That's the Way the Cookie Crumbles: Evaluating HTTPS Enforcing Mechanisms

Suphannee Sivakorn
Columbia University
New York, NY, USA
suphannee@cs.columbia.edu

Angelos D. Keromytis
Columbia University
New York, NY, USA
angelos@cs.columbia.edu

Jason Polakis
University of Illinois at Chicago
Chicago, IL, USA
polakis@uic.edu

ABSTRACT

Recent incidents have once again brought the topic of encryption to public discourse, while researchers continue to demonstrate attacks that highlight the difficulty of implementing encryption even without the presence of "backdoors". However, apart from the threat of implementation flaws in encryption libraries, another significant threat arises when web services fail to enforce ubiquitous encryption. A recent study explored this phenomenon in popular services, and demonstrated how users are exposed to cookie hijacking attacks with severe privacy implications.

Many security mechanisms purport to eliminate this problem, ranging from server-controlled options such as HSTS to user-controlled options such as HTTPS Everywhere and other browser extensions. In this paper, we create a taxonomy of available mechanisms and evaluate how they perform in practice. We design an automated testing framework for these mechanisms, and evaluate them using a dataset of 30 days of HTTP requests collected from the public wireless network of our university's campus. We find that all mechanisms suffer from implementation flaws or deployment issues and argue that, as long as servers continue to not support ubiquitous encryption across their entire domain (including all subdomains), no mechanism can effectively protect users from cookie hijacking and information leakage.

CCS Concepts

•**Security and Privacy** → **Web protocol security;** *Browser security; Privacy protections;* •**Networks** → *Network privacy and anonymity;*

Keywords

Web Security; Privacy; Eavesdropping; HTTPS; HTTP Strict Transport Security; HTTPS Everywhere

WPES'16, October 24 2016, Vienna, Austria

DOI: http://dx.doi.org/10.1145/2994620.2994638

1. INTRODUCTION

SSL/TLS offer significant protection and are fundamental components for securing our communications. Unfortunately, a recent study [27] found that the majority of top web services do not support ubiquitous encryption and expose their users to HTTP cookie hijacking. Its findings exposed a sad state of affairs, as major services like Google, Bing and Yahoo fail to fully utilize existing security mechanisms and protect their users.

Even though surveys report that 40.5% of popular websites now support HTTPS [29], many websites do not enforce ubiquitous encryption which leads to the significant privacy threats demonstrated in recent work [16, 27, 34]; users are vulnerable to surveillance and information leakage through non-secure cookies, as well as exposed account functionality and potential account takeover.

Migrating to HTTPS is a daunting task with multifaceted, yet diminishing, costs [23], which has resulted in a tangled web of partial support of encryption across websites and flawed access control [26]. Many security mechanisms have been proposed [13, 18, 19] for enforcing encryption in online communications, ranging from server-side mechanisms to client-side solutions. The main server-side mechanism is HTTP Strict Transport Security (HSTS) which was standardized and specified in RFC 6797 [18]. While HSTS is gaining traction, this technology is still in a relatively early state of adoption, with a recent study also showing that many sites deploy the protocol incorrectly [20]. All this has necessitated the emergence of client-side mechanisms, which take the form of browser extensions that allow users to better protect themselves against server-side omissions or errors. HTTPS Everywhere [13], which is the most popular option, was implemented by Tor and the Electronic Frontier Foundation (EFF) and modifies HTTP requests to HTTPS based on a set of community-written rulesets.

In this paper we study existing security mechanisms and defenses, both server- and client-side, explore their *modus operandi*, and evaluate their effectiveness in enforcing HTTPS and preventing cookie hijacking attacks. We study available mechanisms that are already deployed by web services or can be deployed by end users without requiring server modification[1]. While one might expect that enforcing encryption is fairly straightforward, as simple as adding an "`s`" in "`http://`", in practice enforcing encryption is far more complicated, confirmed by the fact that we discovered issues in every mechanism we inspected. Our study focuses on HSTS and HTTPS Everywhere as they are the most

[1]Thus, we omit mechanisms like tcpcrypt [7].

widely adopted server- and client-side mechanisms, respectively, but also explores lesser-used options such as upgrading insecure connections through CSP, and several browser extensions with varying popularity.

