iHTTP: Efficient Authentication of Non-Confidential HTTP Traffic

Abstract. HTTPS is the standard protocol for protecting information sent over the World Wide Web. However, HTTPS adds substantial overhead to servers, clients, and networks [1, 2]. As a result, website owners often pass on HTTPS and resort to only HTTP for hosting websites, leaving clients and servers vulnerable to attacks [3, 4]. Techniques have been proposed to only enable authentication and integrity of HTTP (response) data [2, 5–7]. However, they all suffer from vulnerabilities and poor performance. In this paper, we propose iHTTP, a new approach for enabling lightweight, efficient authentication and verification of HTTP (response) data. We adaptively handle different data encodings to allow for better performance without effecting user experience. We introduce a novel technique, Sliding-Timestamps, to allow iHTTP clients to authenticate the freshness of response data to prevent replay attacks and amortize signing costs. We also introduce Opportunistic Hash Verification to reduce client public key operations required to authenticate full web pages. We show in our experimental evaluation that iHTTP provides similar performance to HTTP, and higher throughput and lower maximum response time than HTTPS for Client-Static data.

1 Introduction

HTTP [8] is the most popular protocol used to construct the World Wide Web because it is lightweight, flexible, and scalable. However, HTTP provides no security protection and as a result technologists have accepted HTTP over TLS/SSL (i.e., HTTPS) [9] as the standard for providing authentication, integrity, and confidentiality while sacrificing being lightweight, flexible, and scalable. Other security protocols such as SHTTP [10] and HTTTPA [11] have also been proposed; however, they suffer from similar flexibility and scalability problems. As a result, data owners often choose HTTPS to provide data confidentiality and HTTP to provide efficient data delivery.

While HTTP data is non-confidential, the integrity of HTTP data is still very important. Recent research has shown that the lack of data integrity can have averse effects on website owners and clients. For example, Vratonjic et al. provided a case study of ad frauds in which HTTP web content was modified on the fly to include or rewrite advertisements. The study estimated that a WiFi hot spot with 100 users could raise $49,400 in annual revenue via HTTP ad content rewriting [3]. Research by Stamm et al. demonstrated an attack on HTTP called "Drive-By Pharming", where a malicious javascript reconfigures a victim router's DNS to redirect clients to fake pages. As a result, attackers are able to launch phishing attacks or present bogus data to clients [4]. Finally, the Bahama Botnet was caught providing counterfeit searches to victims, which altered ads and organic search results allowing for click frauds [12].

The discussion above highlights the distinct need to protect even non-sensitive HTTP content from malicious modification. A simple solution is to enable HTTPS for all websites. However, HTTPS adds substantial overhead to both the server and the clients [1]. Furthermore, HTTPS does not support network caching, which is known to significantly improve performance [13] and reduce upstream bandwidth usage [2]. It is expected that the number of cacheable objects will continue to grow as the web moves to providing a richer user experience [14]. There clearly exists a need to provide data authentication and integrity of HTTP data without sacrificing flexibility and scalability as with HTTPS.

1.1 Previous Work

A viable solution to the above problems must have a minimal impact on performance, flexibility, and scalability, which existing security techniques such as HTTPS clearly lack. Towards this end, recent research has attempted to provide lightweight integrity verification and authentication mechanisms for HTTP.

Web Tripwires was proposed to verify if web content has changed between the server response and client rendering by embedding a javascript measurement agent in HTML pages [15]. However, Web Tripwires provides no security protection for the measurement agent itself and can be easily bypassed.

HTTPI provides authentication and data integrity of HTTP (response) data by using a pre-established session key for keyed hashing of client requests and server responses [6]. HTTPI sessions are described as long-lived with no indication of key management. This of course poses significant threats to the security of the system [16]. Furthermore, the evaluation of HTTPI does not consider the cost of TLS/SSL handshakes. Indeed, HTTPI in some cases can show performance as poorly as HTTPS.

Lesniewski-Laas et al. used HTTPS to only provide authentication and integrity while enabling caching [5], which leads to a significant reduction in the origin server loads. However, this technique requires the modification of and the trust in network caches, making it difficult to adopt.

To decouple authentication and integrity from key management, SINE seeks to provide data integrity and origin authentication by digitally signing HTTP (response) data [2], which is signed once and used to serve many client requests to enable network caching. However, SINE is vulnerable to replay attacks of stale authenticated data, and does not support chunked transfer encoding as introduced in HTTP 1.1. HTTPi further enables support of chunked transfer encoding, and subsumes SINE by providing network caching and progressive rendering of both chunked and non-chunked data [7].

SINE and HTTPi are promising in enabling data origin authentication and integrity without key management. However, they suffer from issues that seriously undermine their usability. SINE enables the caching of authenticators for long periods of time by setting a static expiration. This feature allows attackers to launch replay attacks using non-expired data. In other words, clients cannot verify the freshness of data. HTTPi attempts to address the freshness issue by requiring that each response to a client request, or a chunk of a response, be digitally signed, thus severely impacting the server performance. In both SINE and HTTPi, clients are required to perform at least one public key operation per server response. As a result, clients potentially are required to make hundreds of public key operations to render a single HTML page.

1.2 Our Contributions

Among the previous research, SINE and HTTPi are the most promising candidates for lightweight HTTP integrity protection. However, both fail to address the issues outlined previously. In this paper, we propose a new approach called iHTTP to address these problems. In particular, we seek an authentication scheme with performance similar to HTTP. Our approach is inspired by SINE and HTTPi. However, we propose new techniques to achieve stronger security features without sacrificing performance.

We observe that HTTP response data can be categorized into two types: *Client-Static data*, and *Client-Unique data*. Client-Static data refers to site resources which are not specific to a client. For example, many clients requesting a single image via a common URL will receive the same HTTP content. Client-Unique data refers to response data that is directed to a specific client. For example, a client requesting a common URL which returns the client's WAN IP address will return unique HTTP content. In this case each response is unique to the client who requested it. In this paper, we focus on techniques to authenticate and verify Client-Static data.

iHTTP adopts three techniques to achieve lightweight authentication of HTTP response data. The first is to handle encoding data adaptively. HTTP 1.1 data can be encoded in two ways: Content and Transfer [8]. We observe that servers and clients handle encoded data in different manners. For example, compressed data is buffered while non-compressed data can be processed as a stream. We introduce a rule to adaptively apply existing integrity techniques based on encoding format. By processing unique encodings differently, we can reduce the server response size and the number of cryptographic operations without sacrificing flexibility or user experience.

