hashes over all HTTP headers. URL_{req} is the requested URL, and T is the timestamp. The figure shows the order in which content is sent in the response body.

servers often use HTTP chunked transfer encoding [16] and encode each piece of available response data in a chunk, which consists of its own length and body. A web server typically uses chunked encoding in two scenarios: (1) content is static, but its retrieval (e.g., from the server database) or processing is slow, and (2) content is dynamically generated with a chunk being a logical boundary. Chunks are sent as soon as they are available, inband in the body of the same HTTP response. Note that a particular chunk may not arrive at the client in one shot, but possibly be further broken into pieces due to network congestion. Nevertheless, the browser can consume partial chunks progressively.

3.2 HTTPi Segments for Progressive Content Loading

In HTTPi, the key challenge in supporting progressive content loading is to configure a sensible granularity of content verification. This design must (1) leverage browsers' progressive content loading; (2) be compatible with HTTP chunked transfer encoding; (3) be resilient to the underlying TCP congestion control, which cannot be predicted by servers in an offline fashion; (4) allow cacheability; and (5) incur low overhead. We use the term *HTTPi segment* to refer to the unit of verification in HTTPi. Let S denote the size of an HTTPi segment.

Using HTTP chunk size as S would still be too coarse-grained. An HTTP chunk can be arbitrarily large, leading to same problems as with the strawman solution described above.

If a server could predict how much data arrives at its clients, it could enable verification for just that data. For a single, live connection, a server can indeed do this by obtaining the current TCP congestion control window size and the receiver window size from the network layer. However, because of dynamic network conditions, such prediction would not work well for requests at different times or from different users and would defeat cacheability. Therefore, S needs to be a constant.

We choose to use the typical TCP segment size (1400 bytes) for S. TCP segment is the unit of TCP transfer. The rationale here is that the browser will need to wait for *at most* one packet to arrive to receive a full HTTPi segment, perform the verification and render the segment. This wait is as minimal as it can get.

Although an HTTPi segment is the unit of verification, it does not need to be the unit of signing. In our design, we amortize the signing cost over multiple segments in the response body. In more detail, whenever a web server has some HTTP response data ready (whether it is the entire HTTP response or an HTTP chunk becoming available), for every S bytes, we take a hash, then we compute the signature for multiple hashes concatenated in the right sequence. For HTTP headers, we hash each header and use a single signature over these hashes, since browsers do not consume partial header values. We further amortize the signing cost by signing the hashes of HTTP headers along with the hashes of HTTP content

using a single signature. We put content hashes and signatures inband with the response body. We rejected putting signatures and hashes in an HTTP header, because our scheme needs to support HTTP's chunked encoding, where chunks after the first chunk do not carry headers. For responses using chunked encoding, we maintain the `chunked-encoding` header, prepend HTTPi's metadata (such as signatures and hashes) to each chunk body, and recalculate each chunk's length. This preserves compatibility with intermediate proxies.

The application determines signing granularity based on whether the content being signed is known in advance (i.e., static content), or is generated on the fly (i.e., dynamic content). Figure 1 illustrates our protocol. In Figure 1(a), for static content, we amortize the cost of signing by using a single signature over segments for all chunks generated (e.g., X_1 and X_2 in a single signature). For dynamic content, the hashes are computed at the time of content generation. The signature is calculated over all the segments of a single chunk (Figure 1(b)). The sequence of hashes for the headers (X_H) is placed only in the first signature. We also place the URL of the requested page (URL_{req}) in the first signature and the current timestamp (T) in each signature as a preventive measure for certain attacks (Section 3.3).

Note that for static content, signing can be done offline. For dynamic content, signing incurs a computation overhead of one SHA1 computation per 1400 bytes, resulting in the bandwidth overhead of just 1.4% (20/1400). The signature overhead is one signature per chunk for dynamic content. We will show in Section 7 that much of the web is static and cacheable, and that HTTPi incurs negligible overhead over HTTP.

Any segment that fails the integrity check is not rendered. In such cases, the browser informs the user about the integrity failure and removes the security indicator from the page. For JavaScript, we do not perform progressive content loading because today's JavaScript engines require an entire script to be received before starting its execution.

3.3 Security Analysis and Design Enhancements

Out-of-sequence Segments. The segment hashes are arranged in a sequence before signing. If a network attacker tries to reorder the segments, it will break the hashes sequence and signature verification would fail.

Injection and Removal Attacks. Attacker will not be able to launch injection attacks successfully because the injected content will not be verified by the browser. Removal attacks cannot happen to the segment group of a signature for the same reason.

Nevertheless, removal attacks can happen across signature groups (a set of chunks for static content or a single chunk for dynamic content). When HTTP chunks are used by a server, each signature group will have a set of HTTPi segments and a signature for them. A network attacker can remove a signature group without being noticed at the client. To address this issue, we insert the hash of the last segment of the previous chunk at the beginning of the hash sequence of the current chunk (Figure 1(b)); and we insert the header hash at the beginning of the hash sequence of the first chunk.

Content Replay. Network attackers could mix-and-match old content and new content to cause disruptions. Our design prevents such attacks by placing timestamp T in each signature. For HTTPi responses involving multiple signatures, the browser must verify that the timestamp is the same across all signatures.

The network attackers could alternatively replay a completely different response for a requested resource. To thwart this attack, the browser verifies its own value of the requested URL against the signed URL_{req} value.

Stripping Attacks. Both HTTPS and HTTPi are prone to "stripping" attacks that hijack a user's initial insecure HTTP request and remove redirects to secure content. Although it is possible to notice stripping attacks by manually checking browser security indicators, users often ignore these indicators [32]. The HTTP Strict Transport Security protocol (HSTS) prevents these attacks by allowing web sites to specify a minimum level of security expected for connections to a given server. The policy can be delivered

via an HTTP header [21]. To prevent attacks on the user's first visit to the site, the policy can also be delivered via DNSSEC [23]. We use an extension to HSTS, `allowHTTPi`, to allow servers to specify HTTPi as the minimum level of security. The `allowHTTPi` token is appended to the server's existing `Strict-Transport-Security` declaration. Older browsers that do not support HSTS will ignore this header, while older browsers that support HSTS but not our extension will default to HTTPS for all content.