Our experimental approach lies in empirically identifying instances where each mechanism fails to instruct the browser to connect over an encrypted channel, and understanding the underlying pathology. At the core of our study lies a dataset of approximately 1.4 billion HTTP requests collected from the public wireless network of Columbia University's campus over the period of one month, which allows us to extensively simulate real-world user browsing behavior. We extract the target URLs, and process them with our testing framework that validates and analyzes the presence of the server-side mechanisms, and also replicates the functionality of the client-side solutions. Subsequently, we inspect all instances of unencrypted connections observed, and conduct an in-depth analysis of unencrypted connections towards domains that have adopted mechanisms for enforcing HTTPS. Among other issues, we have identified deployment issues in the HSTS preload list for the default browser in iOS, even in the latest available version of the browser, which results in significantly reduced coverage. We study client-side solutions and identify their design flaws that result in incorrectly handled unencrypted connections. To our knowledge, this is the first study that offers a comprehensive evaluation of HTTPS enforcing mechanisms by also studying client-side mechanisms. The main contributions of this paper are:

- We provide a taxonomy, and conduct a *comprehensive* experimental evaluation, of existing security mechanisms that force connections over HTTPS or block connections over HTTP, including server-side mechanisms like HSTS and client-side solutions like HTTPS Everywhere.
- Using a real dataset of 1.4 billion HTTP requests from the public WiFi network of our university's campus, we explore the server-side adoption of HTTPS enforcing mechanisms. Surprisingly, we find that Apple's update cycle for mobile browsers results in an outdated and severely limited HSTS preload list, exposing users to eavesdropping and cookie hijacking attacks.
- We present an extensive analysis of HTTPS Everywhere, the most popular client-side mechanism for securing user communications, which is also a default component of the Tor browser. We identify a series of flaws in the rulesets that result in the mis-handling of unencrypted connections and the subsequent exposure of users.

2. SERVER SECURITY MECHANISMS

Figure 1 presents our taxonomy of existing security mechanisms, based on the endpoint that has to deploy the mechanism. As can be seen in the figure, a number of options exist for both the server and end-user, which vary in terms of breadth and effectiveness, as well as intended use. In this section, we explore the mechanisms currently available to servers for enforcing connections over HTTPS or preventing HTTP connections.

2.1 HSTS

The HTTP Strict Transport Security (HSTS) mechanism enables websites to instruct browsers to only establish connections to their servers over HTTPS. It is specified in RFC 6797 published in 2012 [18]. The policy is declared from the web servers via the `Strict-Transport-Security` HTTP header field. To enforce this, the browser maintains a record of the sites that have responded with an HSTS header. Then if the domain the user is connecting to matches a record, the browser will redirect itself (through a `307 Internal Redirect`) or directly modify hyperlinks to HTTPS. This covers any request that would normally be transmitted over HTTP. HSTS is currently supported by approximately 75% of browsers [10]. While this mechanism is gaining significant traction, a recent study [20] reported that only 1.1% of the top 1 million Alexa sites set an HSTS header.

HSTS Header. The HSTS header in the server's response contains the following attributes.

`max-age`: this directive instructs the user's browser for how long to cache the HSTS policy after the receiving the HSTS header, i.e., for how many seconds to maintain an entry for specific domain. This value is updated after each received response from the given domain.

`includeSubdomains`: this *optional* flag indicates whether the HSTS policy will be applied not only to this domain, but also all the subdomains.

`preload`: this *optional* flag specifies whether the site is currently in the HSTS preload list (see below) or under submission to the preload list.

2.1.1 HSTS Preloading

While technically HSTS preload lies on the client-side and is enforced by the browser, the server has to fulfill a set of requirements and apply for inclusion within the list. Thus, we categorize this mechanism as a server-controlled solution.

As HSTS instructs the browser to connect over HTTPS *after* the request has been transmitted, i.e. in the response, the HSTS mechanism does not protect the initial request towards a specific domain. While this is a significantly reduced attack window, nonetheless, users remain vulnerable during the initial connection. To rectify this, major browsers have adopted HSTS preloading. These browsers maintain a list of domains that have a hard-coded HSTS policy, and do not rely on the HSTS header in the response for caching a policy, thus protecting even the initial request.

In addition to upgrading traffic to HTTPS, HSTS preloading is also used to enforce certificate pinning [25], which is designed to prevent attacks that employ rogue certificates [28]. However, this functionality is out of the scope of our study and, thus, we do not explore it in detail.