The second technique is the decoupling of freshness verification and signature generation. Previous signature-based HTTP integrity techniques [2, 7] tightly couple data freshness and signature generation. As a result, these techniques either suffer from performance issues or are susceptible to replay attacks. To address this issue, we present a technique called *Sliding-Timestamps*, which decouples signature generation from data freshness authentication by using authenticated hash chain values to calculate an extended timestamp. Servers simply need to release specific hash values to extend the freshness of signatures, which foregoes signing data to update freshness.

The third technique is aimed at reducing the cost of client verification. Signature-based HTTP integrity techniques require clients to verify at least one signature per response. As a result, clients may be required to verify hundreds of responses to render a single web page, which hinders user experience and requires unnecessary computation. Towards this end, we present a technique called *Opportunistic Hash Verification* to provide clients an opportunity to verify responses without signature verification. Using the descriptive nature of HTML, servers can provide contextual authentication information about anticipated future responses. This will reduce client overhead for rendering complex web pages.

We validate iHTTP through a prototype implementation and experimental evaluation. In our experiments, we compare iHTTP with existing standard protocols, HTTP and HTTPS, and the most recent signature-based HTTP integrity technique HTTPi. We show that iHTTP outperforms HTTPS and HTTPi significantly. Furthermore, our results show iHTTP achieves similar throughput to native HTTP for Client-Static data. We also provide an evaluation of the impact of Client-Unique data on signature-based HTTP techniques.

The rest of the paper is organized as follows. Section 2 discusses our design goals and assumptions. Section 3 reviews some signature-based HTTP integrity techniques, on which iHTTP is based. Section 4 presents the iHTTP protocol in detail. Section 5 provides security and performance analysis of iHTTP. Section 6 reports the implementation and evaluation, and Section 7 concludes this paper.

2 Design Goals, Assumptions, and Threat Model

Design Goals: The high-level design goals for iHTTP are given below:

– Data Origin Authentication: Clients should be able to verify whether the received iHTTP data was generated by a trusted identifiable source.
– Data Integrity: Clients should be able to verify whether the received iHTTP response data has been modified by intermediate parties.
– Content Freshness: Client should be able to verify whether the received iHTTP response data is "out of date".
– Low Performance Impact: iHTTP will have minimal impact on servers and clients, allowing for high throughput and low response time.
– Flexibility: iHTTP should allow caching of iHTTP data without modifying network caches or proxies.
– Standards Compatible: iHTTP should be HTTP 1.1 [8] compatible as it is currently the latest HTTP specification and widely adopted on the Internet.

Assumptions: We assume that our client and server machines are trusted and server private keys are protected. We also assume that the clocks on client and server machines are loosely synchronized. We assume data sent over iHTTP is non-confidential as our goal is to provide efficient authentication and integrity protection of HTTP data. Finally, we assume that iHTTP is serving Client-Static data as opposed to Client-Unique data.

Threat Model: We assume that attackers have the ability to intercept, add, modify, delete, reorder, and store all data sent between the client and the server. Attackers can sign data with authentic certificates not associated with the origin server's domain and/or IP address. Finally, attackers can slow down the delivery of data for limited periods of time. The client assumes a reasonable response time from the server.

We consider the following attacks out of the scope of this paper: Attacks aimed at disrupting network availability or undermining cryptographic primitives. Such attacks constitute general attacks on network implementations and protocol weaknesses. We also do not defend against vulnerabilities targeting web applications or scripting software.

3 Preliminaries

In this section we discuss two signature-based HTTP integrity techniques, naive and progressive authentication, presented in [2,7] for the authentication of HTTP content. These techniques form the foundation of the new techniques we develop in this paper. The notation used in this paper is included in Figure 1. Note that authenticator refers to the collective group of information sent to clients to authenticate and verify HTTP responses.

K – Private Key provided by Certificate
Sign_k\{X\} – Sign X with private key K
H(X) – Hash of X
A.t – Timestamp
A.e – Expiration
A.u – Object's URL
A.l – Message length
HTTP – HTTP Response
HTTP.Headers – HTTP Dependent HTTP headers
HTTP.Body – HTTP response body
HTTP.Content – Response Specific Authentication Data
SH – Sign_k\{H(A.t|A.e|A.u|A.l|HTTP.Headers|HTTP.Content)\}
Authenticator – SH, A.t, A.e, A.u, A.l, HTTP.Content

Fig. 1. Notation

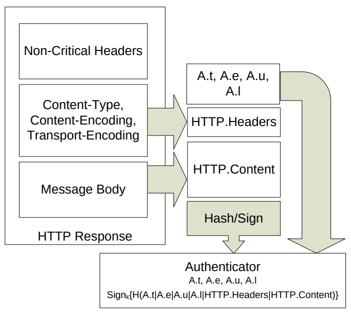
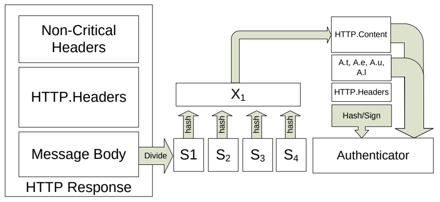


Fig. 2. Naive Authentication

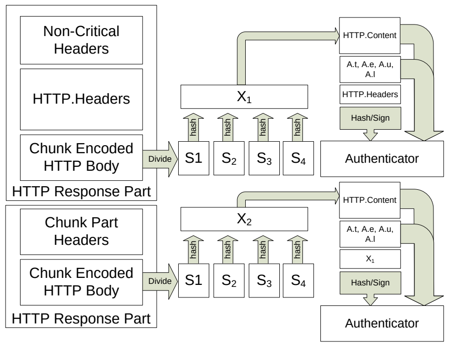
Naive Technique: With the naive technique, a server first buffers the HTTP response prior to sending data. The server then generates the authenticator specific to the HTTP response by signing the hash of a server timestamp (`A.t`), expiration (`A.e`), requested URL (`A.u`), and HTTP response body ($Sign_k\{H(\mathtt{A.t}|\mathtt{A.e}|\mathtt{A.u}|HTTP.Headers|HTTPBody)\}$). The authenticator is then sent as an HTTP header along with the associated timestamp, expiration, and URL. Clients can then verify the integrity of the HTTP response by first buffering the HTTP message, performing one hash operation over the data, and one (public key) signature verification operation. Figure 2 depicts this approach.