Denial of Service. HTTPi is vulnerable to denial-of-service attacks where a network attacker strips off the integrity header from the response, preventing the content from being rendered. Alternatively, the attacker can allow some but not all response segments through to the browser. This could potentially corrupt the application's internal logic. For example, the attacker could strip off JavaScript that alters page layout. One possible countermeasure is to use a timeout for inter-segment arrival at the client and raise an integrity failure alert when this timer expires. However, this requires estimating typical inter-arrival time for each client, which may not be reliable. We choose to allow the browser to wait infinitely for packets to arrive. If the user clicks on stop, we alert the user that the content is incomplete. Since we do not execute JavaScript until it is fully received, partially rendered JavaScript would not be an issue for the integrity of the site.

4 Mixed-Content Treatment in Browsers

The mixed content condition occurs when a web developer references an insecure (HTTP) resource within a secure (HTTPS) page. This puts the privacy and integrity of the otherwise secure page at risk, because the insecure content could be modified by a network attacker. Scripts are particularly problematic because they run with priveleges of the including page, and malicious scripts could read or alter content delivered over a secure connection.

Browsers differ in how they handle mixed content. IE 8 and below prompt the user before displaying mixed content, while Firefox and Chrome show a modified browser lock icon. From a security standpoint, the best behavior would be to block all insecure content in secure pages without prompting the user. The latest release of IE9 enforces this behavior on scripts and stylesheets, but not images; this policy is similar to the one proposed by Gazelle [37]. However, automatically blocking insecure content carries serious compatibility implications. It might confuse the user, since pages relying on insecure resources could appear broken. Worse, the user might think that a broken page indicates a bug in the browser and switch to an older version or to a completely different browser to fix it.

We argue that mixed-content vulnerabilities should be fixed by web developers, both for security and user-experience reasons. Web developers have a better understanding of impact that embedded content can have on their site's security. They are also in a better position to develop a user-friendly fallback mechanism in case some content fails a security check and is not rendered.

By default, we require that all active content embedded in HTTPi and HTTPS pages, such as scripts and stylesheets, be rendered over HTTPi or HTTPS. To allow web applications to customize this default behavior, we use an HTTP header that is compatible with the Content Security Policy (CSP) [34] header to specify the server's end-to-end integrity requirements for dependent resources. The CSP policy syntax is convenient for our purposes, as it already allows sites to specify which origins they want to include content from. An example policy may be:

```
X-Content-Security-Policy:
     allow https://login.live.com
           httpi://*.live.com:443
```

The policy above informs the browser that all embedded resources from `login.live.com` should be retrieved over HTTPS and content from all other subdomains of `live.com` needs to be downloaded over HTTPi. If the servers hosting the embedded content do not support the corresponding protocol, then the content is considered unsafe as per the web page's requirements and hence should not be rendered by the

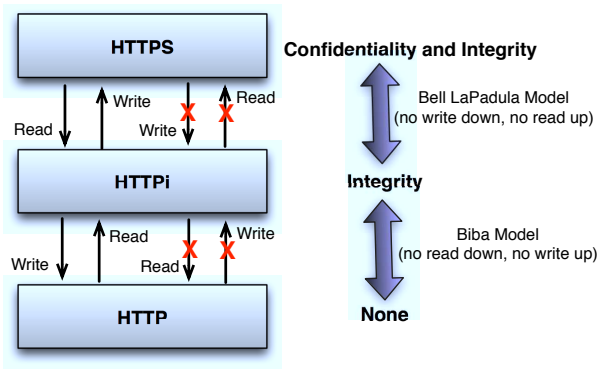


Figure 2: Interactions in Mixed Content Rendering.

browser. Our design also supports finer-grained integrity requirements, i.e., at the level of resource types or specific resources themselves. However, specifying such finer policies must be done sensibly, as it increases bookkeeping at the server and could break existing interactions within embedded content if policies are specified incorrectly.

The CSP syntax provides an excellent way for web developers to handle mixed content, and it does not require changes to web application code. The latter would require explicitly modifying *all* insecure references of embedded resources, which is not only time-consuming but also difficult to do correctly. A secure (HTTPS or HTTPi) URL could return a redirect to an insecure resource, which could be difficult to detect by examining the DOM alone. Moreover, a script delivered over a secure channel could still make references to insecure content. In our design for HTTPi, the browser enforces policies specified by CSP for all statically or dynamically generated URLs.

5 Access control across HTTPS, HTTPi, and HTTP content

HTTPi content can be embedded in an iframe through the use of the "httpi" scheme, such as `<iframe src='httpi://a.com/'>`, or through the use of an additional iframe "integrity" attribute, such as `<iframe src='http://a.com/' integrity>`. The former presentation is consistent with other protocol schemes. The latter has the benefit of safe fallbacks for backward compatibility; on an older browser, HTTPi content would simply render as HTTP content[2]. Note that regardless of representation, the network messages are still sent over HTTP to be backward compatible with the existing web caching infrastructure.

The same-origin policy labels principals with the origin defined as the triple of `<protocol, domain, port>` [36, 37]. Therefore, content from the same domain and port number but with different protocols is rendered as separate principals, which can only communicate explicitly through messages (i.e., `postMessage` [7]).

In this section, we consider the default interaction and access control model for HTTPS, HTTPi, and HTTP content served from the *same* domain and port. For example, a top-level HTTPi page may embed two iframes, one containing HTTP content and the other containing HTTPS content; and all three pages are from the same domain and port. Here, following SOP is safe, but it disallows all interaction among HTTP, HTTPi, and HTTPS content. Instead of direct function calls or accesses to DOM objects, developers would be forced to layer such interaction on top of asynchronous `postMessage`-based protocols, which carries extra marshalling costs and may be hard to design correctly, as illustrated by recent flaws found in Facebook Connect and Google Friend Connect [20]. Consequently, a developer may be discouraged from converting some content on an HTTPS site into HTTPi to benefit from its cacheability properties.

[2] A site could also specify an HTTPS URL in the 'src' attribute to use HTTPS fallback

As a concrete example, consider an online shopping site that is rendered over HTTPS to protect users' private data such as credit card information. The site presents users with a map to select a site-to-store pick-up location during checkout. It may be desirable to deliver the store information and map content over HTTPi, but this raises a problem of allowing the HTTPS part of the site to read the store selection made by the user, an interaction that would be disallowed by SOP. As a result, the site's developers may be forced to refactor their code to use `postMessage`.