Chrome preload. The Chromium project is in charge of maintaining the preload list which is shipped with the Chrome browser [11]. Most of the domains contained in the preload list have the `force-https` mode set. However, some domains do not set that mode, indicating that they are assigned to the `Opportunistic` mode in Chrome. For domains with the `Opportunistic` mode set, Chrome will not enforce HTTPS, but will perform certificate pinning. If the user connects over an encrypted channel (by explicitly typing "`https://`" in the address bar, or if the website redirects to HTTPS), Chrome will verify the certificate pinned in the preload list.

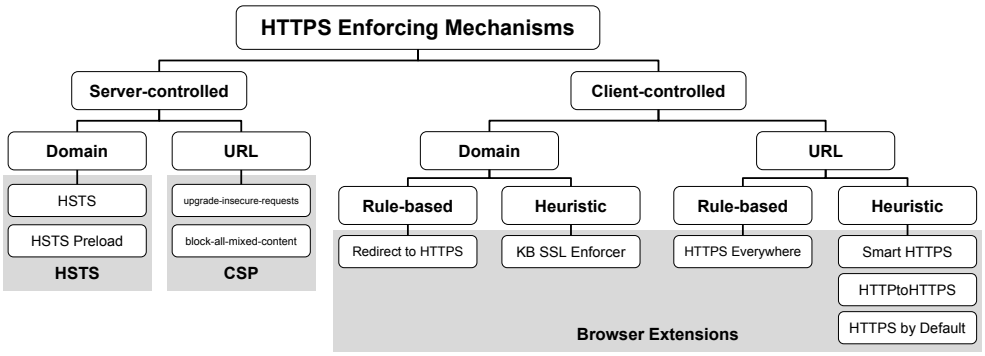


Figure 1: Taxonomy of HTTPS enforcing and HTTP blocking mechanisms.

To be added and remain on the HSTS preload list websites must satisfy a set of requirements set by the Chromium project [2], which includes sending an HSTS header at all times. The project also specifies ways to be removed from the list, which has a slow turn-around time to reach the users, due to the manual nature of this process. Domains are also suggested to continue to serve an HSTS header without the `preload` directive and `max-age` set to 0. As the HSTS preload record does not sync, update or expire automatically during a domain transfer, new owners of domains will have to check and make sure that service is not accessibility is not disrupted due to the lack of support for HTTPS and HSTS. Naturally, as adoption increases, this approach for populating the preload list will encounter significant scalability issues.

Firefox preload. Firefox currently builds a custom list that is derived from the entries in Chrome's list that have the `force-https` mode set, but filters out hosts that do not respond with valid HSTS header or do not meet certain requirements (e.g., set a `max-age` of less than 18 weeks). When a mismatch is found between the HSTS policy in Chrome's list, and the one returned by the server in the HSTS header, Firefox assigns higher priority to the one contained in the server response, and uses that policy in the preload list. Furthermore, if a server responds with `max-age=0`, Firefox considers those sites to be *knockout* entries (e.g., a domain might have a new owner that does not want to support HSTS) and are not included in the preload list [12].

2.2 Content Security Policy

The Content Security Policy (CSP) [30] mechanism allows web servers to deliver a policy to browsers using an HTTP response header. It is widely used for protecting against cross-site-scripting attacks, as it allows the server to declare which dynamic resources are allowed to be loaded through a whitelist. Alternatively, from the HTTP header approach, CSP can be set within the `<meta>` tag in the HTTPS body. However the recommended approach is to enable it via an HTTP response header, as the policy in the tag is not applied to content which proceeded it [31]. Furthermore, CSP works at the page level, not at a domain scale, i.e., the policy in the header will be applied to the specific webpage, and not used to create a policy for the entire domain.

Enforcing HTTPS. The CSP header in an HTTP response can contain the `upgrade-insecure-requests` [33] directive to instruct the browser to "upgrade" all HTTP requests to HTTPS before the fetching request is transmitted.

Table 1: Support of CSP directives in current version of major browsers.

Browser	`upgrade-insecure-requests`	`block-all-mixed-content`
Firefox 47.0	✓	✗
Chrome 52.0	✓	✗
Safari 9.1	✗	✗
Opera 38.0	✓	✗

This can therefore mitigate threats by preventing insecure requests from being transmitted over the network. However, as webpages may reference resources that are hosted on third party servers that do not support encryption, it is not always feasible for a website to instruct the browser to upgrade all connections to HTTPS without breaking the user's browsing experience. This can result in pages with mixed content.