The Naive Technique lacks progressive processing and rendering support, which hinders user experience. Thus, Progressive Authentication was proposed to overcome this shortfall [7].

Progressive Authentication of Data Streams: Progressive Authentication enables the authentication of data streams by providing hashes of data



(a) Progressive Authentication



(b) Chunked-Transfer Authentication

Fig. 3. Progressive Authentication Techniques

blocks as part of the authenticator [7]. To enable progressive rendering, the server first divides the entire HTTP response body into equally sized segments $(S_1, ..., S_n)$. Each segment is then hashed to get $H(S_i)$ and the hashes are concatenated into a list $X_1 = H(S_1)|...|H(S_n)$. The signature for the response is generated as $Sign_k(H(\texttt{A.t}|\texttt{A.e}|\texttt{A.u}|\texttt{A.l}|HTTP.Headers|X_1)) = SH_1$ [7]. The authenticator for progressive authentication includes $(X_1, \texttt{A.t}, \texttt{A.e}, \texttt{A.l}, SH_1)$ and is sent along as an HTTP header. Upon receiving the authenticator, the client immediately verifies $X_1, \texttt{A.t}, \texttt{A.e}, \texttt{A.l}$, and $HTTP.Headers$ via the signature SH_1. Once the authenticator is verified, X_1 can be used to immediately authenticate any segment upon arrival [7].

Chunked-Transfer Support: Chunked Transfer-Coding was introduced in HTTP 1.1 to allow for servers to send partial information to clients without knowing the response size [8]. Chunked-Transfer Authentication enables support of chunk encoded data by generating an authenticator for each individual chunk. Essentially each chunk has a unique authenticator that protects the chunk data and chunk order. The authenticator of the first chunk is added as an HTTP header while the authenticators of subsequent chunks are embedded as part of data [7]. Figure 3(b) shows the process for creating the authenticator for chunk encoded data. Please refer to [7] for details.

4 Our Approach - iHTTP

iHTTP is an HTTP integrity approach for efficiently enabling HTTP authentication and integrity, preventing replay attacks, and reducing overhead for both clients and servers. iHTTP achieves these goals by adaptively handling data encoding, enabling Freshness Authentication with Sliding-Timestamps, and providing Opportunistic Hash Verification.

In the following, we first describe a generic authenticator generation process for handling different data-encodings, then expand the authenticator generation process to add Sliding-Timestamps to enable authenticator caching, and finally describe Opportunistic Hash Verification to reduce client verification cost.

4.1 Authenticator Generation

This subsection outlines the iHTTP authenticator content and generation process. Specifically, iHTTP adaptively uses the Naive and Progressive Authentication techniques to enable better performance by reducing cryptographic operations and payload size.

iHTTP uses cryptographic hash and digital signature to generate message authenticators. Several key pieces of information are required for providing authentication, including 1) a timestamp (`A.t`) used to verify the authenticator generation time, 2) an expiration timestamp (`A.e`) for preventing reuse of expired authenticators, 3) the requested URL (`A.u`) to link the response data to the requested URL, 4) a subset of HTTP headers (*HTTP.Headers*), 5) content length (`A.l`), and 6) a message content identifier (*HTTP.Content*) referring to a unique identifier that is generated based on the HTTP message body. After *HTTP.Content* generation, the server hashes and signs the above items as $Sign_k\{H(\texttt{A.t}|\texttt{A.e}|\texttt{A.u}|\texttt{A.l}|HTTP.Headers|HTTP.Content)\}$. For simplicity we refer to this collective group of data as an authenticator.

HTTP.Headers is included in the signature to ensure clients process response data in the correct manner by verifying the response format and properties. This prevents data misuse attacks in which the client is persuaded to handle data in a different manner than specified by the server. HTTP headers are categorized into two groups: End-to-end or hop-by-hop. End-to-end must be stored and forwarded by caches in the original form (with the exception of Content-Length, which may be modified by network caches), while hop-by-hop headers may be modified by caches [8]. Thus, all the non-modifiable headers in the response are included as part of *HTTP.Headers*. To handle the exception Content-Length, iHTTP uses `A.l` to verify the data length in the authenticator.

The Naive and Progressive Authentication techniques differ in the generation process for the message content identifier (*HTTP.Content*). We observe that no single previous technique provides the best performance for generating *HTTP.Content* over all data types, encodings, and formats. Thus we adaptively apply the two techniques to achieve the best performance for generating *HTTP.Content*. Figure 1 contains the relevant notation and information with regards to the authenticator.

iHTTP determines the optimal authentication technique for *HTTP.Content* generation based on the response encoding. We observe that compressed responses require clients to buffer data prior to decompression. As a result, the limitation of Naive Technique pointed out by [2,7] does not apply to compressed data. By applying the Naive technique, iHTTP will reduce the number of hash operations from $O(n)$ to $O(1)$ and decrease the authenticator size by 1.4% [7]. iHTTP determines if responses are compressed by checking for compression tokens located in the Content-Encoding and Transport-Encoding headers. If a compression token is present, the Naive technique is used for the creation of

HTTP.Content, which consists of a SHA-1 hash of the HTTP message body. Otherwise, Progressive Authentication is used for creating *HTTP.Content*, which is the concatenated list of SHA-1 hash segments representing the HTTP message body.

An iHTTP server generates a new authenticator when 1) the content being served for the requested URL has changed, or 2) when the authenticator expires.

Local Authenticator Caching: The techniques presented in this paper rely upon locally caching the server generated authenticators. The Local Authenticator Cache is only concerned with caching the latest generated authenticator per requested URL. Thus, once a new authenticator for a URL is generated, the previous URL specific authenticator can be discarded. Caching may be implemented using many different techniques and data structures. Specific caching implementations may provide better performance in different environments and platforms. Thus, determining the optimal caching mechanism for Local Authenticator Caching is orthogonal to this work.