We observe that the SOP semantics are more restrictive than actually required to ensure security for such scenarios. Our goal is to allow legitimate communication while preserving the security semantics, namely the confidentiality and/or integrity, of the rendered data. Our default communication policies are inspired by the combination of the Bell LaPadula [9, 10] and Biba [11] models. It is important to note that our goal is *not* to enforce information flow invariants often associated with those models (e.g., frames of any origin can already freely communicate via `postMessage`), but rather to use these models to determine a secure and convenient *default* isolation policy for our setting. We summarize these models as the following set of rules:

Bell LaPadula model (for confidentiality):

- The Simple Security Property: a subject at a given security level may not read an object at a higher security level (no read-up).
- The *(star) property: a subject at a given security level must not write to any object at a lower security level (no write-down).

Biba model (for integrity):

- The Simple Integrity Axiom: a subject at a given level of integrity may not read an object at a lower integrity level (no read down).
- The * (star) Integrity Axiom: a subject at a given level of integrity must not write to any object at a higher level of integrity (no write up).

In view of these models, we represent the three protocols (HTTP, HTTPS and HTTPi) by two confidentiality levels (C_{high} and C_{low}) and two integrity levels (I_{high} and I_{low}), which models the high and low requirements for confidentiality and integrity, respectively. HTTPS can be realized by the tuple $<C_{high}, I_{high}>$, HTTPi by $<C_{low}, I_{high}>$ and HTTP by $<C_{low}, I_{low}>$. Using this model, we define the access control rules across HTTP, HTTPi, and HTTPS as follows:

HTTPS and HTTP. HTTPS' confidentiality label C_{high} is higher than HTTP's confidentiality level C_{low}, thus resulting in "no read up, no write down" requirement of the Bell LaPadula model. The integrity levels of HTTPS and HTTP, I_{high} and I_{low} respectively, with $I_{high} > I_{low}$, results in "no write up, no read down" condition of the Biba model. Combining these two requirements results in no reads or writes to either side being allowed between HTTPS and HTTP. This derivation is consistent with the SOP.

HTTPi and HTTP. Since confidentiality levels of HTTPi and HTTP are equal, only the integrity levels enforce the "no write up, no read down" policy from the HTTPi content to HTTP resources (Figure 2). Firstly, this implies that a script belonging to the HTTPi principal can write to the HTTP part of the page without reading its content. One reason to prevent an HTTPi script from reading HTTP content is to prevent the HTTP input from influencing the logic within the HTTPi content. However, an HTTPi script might still desire to read the HTTP page to identify the DOM element to write to. So, our requirement is to allow the read operation on the HTTP content without allowing the logic of HTTPi content from being affected. One way to realize this is by performing complete information flow check in the HTTPi code, which might not be practical. We use an alternative approach in which the HTTPi content itself writes the code for reading the HTTP content, and this code is injected into the HTTP content. This injected code runs within the HTTP principal and hence can freely read and write to the content. Since HTTPi relinquishes the transferred code to the HTTP integrity level (I_{low}), that code cannot affect the logic of HTTPi's own code, though it still can read from HTTPi content. Secondly, HTTP can read the HTTPi content, but cannot write to it. We realize this in our design by providing only a shadow copy of the HTTPi content to HTTP, with no direct reference

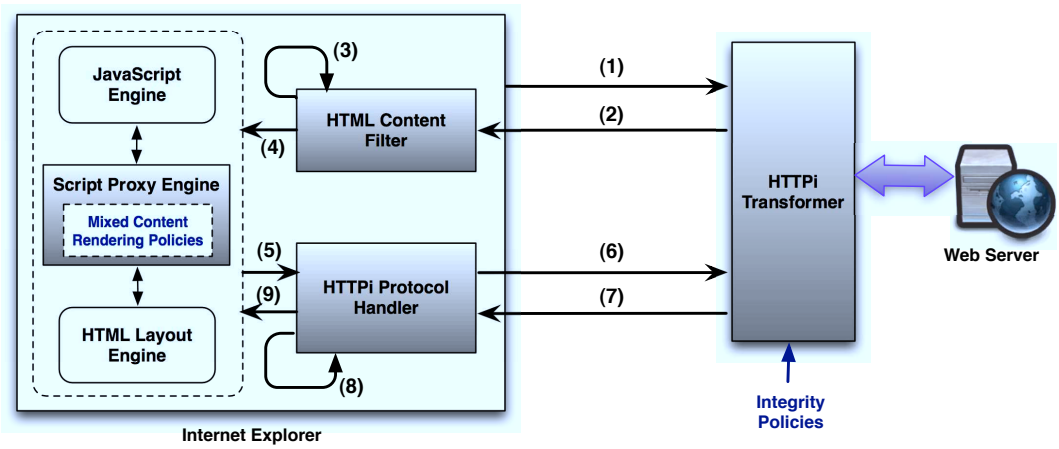


Figure 3: High-Level Architecture of our HTTPi Implementation with the operational steps to retrieve content over HTTPi as follows: (1) IE makes an initial request for a specific page. (2) Server-side proxy identifies that the request is for a HTTPi-enabled resource and appends integrity policy headers to the response. (3) HTML content filter processes the response by modifying URLs that match STS policies to point to their corresponding HTTPi links. (4) HTML content filter releases the modified response to IE's rendering engine. (5) The HTTPi protocol handler is invoked for every HTTPi object encountered during rendering. (6) The HTTPi protocol handler makes a HTTP call to the server requesting the object. (7) Server-side proxy traps the request, makes an independent HTTP call to the backend web server to get the response, hashes and signs the response, and returns it back to the HTTPi protocol handler. (8) The HTTPi protocol handler verifies the signature and hashes corresponding to the different segments in the response. (9) Successfully verified segments are passed to the rendering engine for progressive loading. The Script Engine Proxy (SEP) subsequently mediates all mixed-content interactions while a web page renders.

to real HTTPi objects.

HTTPS and HTTPi. Since HTTPS and HTTPi integrity levels are equal, only the confidentiality levels force the "no read up, no write down" rule from HTTPS to HTTPi resources (Figure 2). Both read and write operations can be realized similarly to the previous scenario. We allow HTTPi content to write to HTTPS since the code for HTTPi is at the same integrity level as HTTPS content and written by the same developers. HTTPi scripts can write the code for reading the HTTPS content into the HTTPS' DOM and effectively, that code becomes part of the HTTPS principal. This allows reading of the HTTPS code by the injected code without leaking any of the read data back to HTTPi's main code. For reading HTTPi content without allowing any write, a shadow of HTTPi's DOM is provided to HTTPS. Coming back to the shopping site example earlier in this section, this rule would allow HTTPS content to read the store selection made by the user and correspondingly send the merchandise to the selected store.