2.2.1 Mixed Content

Mixed content occurs when content is referenced over an insecure HTTP connection within a page that is served over HTTPS [32]. The support for mixed content exposes users to risks, as it allows man-in-the-middle attackers to change website functionality by modifying the HTTP request contents of a webpage's active resources, e.g., script (`<script>`, `XMLHttpRequest`), CSS, fonts, and frames (`<iframe>`). With the inclusion of insecure references to non-active (display) resources such as images, audio, video, even if adversaries are not able to modify critical functionality, the user's HTTP cookies can be exposed to hijacking attacks.

Blocking mixed content. Currently, mixed active content is blocked in major browsers by default [21], while mixed passive content is allowed but accompanied by visual warnings [15]. Our experiments reveal that current versions of Chrome, Firefox, Opera and Safari do not support this CSP mechanism (Table 1). Firefox has recently added initial support for blocking all mixed content through the `block-all-mixed-content` CSP directive in the Firefox Nightly release.

3. CLIENT SECURITY MECHANISMS

In this section we explore the functionality and mode of operation of the security mechanisms at the disposal of users. Specifically, we study 6 browser extensions that attempt to solve the problem of insecure connections over HTTP. Table 2 provides general some information for these extensions, including browser support and number of downloads.

Table 2: Overview of available client-side solutions.

Extension	Browser Support	#*	Last Update
HTTPS Everywhere	Firefox, Chrome, Opera, Firefox for Android, Tor	1.7M	04/2016
Redirect to HTTPS	Opera	123.5K	03/2011
KB SSL Enforcer	Chrome	41.1K	03/2015
Smart HTTPS	Firefox, Chrome, Opera	23.2K	01/2016
HTTPtoHTTPS	Firefox	5.0K	06/2013
HTTPS by Default	Firefox	2.6K	05/2015

*Total downloads across all supported browsers.

3.1 HTTPS Everywhere

HTTPS Everywhere is a browser extension that was developed by the Tor Project and the Electronic Frontier Foundation [13]. The extension operated through rulesets that contain a collection of rules for each domain, that are written as JavaScript regular expressions. Each HTTP request is checked against the rulesets and, if matched, modified to connect over HTTPS. However, since a website's functionality may break under HTTPS, rulesets may contain exceptions for each domain, that instruct the browser to keep the connection over HTTP. As this exposes the user to risk, HTTPS Everywhere has an opt-in option to block all HTTP requests. While this can protect users from HTTP cookie hijacking, it will also break the browsing experience, rendering it an ineffective approach.

3.1.1 HTTPS Everywhere Rulesets

Rulesets are the core of this extension, and consist of per-domain XML files that contain a series of rules that guide the functionality of the extension. An example ruleset can be seen in Listing 1, along with the relevant attributes.

Listing 1: Example ruleset structure.

```
<ruleset name="MySite">
  <target host="mysite.com"/>
  <target host="www.mysite.com"/>
  <target host="*.mysite.com"/>

  <securecookie host="^mail\.mysite\.com$"
    name="^SID$"/>

  <exclusion pattern=
    "^http://excludeme.mysite.com/"/>
  <exclusion pattern=
    "^http://(www\.)?mysite.com/excludeme/+"/>

  <test url="http://www.mysite.com/excludeme"/>
  <test url="http://mysite.com/excludeme/"/>
  <test url="http://www.mysite.com/"/>

  <rule from="^http:" to="https:"/>
</ruleset>
```

Target Host. The target host tag specifies which domain or subdomain should be checked against the rule listed in the particular ruleset. Each ruleset may contain multiple target hosts for a single rule. The target hosts can include the wildcard (`*`) symbol along with a domain name, for covering other subdomains and suffix regional domains.

Rule. The rule contains the appropriate information to guide the extension in rewriting the URL. The `from` and `to` attributes are expressed as JavaScript regular expressions. The extension uses the expression in the `from` attribute to identify links that have to be modified, and rewrites the link according to what is specified in the `to` attribute. The rule tag may also contain the `downgrade` attribute which, when set to `"1"`, results in the link being rewritten from `https` to `http`. This option is useful when a page's functionality breaks over HTTPS, as it allows the remaining pages to be connected to over a secure connection.

Secure Cookies. The secure cookie tag instructs the extension to set the `secure` flag for a specific cookie. The `host` attribute matches the hostname and the `name` attribute is matched against the cookie's name, in order to identify which cookie is to be set to secure.

Exclusion. The exclusion tag is for specifying instances of insecure URLs that should not be rewritten by HTTPS Everywhere. The `pattern` attribute contains the regular expression used for matching URLs.