4.2 Freshness Authentication

As discussed previously, authenticator caching is enabled using a static expiration time. This presents a dilemma to iHTTP. If the expiration time for an authenticator is set too long, an update to the content at the corresponding URL may have occurred before the expiration time. As a result, an attacker may replay the old data using the authenticator before its expiration time. This can certainly be mitigated by using a short expiration time. However, a short expiration time will result in frequent generation of the authenticators, which involve expensive public key operations.

In the following, we present a freshness authentication technique that can provide fresh authentication tokens with light overhead.

Sliding-Timestamp: The problem described above is a result of a tight coupling between authenticator generation and the client URL request. The tight coupling is due to the strong constraint of generating an authenticator to prove freshness. Thus, it is desirable to provide authenticator freshness verification without actually signing the authenticator data.

We propose to meet this goal by allowing the server to extend a given authenticator's timestamp using a server generated hash chain. The server generates the hash chain at the time of authenticator generation and signs the commitment of the hash chain to bind the hash chain and authenticator. Each intermediate hash in the chain represents a calculated amount of time to extend the authenticator's timestamp. The one-way nature of hash chains allows clients to authenticate the server's decision to extend the timestamp without requiring the server to sign the authenticator data again [17]. Figure 4 illustrates this approach.

The hash chain is created using a server generated random number N, the size of the chain n, and one-way cryptographic hash function H as $X_1 = H(N), ..., X_n = H(X_{n-1})$. The random number N is kept secret at the server; this value can be used to calculate intermediate hash values. X_i represents the i^{th} hash in the chain. We consider the number of hash operations to generate X_n

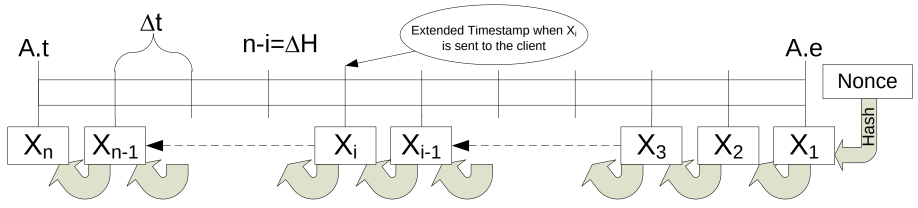


Fig. 4. Sliding-Timestamp Generated using a Hash Chain

beginning with X_i as ΔH and we represent this process as $H^{\Delta H}(X_i) = X_n$. We introduce Δt as a short server-defined configurable time-increment used for extending the authenticator timestamp. Δt represents the time duration associated with each hash operation. Using the above properties, the extended timestamp is calculated by $\texttt{A.t} + \Delta H * \Delta t$, where $\texttt{A.t}$ is the authenticator timestamp.

The server generates a hash chain during each authenticator generation. The size of the hash chain, n, is determined by $\frac{\texttt{A.e}-\texttt{A.t}}{\Delta t}$. The server stores n, Δt, N, and X_n in the local server cache associated with each authenticator. The authenticator signature is modified to include Δt and X_n, i.e., the authenticator signature is generated as $Sign_k\{H(\texttt{A.t}|\texttt{A.e}|\texttt{A.u}|\texttt{A.l}|HTTP.Headers|HTTP.Content|\Delta t|X_n)\}$. Prior to sending an HTTP response, the server must generate the appropriate X_i to authenticate the freshness of the authenticator. X_i is calculated based on the current server timestamp (c), the authenticator's timestamp ($\texttt{A.t}$), Δt, and size of the hash chain (n). The server calculates $i = n - \lceil \frac{c-\texttt{A.t}}{\Delta t} \rceil$ and then generates the i^{th} hash of N as $H^i(N) = X_i$. (Alternatively, the server may pre-compute all hash values and use each appropriately.) The server sends X_i, ΔH, Δt, and X_n as part of the authenticator.

Figure 5 contains the updated final authenticator and signature. The server generates a new authenticator only if $HTTP.Content$ for an HTTP response and locally cached authenticator do not match or if the authenticator expires. Otherwise, the server uses the locally cached authenticator for the response along with the intermediate hash chain value X_i to extend the timestamp.

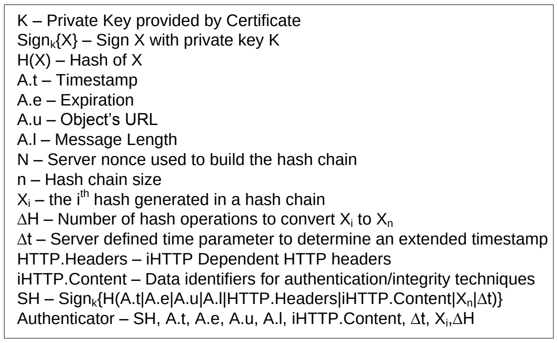

K – Private Key provided by Certificate
$Sign_k\{X\}$ – Sign X with private key K
H(X) – Hash of X
A.t – Timestamp
A.e – Expiration
A.u – Object's URL
A.l – Message Length
N – Server nonce used to build the hash chain
n – Hash chain size
X_i – the i^th hash generated in a hash chain
ΔH – Number of hash operations to convert X_i to X_n
Δt – Server defined time parameter to determine an extended timestamp
HTTP.Headers – iHTTP Dependent HTTP headers
iHTTP.Content – Data identifiers for authentication/integrity techniques
SH – $Sign_k\{H(A.t|A.e|A.u|A.l|HTTP.Headers|iHTTP.Content|X_n|\Delta t)\}$
Authenticator – SH, A.t, A.e, A.u, A.l, iHTTP.Content, Δt, X_i,ΔH

Fig. 5. iHTTP Authenticator

Prior to verifying X_i and the extended timestamp, the client first verifies the authenticator contents via signature. To verify the extended timestamp, the client first verifies X_i is a member of the hash chain rooted at X_j, where X_j is a previously verified hash chain value for the same signature or $X_j = X_n$ when the authenticator signature has not been previously received. X_i is verified if $X_j = H^{\Delta H\prime}(X_i)$ where $\Delta H\prime = \Delta H - \Delta j$ and Δj is the number of hash operations to generate X_n from X_j. If X_i is verified, the extended timestamp is calculated by $\mathtt{A.t} + \Delta H * \Delta t$. The request is then proven fresh if the calculated extended timestamp is greater than the request timestamp. If the authenticator is verified, the client stores that last verified authenticator where $\Delta j = \Delta H$ and $X_j = X_i$ for a unique URL.