6 Implementation

HTTPi requires both browsers and web servers to adhere to the protocol. Accordingly, our implementation consists of server-side and client-side modules. Figure 3 shows the high-level architecture of our system. Our server-side implementation consists of an HTTPi Transformer, which implements content hashing, segmentation, and inclusion of the integrity policy requirements for HTTP responses. Our client-side implementation consists of three modules that we add to IE8: (1) the *HTML content filter* transforms a given page to adhere to integrity policy requirements, (2) the *HTTPi protocol handler* processes incoming HTTPi content, and (3) the *Script Proxy Engine* provides JavaScript and DOM interposition to enforce our mixed-content access control policies. In this section, we describe how we implemented these modules and the

challenges we faced. Overall, our implementation consists of 1,100 lines of server-side code, and 3,500 lines of client-side code.

6.1 Server-side Implementation

We completed two implementations of the server-side component of HTTPi to explore two deployment tradeoffs. First, we extended the IIS 7 web server with a C# HTTPi Transformer module that encapsulates the functionality to generate HTTP responses with signatures and content hashes that adhere to HTTPi. Although we chose IIS, similar module functionality is available for other web servers. This option is useful if a server is willing to immediately integrate HTTPi functionality into its current setup. It also has obvious performance benefits as the module is closely coupled with the server.

In our second deployment option, we integrated the HTTPi Transformer into a web proxy that translates typical HTTP responses into HTTPi responses by embedding all the hashes and signatures needed by HTTPi. We leveraged the public-domain Fiddler debugging proxy [25] and its FiddlerCore [15] extensibility interfaces. This option is independent of web server implementation and allows servers to continue supporting HTTP as the delivery protocol for backward compatibility, while switching to HTTPi for select requests that pass through the proxy. This eases deployment: the proxy can be deployed anywhere in the network and guarantees integrity between the proxy and a compatible browser. This could be desirable for corporations that do not require integrity checks for intranet users, but want to ensure integrity of their sites for external users.

For our evaluation, we used the proxy deployment, because (1) it allowed us to test our prototype against publicly deployed web sites without having any control of their web servers, and (2) it allowed fair comparison of HTTPi with HTTPS and HTTP (Section 7.2.3) by cleanly switching to a desired protocol between the client and the proxy even when the backend server did not support the protocol.

6.2 Client-side Implementation

6.2.1 Filtering content to enable HTTPi

The browser will issue HTTPi requests when it (1) encounters a link (e.g., when processing an embedded resource) with an `httpi://` scheme, or (2) when the Strict Transport Security policy or Content Security policy require a request to use HTTPi (see Sections 3.3 and 4). For (2), our HTML Content Filter modifies every HTML response associated with STS or CSP to ensure that it adheres to the minimum security levels specified in those policies. For example, resource links requiring integrity are transformed to `httpi://` links by modifying the URL scheme. The HTML Content Filter is invoked before the browser renders a page, thus ensuring that the HTTPi protocol handler will be called for all resources requiring integrity during rendering. We implemented this module by using IE's public MIME filter COM interfaces [1] and subsequently registered it as a filter for HTML content.

One limitation of this approach is that it may miss dynamically-generated links where the URL is constructed by JavaScript at runtime. We are currently solving this by performing HTTPi redirection at the time of actual HTTP requests; our evaluation is independent of this implementation enhancement and was performed without it.

6.2.2 HTTPi Protocol

The HTTPi protocol handler encapsulates all client-side handling of HTTPi content and is triggered when the browser's rendering engine encounters an HTTPi link. Upon invocation, it makes an independent HTTP call to the server to retrieve the content. It then verifies the integrity of the content in segments using

the algorithms described in Section 2. Once the integrity of a particular segment is verified, its content is released to the renderer for progressive loading.

We implemented this module as an asynchronous pluggable protocol (APP) [1] IE module associated with the `httpi://` scheme. While IE conveniently provides this generic protocol extension point, its usage carries a performance cost. IE's internal logic is well-optimized for HTTP and HTTPS but not for APP; this makes a comparably performant APP protocol difficult to implement. Considerable time was spent on making our code as optimal as possible by parallelizing various operations such as network read and signature verification. Despite our limited knowledge of IE's internal optimizations and with the handicap of using a generic interface, we were still able to achieve acceptable performance as compared to HTTPS and HTTP (Section 7.2.3).

6.2.3 Access control for mixed content

Another big implementation challenge was to customize SOP to enforce our mixed-content access control policies. Unfortunately, IE does not allow changing the code for SOP with public APIs. As a result, the only alternative was to implement our solution as an additional layer on top of the existing SOP and then find a way to enforce mixed-content policies within the limits imposed by existing SOP logic. This certainly complicated our implementation.

To solve this problem, we use two steps. First, we modify the security origin (defined as the tuple `<protocol, domain, port>`) of all web page objects to change the protocol field to HTTP, i.e., the one with the lowest integrity and confidentiality level. We achieve this by providing a custom implementation for the `IInternetProtocolInfo` interface [4] from within the APP for HTTPi. Note that changing the security origin of an element does not affect the URL associated with that element. Per SOP, all the objects on the page can now interact without restriction. Our second step is to enforce access control rules that govern such interactions. We build on earlier work [33, 36] that implements a JavaScript engine proxy (called Script Engine Proxy or SEP): SEP is installed between IE's rendering and script engines, and it mediates and customizes DOM object interactions. SEP is implemented as a COM object and is installed into IE by modifying IE's JavaScript engine ID in the Windows registry. We extend SEP to trap all reads and writes across the page's objects and ensure that our mixed-content access control polices (Section 5) are enforced. We use the URLs associated with the accessing object and the object being accessed in making an access control decision. In summary:

- If the original origins of the caller and callee objects differ in `domain` and/or `port`, the browser prevents any interactions across them as per SOP.
- If the original origins of the caller and callee objects differ in only `protocol`, the SOP would allow the objects to interact (as we modify the protocol field of the security origin to HTTP). In this case, we mediate the interaction within our customized SEP to enforce our access control policies.