Test. The test tag is used by rule "authors" for including test URLs can be used to validate the coverage of the rule. Its mandatory for each rule in a ruleset to have $n + 1$ of these implicit test URLs, where n is the number of {`*, +, ?, |`} characters in the rule's regular expression [14]. A test URL can only match against one `rule` or one `exclusion`, and the goal is to cover all the targets of the ruleset, and all the branches of the regular expressions within.

3.1.2 Adding Rulesets

Any voluntary contributor can create and submit new rules to HTTPS Everywhere; new rules can be submitted though their Github directory as a pull request. New rules can also be submitted to the ruleset open mailing list of HTTPS Everywhere.

3.1.3 Ruleset Validation

HTTPS Everywhere has an automated checker that runs basic tests on all rulesets that have been submitted by volunteers. Apart from checking the basic syntax, the checker also verifies all the test URLs specified by the ruleset authors. Any rulesets that fail the checks will be, by default, turned off and inactive in the following released version.

3.1.4 Matching URLs to Rules

Since HTTPS Everywhere does not prohibit overlapping target hosts in different rulesets, one URL can match the target host in multiple rulesets. For each ruleset, the URL will be modified according to the first rule (`<exclusion>` or `<rule>`) that matches it. Therefore an URL can match more than one rules from different rulesets. If there are more than one matching rules, HTTPS Everywhere will modify the URL even if only one of those rules rewrites it. If multiple matching rulesets have URL modification entries, only the first one is enforced and the rest are ignored. If none of the rules that match the URL modify the URL, it will remain the same.

3.1.5 Modifying and Removing Rulesets

Rules can also be modified or removed through pull requests or emails sent to the HTTPS Everywhere ruleset mailing list. Similar to the management of the HSTS preload list, removal is a manual process, which can lead to service accessibility issues between releases if a domain expires or ownership is transferred.

3.2 Alternative browser extensions

There are other browser extensions that attempt to solve the same problem by redirecting requests to HTTPS. While not as popular as HTTPS Everywhere, they still have a considerable number of users. Nonetheless, they follow far more simplistic approaches for enforcing HTTPS, with significant shortcomings that we present here. As two of the mechanisms are severely outdated and don't support recent browser versions, we omit them from our analysis.

3.2.1 KB SSL Enforcer

KB SSL Enforcer [5] is a Chrome browser extension that automatically detects the availability of HTTPS for a domain prior to upgrading to a secure connection. Local lists are maintained with the domains for which to *enforce* HTTPS, and those to *ignore* due to a lack of support. The domains in the enforce list will always be contacted over HTTPS. To detect availability of HTTPS, the extension opens an HTTPS request using `XMLHttpRequest` to contact the specific domain and check the HTTP response status codes whether the request succeeds `(200, 204)`. Depending on the outcome, the domain is added to either the enforce or ignore list. The extension also looks for a HTTPS redirection in the HTTP response headers (`Location`); if found, the domain is added to the enforce list.

However, the extension does not correctly handle sites that redirect (through `<meta http-equiv="refresh" />` or JavaScript) HTTPS connections to HTTP, as they result in an infinite redirection loop. This is due to the server responding with a `200` code to the initial request that is over HTTPS, before redirecting the user to HTTP. Once the extension sees the `200` code, the domain will be added to its *enforce* list. Thus, the next time the user tries to connect to this website, the extension will force it to connect over HTTPS, which will then be automatically redirected by the server to HTTP, which will then be modified again by the extension, resulting in a never-ending loop of redirections. Furthermore, since the extension enforces encryption on both the domain and subdomain level, any path of a domain in the enforce list that does not support HTTPS will not be loaded correctly.

3.2.2 Smart HTTPS

Smart HTTPS [6] maintains a local whitelist and blacklist for domains. Whitelist URLs send HTTP request by default. Blacklist URLs send HTTPS by default. All URLs that are typed by the user will automatically be added to the whitelist. If the user wants to force the URL to load over HTTPS by default, the URL needs to be added manually. Since each URL has to be added manually by the user, simply adding `https://www.example.com` to the blacklist does not enforce other subdomains or subdirectories unless explicitly specified. Also, adding HTTPS to HTTP redirection pages to the blacklist also causes an infinite redirection loop, since the extension will force the connection to be over HTTPS.

3.2.3 HTTPS by default

HTTPS by Default [3] is another extension that follows a simplistic approach. It adds an "`s`" at the end of `http` by default, for any URL typed in the address bar. However, the add-on does not handle other type of requests that are sent from elements in the page, e.g., when retrieving a page that is triggered by the user clicking on an HTTP link. Even though the extension employs HTTPS by default, it does not have a fallback mechanism for websites that do not support HTTPS, resulting in a secure connection error.