Given X_i, the server provides X_{i-1} to extend the authenticator timestamp by Δt. Since it is infeasible for third parties to compute X_{i-1} from X_i given the one-way property of cryptographic hash functions, the clients can be sure only the server can provide X_{i-1}. As a result, this approach decouples the client request and authenticator generation and allows authenticator caching to prevent resigning when *HTTP.Content* matches the cache value for the requested URL.

Network Caching of iHTTP Objects: Servers may direct network caches to store iHTTP data using standard HTTP 1.0/1.1 caching directives. In this manner, no changes are needed by network caches to enable clients and servers to use iHTTP. Thus, iHTTP can be adopted incrementally with gradual changes to the underlying network infrastructure. However, iHTTP does require that the server use cache-directives to allow for effective caching while enabling iHTTP data freshness.

iHTTP uses the "must-revalidate" cache directive to force caches to revalidate every request with the origin server. This ensures the iHTTP server can respond with an updated fresh authenticator for each request. When a NOT-MODIFIED response is provided, the server will provide an updated authenticator and Sliding-Timestamp. The cache can overwrite any changed headers seamlessly and forward necessary data to the client [8].

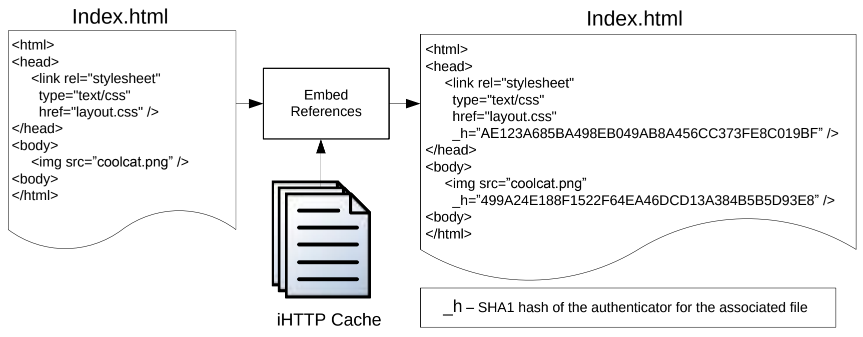
Forcing network caches to query the origin server for each request does result in an unnecessary request when the cached authenticator is fresh. For example, two different clients may request the same file at the same time, resulting in the same authenticator. This limitation is due to the fact that iHTTP does not require cache modification. For caches wishing to handle iHTTP, caches can calculate if the cached authenticator is fresh based on the Sliding-Timestamp and bypass the origin request.

iHTTP also uses the "no-transform" directive to prevent caches from modifying response data and a set of end-to-end headers, which would cause authentication failure.

4.3 Opportunistic Hash Verification

In the above design, iHTTP clients have to authenticate each iHTTP response via at least one expensive signature verification operation. HTML templates often outline multiple HTTP objects required to render a full web page, which results in multiple responses per page.

In HTTP, clients make two types of requests: initial requests and supporting requests. Initial requests are associated with the top level document of any web



```
<html>
<head>
    <link rel="stylesheet"
      type="text/css"
      href="layout.css" />
</head>
<body>
    <img src="coolcat.png" />
<body>
</html>
```

```
<html>
<head>
    <link rel="stylesheet"
      type="text/css"
      href="layout.css"
      _h="AE123A685BA498EB049AB8A456CC373FE8C019BF" />
</head>
<body>
    <img src="coolcat.png"
      _h="499A24E188F1522F64EA46DCD13A384B5B5D93E8" />
<body>
</html>
```

Fig. 6. Opportunistic Hash Verification

page and will always require clients to verify the authenticator signature to ensure freshness. Client supporting requests, on the other hand, are made for supporting resources (e.g., iframes, images, javascript, css) required to render the HTML page.

Note that the number of supporting resources required to render modern websites is fairly large and continues to grow. For example, CNN.com contains 103 unique references to supporting files. Thus, enabling iHTTP for CNN would require that clients perform 104 expensive signature verification operations (including the initial request). This will pose a problem for resource constrained clients. In the following, we propose an Opportunistic Hash Verification technique to reduce the number of client signature verification operations when authenticating an entire web page.

Hash Embedding: The basic idea of Opportunistic Hash Verification is to amortize the expensive signature verification operations at the client by matching authenticated cryptographic hashes with client generated hashes based on the received iHTTP responses. This approach uses the descriptive properties of HTML to allow servers to provide contextual information about potential client requests. Authenticating an HTML document in turn verifies the contextual information and allows clients to bypass most signature verifications.

Specifically, the server will parse all responses containing HTML. HTML tags that may generate additional client requests (i.e., link, img, script, iframe, etc.) are located and each tag's source value is used to search the authenticator cache. The matched authenticator will be cryptographically hashed as $H(\texttt{A.t}|\texttt{A.e}|\texttt{A.u}|\texttt{A.l}|HTTP.Headers|HTTP.Content|\Delta t|X_n)$ and the hashes are embedded in the HTML as attributes of the associated tags. For example, Figure 6 depicts an HTML file which contains two supporting resources: a style sheet and image. The server uses the "src" attributes of these resources to look up the corresponding authenticator in the iHTTP local cache. The associated authenticator is hashed and embedded as part of the HTML. Using the embedded hash gives the client the opportunity to verify the expected authenticator without a signature verification operation.

Upon receiving a response, a client hashes the authenticator content and compares it with the verified embedded hash value (e.g., the "`_h`" attributes on the right side of Figure 6) for the requested URL. If the hash values are equal, the authenticator content is verified. To verify the freshness of the response, the client then uses the Sliding-Timestamp technique presented earlier. If the hash values match and the response is fresh, the *HTTP.Content* can be used immediately to verify the message body. In the case when the hash values of the authenticators do not match, the iHTTP client simply falls back to verifying the authenticator via public key operation as the content of the requested file may have changed in the short time between the server embedding the hash and the sending the HTTP response to the client supporting request.