The read operation is straightforward: SEP allows the caller to have read access to the callee's objects. The write operation could be implemented in a similar fashion; however, some writes must first access an object to which the write subsequently occurs. For example, if the caller wants to write to a specific element on a callee object, it might need to read the handle to that element using functions such as `getElementById` or `getElementsByName`. However, if the caller only has write privileges with no read access, it cannot make such calls.

We solve this problem by introducing a new JavaScript function `writeUsingCode`, which is interpreted by our SEP implementation; the browser's JavaScript engine does not need to understand this function. Instead of directly making read calls looking for an element of the callee object, the caller uses the function to pass JavaScript code that encapsulates such read calls and the subsequent write call to the corresponding element. The SEP intercepts `writeUsingCode` and makes calls to the underlying JavaScript engine to execute the code with the origin of the callee object. Any unintended feedback mechanism intro-

Protocol	Total Objects		Publicly Cacheable Objects	
	Count	Size	Count	Size
HTTP	346,629	1532 MB	251,826 (72.65%)	1385 MB (90.41%)
HTTPS	5,036	21.95 MB	3,659 (72.66%)	19.39 MB (88.33%)

Table 1: Measurement of publicly cacheable web content from the top 1000 Alexa sites.

duced by this code is prevented by SEP's access control policies.

7 Evaluation

We have implemented a HTTPi system that works end-to-end. We used our proxy-based implementation as a server-side HTTPi endpoint to verify our system for correctness against a number of popular web sites, such as Google, Bing Maps, and Wikipedia. In each case, the browser successfully rendered the web pages and all integrity checks were correctly included at the server side and verified at the browser. Any tampering of the web page in the network was correctly detected and failed the integrity check at the browser. We evaluated the access control interactions for mixed content by developing a set of custom web pages that included such interactions. Our system correctly enforced the access control policies for such interactions.

Next, we provide experimental evidence to support our claim that today's web sites can benefit from cacheability enabled by HTTPi. To this end, we first perform a web cacheability study to answer two questions: (1) what web sites have cacheable content, and (2) what users are taking advantage of shared caches on the web. Next, we evaluate the performance of our HTTPi prototype and compare its overhead to that of HTTPS and HTTP.

7.1 Study of Web Cacheability

With HTTPi, web sites decide what content uses HTTPi as the underlying transport mechanism. Therefore, any content that web sites currently allow to be cached by intermediate web servers, such as CDNs and web caches, becomes an ideal target for HTTPi. To better estimate the amount of such content, we performed a cacheability analysis on the top 1,000 Alexa sites that includes both top-level pages and content embedded on the sites visited. We analyze the HTTP caching headers, such as `Cache-control`, `Expires`, or `Pragma`, to decide what content is deemed cacheable [16].

Experimental Setup. To facilitate automatic analysis for a large number of URLs, we used a customized crawler from earlier work [33], which utilizes IE's extensibility interfaces to completely automate the browser's navigation. To invoke functionality beyond a site's home page, the crawler uses simple heuristics that simulate some user interaction, such as clicking links and submitting simple forms with junk data. We restrict all simulated navigations to stay within the same origin as a site's home page. We monitor the browser's network traffic in a proxy to intercept all HTTP/HTTPS requests and analyze HTTP headers relevant to web caching. The proxy is included as a trusted certificate authority at the browser to allow it to inspect HTTPS content [25].

Prevalence of cacheable content. Table 1 shows the results of our web cacheability experiment. Note that our results only consider content that is marked as public and excludes any private content that is user-specific and hence is intended to be cached only at the user's browser. As we can observe from the table, a large majority (98%) of web objects are served over HTTP. We found that approximately 73% of these objects are cacheable. The cacheability is even higher (90%) when considering content size instead of object count, indicating that web applications typically want larger-sized content, such as images, to be cached in the network. The limited number of HTTPS objects that we encountered follow a similar trend, with a large number (73%) being cacheable objects. The presence of a considerable number of public, cacheable HTTPS

objects is an indication that web applications intend to cache objects in the web, but are discouraged by the lack of security in HTTP. They are left with no choice but to trust the CDNs for this type of content. When only integrity of such content is desired, HTTPi is an ideal alternative for these HTTPS objects.

Note that even though there is considerably less HTTPS content than HTTP in our study, cacheability benefits of HTTPi over HTTPS would still be significant given that (1) we have measured the most popular sites that many people visit (thus, significant benefits from shared caching), and (2) this will take much load off HTTPS servers (as 73% of the objects are cacheable). Moreover, 73% of objects served over HTTP could benefit from improved integrity under HTTPi (for example, map tiles and scripts for Google maps).

Presence of in-network caches. To see how many users are benefiting from web caches today, we measured the prevalence of forward caching proxy servers, which are a significant source of in-network caching. More specifically, we conducted an experiment to determine how the country and the user agent affects whether a forward network proxy is being used. We used rich media web ads as a delivery mechanism for our measurement code, using the same ad network and technique previously demonstrated in [22]. We spent $80 to purchase 115,031 impressions spread across 194 countries. Our advertisement detected forward proxies using XMLHttpRequest to bypass the browser cache and store content in the network cache. Overall, 3% of web users who viewed our ad were using a caching network proxy. However, some countries had a significantly higher fraction of users behind network proxies. Popular countries for forward proxies included Kuwait (63% of 372 impressions), United Arab Emirates (61% of 624 impressions), Argentina (11% of 1,875 impressions), and Saudi Arabia (10% of 4,248 impressions). We also observed higher usage of forward proxy caches (11%) among mobile users, although these users accounted for only 0.1% of the total impressions in our experiment.

Relevance to HTTPi. Our results demonstrate that caching proxies are still prevalent and useful today, particularly for some large communities, such as a whole country of people behind a single firewall and mobile users behind cellular gateways. HTTPi can take advantage of these proxies while offering end-to-end security at the same time.

7.2 Performance Evaluation of HTTPi

We evaluate the performance of HTTPi in two steps. First, we microbenchmark various stages of the protocol and analyze parameters that influence HTTPi's performance. Second, we determine the end-to-end performance overhead of HTTPi over HTTP and HTTPS protocols.

7.2.1 Experimental Overview

Ideally, we would run performance experiments on real deployed web sites. However, current web servers do not understand the HTTPi protocol, and many servers host an HTTP version of a site but not HTTPS. To overcome this, we used our modified server-side Fiddler [25] proxy (Section 6.1) for proxying all requests from the client to the backend server, and converting HTTP requests from the origin server into HTTPi or HTTPS requests to the client, as necessary for our experiments. This setup allows us to measure the cost of using HTTPS and HTTPi for web pages that are currently hosted over HTTP.