4. MEASUREMENT AND ANALYSIS SETUP

In this section, we describe the components of our testing framework and the process of evaluating existing mechanisms that enforce HTTPS. Our testing process can be divided into two main modules: one for *online* tests and one for *offline*.

4.1 Server-side Mechanism Testing

The *online* module focuses on testing mechanisms that lie on the server-side. We use `curl` for probing domains or pages that we want to study. To simulate actual users browsing the pages, we imitate all HTTP request headers sent by Chrome, and allow up to 20 redirections as specified in the Chrome source code (`kMaxRedirects = 20`). We extract the HTTP response headers and body (HTML tag and content) that is relevant to the mechanism we are studying in each experiment.

4.1.1 HSTS Module

The dynamic HSTS header is sent to the browser when it connects to the server. Our module extracts the `Strict-Transport-Security` HTTP header, to obtain the directives given by the specific server.

4.1.2 CSP Module

A server is able to instruct the browser to transparently upgrade insecure requests and/or block all mixed content by setting a Content Security Policy (CSP) in the HTTP response. Like other CSPs, since both upgrade insecure requests and block all mixed content can be set via the HTTP header with the header name and HTML meta tag in the body, our system searches both segments for the `upgrade-insecure-requests` and `block-all-mixed-content` directives.

4.2 Client-side Mechanism Testing

For the *offline* experiments, our goal is to build a testing component that can test any given URL against the client-side security mechanisms that we want to study, without the need to connect to the server. Below we offer details on our modules that test HSTS preload and HTTPS Everywhere.

4.2.1 HSTS Preload Module

This module is designed to check if a URL's hostname is contained in the HSTS preload list. We create a module that takes the URL as an input and tests the presence of the domain's hostname in the HSTS preload list.

System Testing. To make sure that our system simulates the HSTS preload browser behavior correctly, we tested our module against the default preload list on Chromium. Specifically, we verify our validity through Chrome's `net-internals` diagnostic tool for HSTS. Our automated test extracted the entries from the "Query Domain" function of the diagnostic tool, and compared the domains that returned `static_upgrade_mode` against the corresponding entries from our module. Our test set contained 100,000 domains sampled from URLs in our main dataset (detailed in Section 5.1).

Table 3: Public Wi-Fi dataset statistics.

	Requests	%
Total	1,397,563,630	100.00
Contains cookie	509,966,214	36.49

(a) Observed HTTP requests.

	Records	%
URLs	599,034,558	100.00
Domains	699,873	100.00
Domains support SSL/TLS	409,026	58.44

(b) Unique domains and URLs observed over HTTP.

4.2.2 HTTPS Everywhere Module

The straightforward approach of directly executing the actual HTTPS Everywhere extension in an instrumented browser presents a major drawback; it would only allow us to obtain the modified URL and the ruleset that modified it, without any further information on other rulesets that also matched the given URL. It would also incur significant overhead that would prohibit us from experimenting with such a large dataset as the one we use in Section 5. We also implemented and experimented with our own standalone tool that replicates the extension's functionality and leverages the existing rulesets, but abandoned that approach in fear of not capturing the identical behavior to the original tool. To that end, we decided to follow an intermediate approach. We took the extension's code, which is in JavaScript, and slightly modified to output more detailed results (e.g. exclusions matched, no rules matched) and to be able to run with Node.js [24] , giving us the ability to execute the JavaScript without the need for a browser, rendering our experimentation lightweight and efficient.

4.2.3 Other Client Mechanisms

Other options exist, in the form of browser extensions, that allow users to force their browser to issue connections over encrypted channels. Due to space constraints and their simplistic approach to enforcing HTTPS, which results in the ineffectiveness described in Section 3, we omit the other extensions for the remaining of our evaluation.

5. EXPERIMENTAL EVALUATION

Here we describe the findings of our study regarding the coverage, modus operandi, and effectiveness of existing mechanisms that enforce HTTPS.

5.1 Data Collection and Statistics

We setup a logging module on a network tap that received traffic from multiple wireless access points positioned across our university's campus, covering approximately 15% of the outgoing public wireless traffic, over a period of 30 days. For each HTTP request we collected the destination URL and a keyed hash (HMAC) of the cookie's values. We obtained IRB approval prior to running this experiment. Table 3 shows a break down of our dataset, which contains approximately 1.4 billion captured HTTP requests, with over 500 million requests containing at least one HTTP cookie. We found that 58.44% of the unique domains that users accessed over unencrypted connections during our monitoring period are accessible over HTTPS.