5 Analysis

This section provides an analysis of iHTTP, including the security properties essential to the guarantees provided by iHTTP, the performance of iHTTP, and its limitations.

5.1 Security Analysis

Data Origin Authentication: An iHTTP server signs authenticators with a protected private key. The corresponding public key is certified by a trusted Certificate Authority and provided to iHTTP clients. Using the verified server certificate, clients can verify the authenticator signatures and thus verify that the data originated from the server that possesses the certificate's private key. Once the authenticator is verified, *HTTP.Content* authenticates the message body by matching one-way hashes of message content with *HTTP.Content*. When the Opportunistic Hash Verification is used, authenticating HTML responses in turn verifies the data origin of the supporting resources (e.g., iframes, images, javascript, css) through the embedded hash values.

Data Integrity: Clients verify the integrity of data through signature verifications and a series of matching of hash values. The response signature allows the client to verify the integrity of the authenticator data and HTTP headers. Once verified, the client can then use the authenticated *HTTP.Content* field to verify the integrity of the message body. When the Opportunistic Hash Verification technique is used, the authenticated HTML allows clients to use the embedded hash values to verify the integrity of the corresponding supporting resources. In other words, the embedded hashes verify the integrity of the authenticator. Part of the authenticator, *HTTP.Content*, can in turn verify the integrity of the message bodies of these supporting resources. Any modification of either the top level web page or a supporting resource will lead to a mismatch and can be detected.

Freshness: Data freshness authentication is provided by the authenticator timestamp (`A.t`) and the calculated Sliding-Timestamp (`A.t`$+\Delta H * \Delta t$). Without the knowledge of the private key of a server, an attacker will not be able to forge an invalid authenticator timestamp without being detected. Moreover, due to the one-way property of the cryptographic hash function H, the attacker cannot forge the hash value X_i used to extend the timestamp, either, unless X_i was released by the server. Assume the maximum clock difference between any client and a server is δ_{max}. The above analysis means that an attacker can hold an authenticated timestamp valid for at most $\Delta t+\delta_{max}$ long before a client identifies it as out of date. Since the clients and the server are loosely synchronized and Δt is chosen by the server, both Δt and δ_{max} can be kept pretty small. Finally, note that the attacker can generate negative influence only when the server modifies the data at the requested URL after the authenticated timestamp is released.

5.2 Performance Analysis

Low Performance Impact: The performance of iHTTP is highly dependent on the authenticator generation process. iHTTP uses Sliding-Timestamps to assist in authenticator caching which amortizes the number of signature generation operations. As a result, iHTTP has both a throughput and response time comparable to HTTP.

However, iHTTP impacts the size of the response due to the addition of authenticators and embedded hashes. Thus, iHTTP requires more bandwidth to send a response which will effect overall throughput. The actual cost of iHTTP is dependent on the content encoding, hash size, and transfer coding. For non-compressed responses, the authenticator size is dependent on HTTP message content, or more precisely, $Hashsize * \frac{ContentSize}{BlockSize}$. Transfer coding applies one authenticator for each chunk of a response. Unfortunately, the number of chunks per response is dependent on the implementation of the server. Thus care should be taken when chunking data as the performance benefit of chunking can be overshadowed by the cost of enabling iHTTP. Finally, enabling Opportunistic Hash Verification adds a hash value for each HTTP object in a given HTML document. Hence, the size of the response is increased by $(NumberofUniqueReferences) * (HashSize)$. This overhead is also specific to the HTTP content.

Flexibility: iHTTP uses HTTP header directives to configure network caches to store authenticators and HTTP responses while enabling data freshness. Furthermore, iHTTP does not require changes to existing network infrastructure or software. Caches wishing to natively handle iHTTP can further improve performance, which will reduce server loads. Furthermore, no changes are needed for server generation software such as PHP or ASP.NET to enable iHTTP.

Standards Compatible: iHTTP uses standard based configurations to achieve compatibility (e.g., the use of standard HTTP headers to configure caches). iHTTP also supports any combination of chunked and compression encodings. In addition, iHTTP only requires minor modifications to clients and servers. iHTTP does not require changes to the network infrastructure or caches, and thus allows incremental deployment.

5.3 Limitation

As discussed previously, HTTP responses can be classified as either Client-Unique or Client-Static. While iHTTP can correctly serve Client-Unique HTTP responses, iHTTP, as well as other signature-based HTTP integrity techniques [2, 7], are not suitable to handle Client-Unique data for two reasons:

First, client-Unique responses can never be cached due to response data being unique per client, which requires data to be signed for each response. As a result, signature-based HTTP integrity techniques will perform at least as poorly as HTTPS. Moreover, existing signature-based HTTP integrity techniques do not provide mechanisms for allowing clients to authenticate response data as logically correct. As a result, attackers or intermediate parties can redirect authenticated fresh data to the wrong clients.

While iHTTP cannot efficiently handle Client-Unique data, Client-Static responses are naturally cacheable by network caches and do not suffer from this vulnerability. Similarly, iHTTP and signature-based HTTP integrity techniques do not authenticate or verify cookies associated with requests or responses. Thus, servers and clients cannot trust received cookies. However, cookies are widely used by servers to provide a unique client experience, and hence prominently used with Client-Unique data. When cookies are provided with Client-Static data, servers can exclude them from the authenticated data to still retain the benefits of iHTTP.

6 Implementation and Experimental Evaluation

6.1 Implementation

iHTTP requires modification of both client and server to enable its security features. On the server side, we implemented iHTTP as an Apache module for handling iHTTP requests and responses. The Apache module is responsible for authenticator generation, managing the authenticator cache, and embedding hash identifiers into HTML. We used Apache's Portable Runtime API to implement caching. To evaluate iHTTP against the latest proposed HTTP integrity technique, we also implemented HTTPi as an Apache module.

iHTTP is designed to handle both chunked and non-chunked data. For non-chunked data, the iHTTP authenticator is added as part of the HTTP headers. Chunked data must be handled differently as chunks occurring after the first chunked do not contain HTTP headers. Since authenticators are associated with each chuck, iHTTP embeds the authenticator as chunked data.