We use the end-to-end response time as the measurement criterion, defined as time between the instance at which a URL is submitted at the browser and the instance at which the corresponding page is fully rendered. To remove any discrepancies that might arise due to inconsistent network conditions, we deduct the data fetching time at Fiddler from the total end-to-end response time. This gives us an estimate of the end-to-end response time with Fiddler acting as the web server. For a fair comparison, we also perform similar deductions for HTTP and HTTPS.

For our experiments, we use SSL certificate size of 1024 bits. Even though there is a push on the Internet to move towards 2048-bit certificates, many popular sites such as Gmail still use 1024-bit keys.

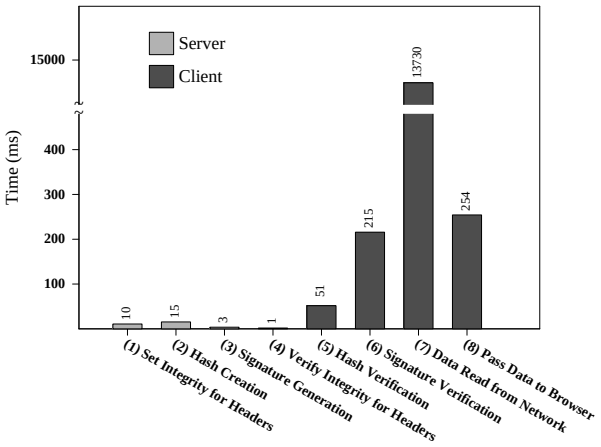


Figure 4: Microbenchmarks for various HTTPi operations (836KB web page, 512Kbps network bandwidth)

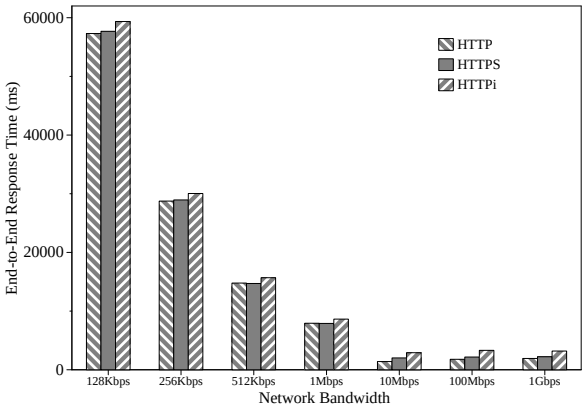


Figure 5: End-to-end response time as a function of network bandwidth available to the client, measured for a 836KB page. These results do *not* include performance benefits of caching for HTTP and HTTPi.

Additionally, it makes HTTPi's performance estimates conservative in comparison to HTTPS, as HTTPS will perform worse for 2048-bit keys.

Using the Akma network delay simulator v0.9.129 [5], we simulated various network conditions to understand their performance impact on end-to-end response time. We provided incoming and outgoing connections with equal bandwidth and fixed their queue sizes at 20 packets. We ran our delay simulator on the server side to cap the server throughput to a desired bandwidth. We deploy our server-side Fiddler code on a Windows 7 machine, with an Intel 2.67 GHz Core i7 CPU and 6 GB of RAM. The client runs on a Windows 7 machine, with an Intel 2.4GHz quad-core CPU and 4GB of RAM. All results are averaged over 10 trial runs.

7.2.2 Microbenchmarks

To understand sources of overheads in our system, we instrumented our HTTPi implementation to measure latencies of various operations, and used a simulated network bandwidth of 512Kbps to load an 836KB HTML page in our HTTPi-enabled browser, with the size picked to maximize measurable overhead and to observe effects of HTTPi's segmentation. Figure 4 breaks down the delays contributing to the end-to-end response time, which we measured to be 15.7 sec.

We find that a large fraction of the total time is spent reading content from the network (bar 7 in Figure 4), which is an expected behavior for slower networks. The overhead costs of hashing all content segments (bar 2) and signing these hashes with a 1024-bit key (bar 3) on the server side is very small. Here, the RSA signature is calculated on a fixed-size single SHA1 hash of 20 bytes (Section 2); this takes just 3ms. Since the header value sizes are much smaller as compared to the content body, both the time to set the header integrity content (hashing and signing) on the server (bar 1) and time to verify it on the client side (bar 4) is low.[3] On the client side, the signature verification time (215 ms, bar 6) is a more significant source of overhead. It is considerably higher than the cumulative hash verification time for all content segments (51 ms, bar 5), supporting our design of using a single signature over multiple segment hashes. The time to pass data from our client-side HTTPi protocol handler into the browser's rendering engine (bar 8) is also considerable; although it is not specific to HTTPi and would also be incurred by other protocol

[3]Note that we do not perform any segmentation for headers. For our measurements, we specify two headers, `Server` and `Content-Type`, to require integrity. This time cost will vary according to the number of headers for which integrity checksum is set.

Experiment	HTTP	HTTPi	HTTPS
Bare-metal Setup	3320	3318	2503
Amazon EC2 Setup	2757	2732	678

Table 2: Impact of HTTPi and HTTPS on server throughput in responses/sec.

handlers in the browser, native protocols like HTTP are more optimized in IE for this step, as we discussed in Section 6.2.2.

In summary, we find that the major HTTPi components (bars 1-6) constitute only 295 ms (1.8%) of the end-to-end response time for this microbenchmark, with largest overhead coming from client-side signature verification.

7.2.3 Comparing HTTPi to HTTP and HTTPS

In this section, we compare HTTPi's performance to that of HTTP and HTTPS and answer two questions: (1) Is the user-perceived latency acceptable for the data received over HTTPi, and (2) What is the performance impact of running HTTPi and the hashing and signing load it incurs on a web server?

User-perceived latency. We compared the end-to-end response time for our 836KB test page rendered over HTTPi, HTTPS and HTTP. Figure 5 shows the results of our experiments performed for different network bandwidths. Note that the performance results do not include caching, and only show the first of potentially many requests for this page. Evaluating performance of a particular cache is not the goal of our experiments; this has been well studied previously [38, 39]. We see that HTTPi incurs minimal overhead over both HTTP and HTTPS, and this overhead is consistently within 0.7-2.0 seconds over both HTTP and HTTPS for different network bandwidths. Since this value does not vary much with network bandwidth, we believe our implementation is fairly successful in matching the network optimizations of HTTPS and HTTP. We believe that there is still ample room for client-side optimization as we discussed earlier in Section 6, and this will certainly reduce the total overhead of HTTPi (since our microbenchmarks showed that client-side overhead is small but not negligible).