Table 4: Base domains and HSTS support.

	Unique Base Domains	%
Connectable over SSL/TLS	409,026	100.00
Support HSTS	9,297	2.27
HSTS + `includeSubdomains`	1,418	0.35
HSTS + `preload`	921	0.23

Table 5: Number of mis-handled HTTP requests, towards (sub)domains covered by HSTS preload.

	HTTP requests	%
Escape HSTS preload	720,170 (382,689 unique URLs)	0.05
Contain cookie	324,061	0.02

5.2 Analysis for HSTS

Using the URLs extracted from our dataset, we study the coverage and effectiveness of HSTS in practice, as detailed in Section 4. We use the preload list released May 2016, which contains a list of 12,602 domains (12,233 base domains). In Table 4, we show how many of the unique base domains from our dataset are connectable over HTTPS. Out of those, only 9,297 contain an HSTS header in the reply, 1,418 of which include the `includeSubdomains` directive in the header. Finally, 921 domains also return headers with `preload` in the header.

Table 5 shows the number of detected HTTP requests toward domains that are covered by the HSTS preload list. While the percentage is relatively small (0.05%), 324,061 of these requests exposed the users to potential hijacking attacks. In Table 6 we breakdown the numbers for unique target domains, and find that out of the 742 domains, 710 apply a `strict` upgrade mode, i.e., have set the `force-https` directive in the preload list. Only 32 of those domains (4.3%) are cases where the HSTS header sets "`max-age`=0", signifying that the server is in the process of requesting to be removed from the HSTS preload list.

Opportunistic Security. There are 259 domains on HSTS preload that are `opportunistic` (Table 7), i.e., do not set `force-https` in the Chrome preload list. The vast majority of those domains belong to Google (250), with 218 of those covering google.com and Google's regional search engines. As demonstrated in previous work [27], this opportunistic approach exposes users to the significant risk of cookie hijacking. These domains and their subdomains do not enforce HTTPS, but 249 perform certificate pinning *when* the connection is over HTTPS. Thus, while these domains use pinning to ensure that the user is connected to the correct server without any MiTM, they do not force the client to always connect over HTTPS. Interestingly, only `learn.doubleclick.net` is opportunistic but not pinned, as it has been excluded due to the use of a different CA.

Partial Security. In the latest release of the preload list that we evaluate, we found that 156 domains do not set `include_subdomains`, with 89 not specifying any directive for the subdomains while 97 explicitly set it to `false`. Sur-

Table 6: HSTS preload escape domain breakdown.

	Domains	Base Domains
Escape HSTS preload	742	332
Static Upgrade Mode		
`strict`	710	326
`max-age=0`	32	5

Table 7: HSTS preload domains set to `Opportunistic`.

Domains	#	Pins
Google and related domains	250	249
Non-Google	9	9
Total	**259**	**258**

Table 8: HSTS preload coverage in different browsers.

Browser	Request remains on HTTP	
	#	%
Chrome	1,382,672,442	98.93
Safari	1,397,419,934	99.99

prisingly, 83 of those 97 sites do this on their base domain name. The risks of partial deployment of HSTS have been discussed in previous work [20, 27]. In the Appendix we include a list of the most popular sites that do not include their subdomains in the HSTS preload list.

Coverage across browsers. Next, we explore the difference in effectiveness due to reduced coverage in other browsers. To obtain the most accurate results, we obtain the *latest* version of both preload lists. The latest version of the list by the Chromium project, released in June 2016, contains 13,139 entries with `force-https` (12,782 base domains). The current version of the preload list (Jun 30, 2016) in Safari contains only 704 entries (462 base domains), covering merely 3.52% of the Chromium preload entries. We quantify the diminished effectiveness, using the HTTP requests from our dataset; we cross-check them with the latest HSTS preload lists from Chromium and Safari and compare the results. As expected, the reduced coverage of Safari has a considerable impact. As seen in Table 8, while Chrome's list prevents almost 15 million requests from being issued over an unencrypted connection, Safari protects two orders of magnitude less requests. The implications of this difference are serious, as it demonstrates that even if iOS users maintain their systems up-to-date, they are still exposed to significant threat due to the minimal coverage offered by the default browser in their devices.