On the client side, we developed a Firefox extension to enable iHTTP support. Our extension relies upon the Mozilla service interfaces for intercepting responses and rewriting data. The extension handles verification as well as support for Opportunistic Hash Verification.

6.2 Experimental Evaluation

Experimental Methodology: First, we provide a microbenchmark to investigate the costs of specific iHTTP operations (e.g., signing, hashing, caching, and hash embedding), which may impact the server performance. These operations also represent the operational costs for the HTTPi implementation. Next we give a macrobenchmark of the iHTTP server module to investigate the throughput and max response time for HTTP, HTTPS, iHTTP, and HTTPi. Finally, we benchmark iHTTP on a resource restricted client to measure the impact on the overall response time for rendering an entire page with Opportunistic Hash Verification enable and disabled.

Experimental Setup: Our hardware platform is an IBM HS22 X-Server with 16 cores and 32 GB of RAM. The iHTTP Apache module is installed with Apache 2.2 web server hosted on a virtual machine (VM) running CentOS 5. Apache is run with the standard configuration for server processes. The VM is configured with dedicated 4 CPU cores and 16 GB of RAM. To run our benchmarks, we created another VM running Ubuntu 11.10 with 2 dedicated CPU cores and 4 GB of RAM, also hosted on the IBM HS22 X-Server. Network communication is provided via ESXi virtual switch.

The iHTTP module is configured with a 2,048 bit SSL Certificate which represents the suggested key strength by NIST [18]. We note that larger keys will have a more severe impact on the performance on existing HTTP integrity protocols and thus iHTTP provides even more benefit as keys become larger.

To test resource constrained clients, we install our iHTTP client module as a Firefox mobile plugin on a Motorolla Droid 2 running Android based Cyanogenmod 7.

Server Microbenchmark: iHTTP has several operations for enabling the protocol that incur overhead. The main operations include hashing, signing, caching, and hash embedding. We instrument the Apache module to record the costs of these operations and display the results in Table 1.

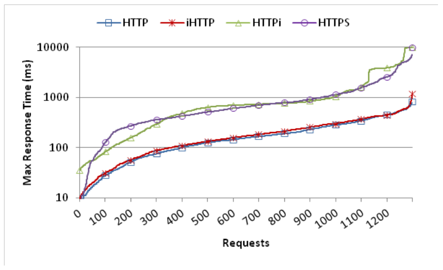
Table 1. Server Microbench Results

Authenticator Creation	4.97771 ms
Signature Generation	4.3207 ms
Hash Embedding	0.13189 ms
Cache Search	0.08751 ms
SHA-1 Operation	0.00042 ms

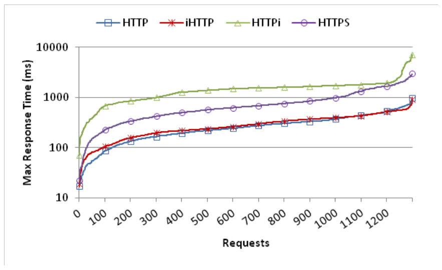
As expected, the time required to sign the authenticator is significantly more costly than the other operations involved with enabling iHTTP. Thus, we can assume a great savings by foregoing signing each chunk with hash based authentication and integrity verification.

Server Macrobenchmarks: To measure the impact of iHTTP on overall performance, we run two different macrobenchmark tests, JMeter Benchmark and SpecWeb2009 Benchmark, to investigate the impact given different website configurations.

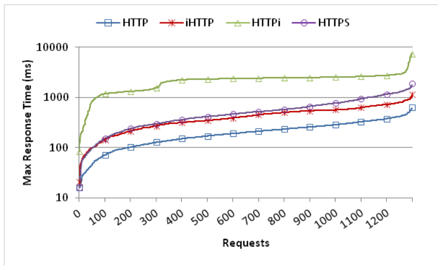
JMeter Benchmark: We deployed a website representing a typical blog or personal website containing only Client-Static data. Four copies were deployed to represent each of the HTTP data formats, non-chunked-not-compressed, compressed, chunked, and chunked-compressed. The sites using chunked data are



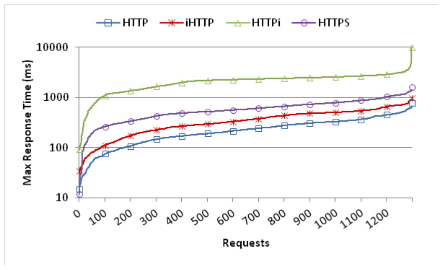
(a) Non-Chunked-Not-Compressed



(b) Compressed



(c) Chunked



(d) Chunked-Compressed

Fig. 7. Distribution of Response Time

configured such that each HTML response contains five chunks. The landing page is 67.91 Kb in size and contains 16 HTTP objects. JMeter benchmark was configured to simulate 130 different simultaneous clients, each making 10 page requests for the site. Each page request resulted in 17 GET requests per page. The interactions were duplicated across the sites to ensure equality and each simulation was run separately. Figure 7 shows the response time with respect to the number of requests.

The figures show that iHTTP outperforms HTTPi in all cases. Furthermore, the response times of iHTTP are very close to HTTP for both Figures 7(a) and 7(b). iHTTP performs not as well for chunked and chunked-compressed data. This is expected since iHTTP must handle each chunk separately, which adds overhead. In general, iHTTP performs better than HTTPS since authenticator caching helps amortize the number of signature operations.

Table 2. JMeter Results for Figure 7(a)

(a) Response Size

HTTP	16087 bytes
iHTTP	17866 bytes
HTTPS	16087 bytes
HTTPi	17226 bytes

(b) Throughput

HTTP	267.2 req/sec
iHTTP	252.8 req/sec
HTTPS	114.6 req/sec
HTTPi	84.1 req/sec

Table 2 shows the response sizes and the throughputs of the HTML documents for Figure 7(a). Here we see the added cost of the authenticators and embedded hashes. The authenticator size is 1,139 bytes, and Opportunistic Hash Verification adds 640 bytes for the HTML page, which are 7% and 4%, respectively, for non-chunked-non-compressed responses. We note that both of these numbers are reliant on the size of the response data.