Web Server Throughput. Our server throughput measurements are performed using httperf [26], an HTTP performance measurement tool. The experiments are performed using two different setups representative of real-world web site deployments:

- Our first setup consists of an IIS server that is hosted on a bare-metal Windows 7 machine, with Intel 2.67 GHz Core i7 CPU and 6 GB of RAM. The Linux client machine running httperf is connected to the server by a 1Gbps network with negligible latency.
- Our second setup is cloud-based; we use a virtual Windows 2008 Server image on Amazon EC2. At the time, this image was the only publicly available image that came pre-installed with IIS 7. It is a "high-CPU medium" instance with 5 EC2 compute units with 1.7 GB of RAM (the fastest instance that was available for this image). This setup mimics a typical EC2 user who wants to host a web server. httperf is executed from a Linux EC2 instance in the same region, using EC2-private LAN with negligible latency.

We use an experimental HTML page of size 4.8 KB, which represents a typical size of a page with no embedded links. We arrived at this page size based on the web estimates that put total page size at 170KB (median) and number of objects per page at 37 (median) [27]. For each page, we increased the offered load on the server until the number of sustained sessions peaked. We found that the server was CPU-bound in all cases. Each session simulated one request to the web page.

Table 2 shows a summary of our results. HTTPi incurs negligible degradation (less than 1%) of throughput compared to the original HTTP page. In comparison, the throughput drop was substantial when using HTTPS, with our bare-metal experiment reporting 25% and EC2 experiment showing 75% drop in the

throughput. This drop is attributed to the heavy CPU load for the SSL handshake. Our bare-metal experiment shows a smaller drop since it has a considerably faster CPU, which handles this load better. Overall, these results demonstrate that web servers can have a significant performance incentive to use HTTPi instead of HTTPS.

8 Related Work

Prior work has explored a number of integrity protection techniques. A proposal on authentication-only ciphersuites for PSK-TLS [12] describes a transport layer security scheme for authentication and integrity, with no confidentiality guarantees. However, this proposal requires a shared secret between each client and the server to key the hash, making it impractical to share the key with all the clients of the application. Our work builds on SHTTP [30]'s proposed signature mode of operation and gives it a practical design for today's web by addressing the previously-unidentified challenges of progressive content loading, mixed content handling, and access control across HTTP/HTTPi/HTTPS.

Web tripwires [28] verify the integrity of a page by matching it against a known good representation (either a checksum or an encoded copy of the page's HTML), using client-side JavaScript to detect in-flight modifications. However, web tripwires have a high network overhead (approximately 17% of the page size), which could hinder the end-to-end response time, especially for slower networks. Moreover, web tripwires can be disabled by an adversary, and they cannot detect full-page substitutions. In contrast, HTTPi has a much lower overhead, and it is cryptographically secure and can prevent any type of integrity breaches. Fundamentally, web tripwires focus on *detection*, while HTTPi focuses on both *detection* and *prevention*.

Other research has proposed cryptographic schemes for web content integrity [8,18,19]. While we share some commonality with these works in integrity computation, our system differs in three significant ways. First, our design is more robust against attacks like stripping and content replay. Second, we design HTTPi to be practical for today's web and address problems such as mixed content treatment, compatibility with "chunked" transfer encoding, and access control across HTTP/HTTPi/HTTPS content, none of which are considered in prior work. Third, we go beyond algorithmic design and also offer a full practical implementation and evaluation of HTTPi for a real-world browser, while earlier research lacks any implementation details.

Stubblefield et al. [35] proposed mechanisms to improve SSL's performance. While their WISPr system shares HTTPi's motivation of supporting in-network caching while preserving integrity, it is designed for another content delivery protocol (subscription-based), rather than for use in existing web sites. WISPr constructs an HTTP page that embeds the encrypted version of the original page; this page can be cached in the network. However, a client needs to download a key from the server in order to decrypt the content, and WISPr only works for static content. In contrast, HTTPi is readily compatible with existing web sites, it supports static *and* dynamic content, and it adds support for progressive loading and mixed-content scenarios common on the web. Whereas no evaluation details are provided for WISPr, we showed that HTTPi is practical in Section 7.

HTTP provides a Content-MD5 header [16] that can carry the MD5 signature of the complete page. This header could be useful in providing basic page integrity, but suffers from many weaknesses if used by itself. For example, network attackers can modify the header since it is not authenticated, or they could completely drop the header without the client knowing. In contrast, HTTPi provides authentication by signing content hashes, and since it specifies the requirements for a page using HSTS in advance, the client can detect whether content requiring integrity is dropped by network attackers. Additionally, with HTTPi, integrity is evaluated over smaller-sized segments, which has better performance than Content-MD5's entire-page approach.

The YURL [13] specification defines an alternate server identification and authentication mechanism that does not depend on centralized authorities like the DNS or PKI. A YURL identifies a site using the

site's public key fingerprint and the web site owner owns the CA fingerprint. However, like HTTPS, and unlike HTTPi, the proposed YURL-based protocol *httpsy* [13] precludes content from being cached at web proxies.

9 Conclusions

We envision HTTPi to complement HTTPS to bring end-to-end security to the entire web. Only when there is end-to-end security, the browser platform and the web are able to have a collectively secure overall system.

We advocate the part of the web that does not have end-to-end security today to adopt HTTPi which incurs negligible performance overhead over HTTP and enjoys the benefit of CDNs and caching proxies just as HTTP. Our study indicates that a significant portion of existing HTTPS content is cacheable and can gain performance and caching benefits by employing HTTPi.

10 Acknowledgments

We would like to thank Shai Herzog and Gil Shklarski for their insightful discussions and support. We are also grateful to Carl Edlund, Jim Fox, Justin Rogers, and Ali Alvi for their help with IE instrumentation.

References

[1] About Asynchronous Pluggable Protocols. `http://msdn.microsoft.com/en-us/library/aa767916(v=VS.85).aspx`. Accessed on May 1, 2011.