SpecWeb2009 Benchmark: SpecWeb2009 allows us to investigate the impact of Client-Unique data on servers and protocols by simulating dynamic web applications. We deployed the SpecWeb2009 banking application, which consists of 15 pages and each page makes an average of 13.6 supporting requests with the minimum being 8 requests and maximum being 19 requests. We configured SpecWeb to simulate 150 simultaneous users for HTTP, HTTPS, iHTTP, and HTTPi configurations. Default configurations were used for KeepAlive and SSL sessions on the SpecWeb clients.

Table 3 shows that both iHTTP and HTTPi perform more poorly than HTTPS and HTTP. First, we observe that each of the 15 generated HTML pages generate Client-Unique content per URL request. Hence, the "account_summary.php" page will contain content specific to the user who requested it. In this case, iHTTP and HTTPi will be required to regenerate the authenticator for each client request.

Table 3. SpecWeb2009 Results

Protocol	Avg Resp	Bytes/Req
HTTP	544 ms	41,818
HTTPS	576 ms	41,828
iHTTP	647 ms	50,627
HTTPi	662 ms	52,147

Client Benchmark: This section compares iHTTP Opportunistic Hash Verification with plain signature-based HTTP integrity techniques. We do not provide comparison of iHTTP with HTTP and HTTPS, since previous research has already provided a thorough comparison of signature-based HTTP integrity techniques with HTTP and HTTPS [7].

We installed the iHTTP client on Firefox mobile version 8.0. We requested the landing page of our static website used in the JMeter benchmark and recorded the load time of 20 requests when both enabling and disabling Opportunistic Hash Verification from the server. The plain signature-based approach requires 16 additional public key operations by the client.

The average time per page load for the plain signature-based approach took 7.4321 second. Pages with Opportunistic Hash Verification enabled on average took 5.8291 seconds to load. This shows that Opportunistic Hash Verification reduces the computational overhead of the client by 21% compared to previous HTTP integrity techniques. Furthermore, the disparity of performance will increase with the number of HTTP objects outlined in the HTML page.

7 Conclusion

In this paper, we proposed a new protocol named iHTTP to enable lightweight authentication of Client-Static HTTP response data. The proposed iHTTP protocol adaptively handles different data encodings to allow for better performance without effecting user experience. It also uses a hash chain based Sliding-Timestamps to provide efficient freshness authentication without using public key operations, and exploits Opportunistic Hash Verification to reduce client public key operations. Our experimental evaluation demonstrated that iHTTP provides similar performance to HTTP, and higher throughput and lower maximum response time than HTTPS for Client-Static data.

References

1. Coarfa, C., Druschel, P., Wallach, D.S.: Performance analysis of tls web servers. ACM Trans. Comput. Syst. **24** (February 2006) 39–69
2. Gaspard, C., Goldberg, S., Itani, W., Bertino, E., Nita-Rotaru, C.: Sine: Cache-friendly integrity for the web. In: Secure Network Protocols, 2009. NPSec 2009. 5th IEEE Workshop on. (oct. 2009) 7 –12
3. Vratonjic, N., Freudiger, J., Hubaux, J.P.: Integrity of the web content: the case of online advertising. In: Proceedings of the 2010 international conference on Collaborative methods for security and privacy. CollSec'10, Berkeley, CA, USA, USENIX Association (2010) 2–2
4. Stamm, S., Ramzan, Z., Jakobsson, M.: Drive-by pharming. In: Proceedings of the 9th international conference on Information and communications security. ICICS'07, Berlin, Heidelberg, Springer-Verlag (2007) 495–506
5. Lesniewski-Laas, C.: Ssl splitting and barnraising: Cooperative caching with authenticity guarantees. Master's thesis, Massachusetts Institute of Technology (February 2003)
6. Choi, T., Gouda, M.: Httpi: An http with integrity. In: Computer Communications and Networks (ICCCN), 2011 Proceedings of 20th International Conference on. (31 2011-aug. 4 2011) 1 –6
7. K., S., HJ., W., A., M., C., J., W., L.: Httpi for practical end-to-end web content integrity. Technical report, Microsoft Research (2011)
8. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Rfc 2616, hypertext transfer protocol – http/1.1 (1999)
9. Rescorla, E.: Http over tls. Internet RFC 2818 (May 2000)
10. Rescorla, E., Schiffman, A.: The secure hypertext transfer protocol – shttp (1999)
11. Torvinen, V., Arkko, J., Naeslund, M.: Hypertext transfer protocol (http) digest authentication using authentication and key agreement (aka) version-2. Internet RFC 4169 (November 2005)
12. Goodin, D.: Botnet caught red handed stealing from google. The Register (September 2009)
13. Erman, J., Gerber, A., Hajiaghayi, M.T., Pei, D., Spatscheck, O.: Network-aware forward caching. In: Proceedings of the 18th international conference on World wide web. WWW '09, New York, NY, USA, ACM (2009) 291–300
14. : Cisco visual networking index: Forecast and methodology, 2009-2014 (June 2011)
15. Reis, C., Gribble, S.D., Kohno, T., Weaver, N.C.: Detecting in-flight page changes with web tripwires. In: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation. NSDI'08, Berkeley, CA, USA, USENIX Association (2008) 31–44
16. Menezes, A.J., Vanstone, S.A., Oorschot, P.C.V.: Handbook of Applied Cryptography. 1st edn. CRC Press, Inc., Boca Raton, FL, USA (1996)
17. Perrig, A., Canetti, R., Tygar, J., Song, D.: Efficient authentication and signing of multicast streams over lossy channels. In: Security and Privacy, 2000. S P 2000. Proceedings. 2000 IEEE Symposium on. (2000) 56 –73
18. Barker, E., Roginsky, A.: Transitions: Recommendation for transitioning the use of cryptographic algorithms and key lengths. SP-800-131a, U.S. DoC/National Institute of Standards and Technology (Jan 2011) See `http://csrc.nist.gov/publications/nistpubs/800-131A/sp800-131A.pdf`.