[2] Cisco Visual Networking Index: Forecast and Methodology, 2009-2014. `http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-481360_ns827_Networking_Solutions_White_Paper.html`. Accessed on May 1, 2011.

[3] Dispelling the New SSL Myth. `http://devcentral.f5.com/weblogs/macvittie/archive/2011/01/31/dispelling-the-new-ssl-myth.aspx`. Accessed on May 1, 2011.

[4] IInternetProtocolInfo interface. `http://msdn.microsoft.com/en-us/library/aa767874(VS.85).aspx`. Accessed on May 1, 2011.

[5] Network Simulator. `http://www.akmalabs.com/downloads_netsim.php`. Accessed on May 1, 2011.

[6] I. Akamai Technologies. Secure content deliver. `http://www.akamai.com/dl/feature_sheets/fs_edgesuite_securecontentdelivery.pdf`.

[7] A. Barth, C. Jackson, and J. C. Mitchell. Securing Frame Communication in Browsers. In *Proceedings of the 17th USENIX Security Symposium*, San Jose, CA, July 2008.

[8] R. J. Bayardo and J. Sorensen. Merkle Tree Authentication of HTTP Responses. In *Special Interest Tracks and Posters of the 14th International Conference on World Wide Web (WWW)*, Chiba, Japan, May 2005.

[9] D. E. Bell. Looking Back at the Bell-LaPadula Model. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)*, Tucson, AZ, Dec. 2005.

[10] D. E. Bell and L. J. LaPadula. Secure Computer Systems: Mathematical Foundations. Technical Report ESD-TR-73-278, MITRE Corporation, Bedford, MA, Nov. 1973.

[11] K. J. Biba. Integrity Considerations for Secure Computer Systems. Technical Report ESD-TR-76-372, MITRE Corporation, Bedford, MA, Apr. 1977.

[12] U. Blumenthal and P. Goel. RFC 4785: Pre-Shared Key (PSK) Ciphersuites with NULL Encryption for Transport Layer Security (TLS), 2007.

[13] T. Close. Petname Tool: Enabling Web site Recognition using the Existing SSL Infrastructure. In *W3C Workshop on Transparency and Usability of Web Authentication*, New York, NY, Mar. 2006.

[14] K. DeGrande. CDNetworks, September 2010. Personal communication.

[15] FiddlerCore. `http://fiddler.wikidot.com/fiddlercore`.

[16] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC2616: Hypertext Transfer Protocol – HTTP/1.1, 1999.

[17] S. Friedl. An Illustrated Guide to the Kaminsky DNS Vulnerability. `http://unixwiz.net/techtips/iguide-kaminsky-dns-vuln.html`.

[18] C. Gaspard, S. Goldberg, W. Itani, E. Bertino, and C. Nita-Rotaru. SINE: Cache-Friendly Integrity for the Web. In *Workshop on Secure Network Protocols (NPSec)*, Princeton, NJ, Oct. 2009.

[19] R. Gennaro and P. Rohatgi. How to Sign Digital Streams. In *Proceedings of the 17th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO)*, Santa Barbara, CA, Aug. 1997.

[20] S. Hanna, R. Shin, D. Akhawe, P. Saxena, A. Boehm, and D. Song. The Emperor's New APIs: On the (In)Secure Usage of New Client-side Primitives. In *Web 2.0 Security and Privacy (W2SP 2010)*, 2010.

[21] J. Hodges, C. Jackson, and A. Barth. Http strict transport security (HSTS), 2010. `http://tools.ietf.org/html/draft-hodges-strict-transport-sec`.

[22] 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 (CCS)*, Alexandria, VA, Oct. 2007.

[23] V. Jirasek. Overcoming man in the middle attack on Strict Transport Security, August 2010. `http://blog.jirasek.eu/2010/08/overcoming-man-in-middle-attack-on.html`.

[24] A. Langley, N. Modadugu, and W.-T. Chang. Overclocking SSL. In *Velocity: Web Performance and Operations Conference*, Santa Clara, CA, June 2010. `http://www.imperialviolet.org/2010/06/25/overclocking-ssl.html`. Accessed on May 1, 2011.

[25] E. Lawrence. Fiddler Web Debugging Tool. `http://www.fiddler2.com/fiddler2/`. Accessed on May 1, 2011.

[26] D. Mosberger and T. Jin. httperf—A Tool for Measuring Web Server Performance. *Performance Evaluation Review*, 26(3):31–37, 1998.

[27] S. Ramachandran. Let's make the web faster. `http://code.google.com/speed/articles/web-metrics.html`.

[28] 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 (NSDI)*, San Francisco, CA, Apr. 2008.

[29] E. Rescorla. RFC 2818: HTTP Over TLS, 2000.

[30] E. Rescorla. and A. Schiffman. RFC2660: The Secure Hypertext Transfer Protocol, 1999.

[31] J. Ruderman. Same Origin Policy for JavaScript. `http://www.mozilla.org/projects/security/components/same-origin.html`. Accessed on May 1, 2011.

[32] S. Schechter, R. Dhamija, A. Ozment, and I. Fischer. The emperor's new security indicators. In *Proceedings of the 28th IEEE Symposium on Security and Privacy*, Oakland, CA, May 2007.

[33] K. Singh, A. Moshchuk, H. J. Wang, and W. Lee. On the Incoherencies in Web Browser Access Control Policies. In *Proceedings of the 31st IEEE Symposium on Security and Privacy*, Oakland, CA, May 2010.

[34] S. Stamm, B. Sterne, and G. Markham. Reining in the Web with Content Security Policy. In *Proceedings of the 19th International World Wide Web Conference (WWW)*, Raleigh, NC, Apr. 2010.

[35] A. Stubblefield, A. D. Rubin, and D. S. Wallach. Managing the Performance Impact of Web Security. *Electronic Commerce Research*, 5:99–116, January 2005.

[36] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and Communication Abstractions for Web Browsers in MashupOS. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, Oct. 2007.

[37] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The Multi-Principal OS Construction of the Gazelle Web Browser. In *Proceedings of the 18th USENIX Security Symposium*, Montreal, Canada, Aug. 2009.

[38] J. Wang. A Survey of Web Caching Schemes for the Internet. *SIGCOMM Computer Communication Review*, 29:36–46, October 1999.

[39] A. Wolman, G. M. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. M. Levy. On the Scale and Performance of Cooperative Web Proxy Caching. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, Charleston, SC, Dec. 1999.