5.3 Analysis for CSP

Below we discuss our findings regarding the use of CSP for upgrading connections to HTTPS, or blocking unencrypted connections. Table 9 breaks down the numbers for the number of landing pages that return the `upgrade-insecure-requests` and `block-all-mixed-content` CSP directives in the HTTP header or HTML meta content tag. Overall, we find very little server-side adoption of these directives as a way to prevent unencrypted connections.

Main Dataset analysis. We select a random subset of 100 million unique URLs that are connectable over HTTPS from our dataset, and study the use of CSP. As shown in Table 9, 27,565 (∼0.03%) of the URLs upgrade the insecure requests, while only 557 blocked all mixed content.

Top site analysis. We also study the use of CSP in the top 1 million sites (according to Alexa), to obtain a more complete picture. We found that only 290 of the top 1 million sites upgrade insecure requests to HTTPS on their landing page, while only 36 block mixed content. The highest ranked domain to upgrade insecure requests is buzzfeed.com (143), while for blocking all mixed content it's github.com (59). However, this use of CSP is far more common in less popular sites, with a median rank of 379,182 and 399,413 respectively. Furthermore, we found that 31 (10.69%) of the domains set the "upgrade-insecure-requests" on HTTP, but not HTTPS. CSP is designed to reduce mixed content on HTTPS pages by modifying content links to be loaded on HTTPS page. As such, setting CSP only on HTTP pages is an incorrect implementation of this mechanism. Similarly, 4 (11.11%) landing pages (domains) set "block-all-mixed-content" only on HTTP.

Table 9: Use of CSP directives for upgrading to HTTPS and blocking mixed content, in 100M URLs from our dataset and the top 1M Alexa sites.

Content Security Policy	Setting	Dataset URLs	Alexa Domains
`upgrade-insecure-requests`	HTTP header	27,565	250
	HTML meta tag	259	40
	Total	**27,824**	**290**
`block-all-mixed-content`	HTTP header	557	36
	HTML meta tag	0	0
	Total	**557**	**36**

Table 10: HTTPS Everywhere ruleset statistics.

	Rulesets	%	Domains	%
Total	**19,807**	**100.00**	**21,839**	**100.00**
Default off	4,374	22.08	4,979	22.80
Platform Dependant				
- Mixed content	1,100	5.55	1,186	5.43
- CA cert	131	0.66	130	0.60
- Firefox	3	0.02	3	0.01
Active on:				
- Firefox	14,607	73.75	16,175	74.06
- Chrome, Opera	14,605	73.74	16,173	74.06
- Tor	15,351	77.50	16,784	76.85

5.4 Analysis for HTTPS Everywhere

In this section we present our findings from the analysis of HTTPS Everywhere, the most popular browser extension for enforcing HTTPS, which was developed by the Tor project and the EFF and has over 1.7 million installations.

5.4.1 Rulesets

We analyzed the HTTPS Everywhere rulesets from the Firefox version 5.1.6 (corresponds to Chrome 2016.4.4) released on April 4, 2016. This version has 19,807 ruleset files containing 48,258 target hosts which cover 21,839 domains[2]. In total, there are 2,024 exclusion rules. Not all rulesets in the release are active, as some rulesets are disabled by default in each release, while others are inactive on specific platforms. We break down the numbers in Table 10.

Default off. Certain rulesets are inactivated by default in each release, either due to mistakes in the ruleset that lead to the ruleset validation tests (see section 3.1.3) failing or it was found that the ruleset causes issues in the browsing experience. These rulesets are indicated by the `default_off` attribute. In total, approximately 22% of the rulesets contained in this release are not activated, demonstrating the difficulty in correctly identifying how HTTPS support changes within a domain and its subdomains, as well as creating the appropriate rulesets.

Mixed Content. In the generally case, any unencrypted content in an encrypted page will be blocked. This is done in most major browsers (e.g. Chrome, Firefox, Opera). How-

[2]We count all regional domains of a website as one.

ever, the Tor Browser (which is a Firefox variant) does not enforce this policy. Rulesets that have the `platform` attribute set as `mixedcontent` will be automatically disabled in Chrome, Firefox and Opera, while they are acceptable in other browsers that allow active and passive mixed contents, such as the Tor Browser. Surprisingly, the reason that the Tor browser allows mix content, is that it also comes with NoScript pre-installed, which provides users with a UI and allows more fine-grained customization[3]. Unfortunately, this may expose less proficient users to risk.

CACert. CACert is a community-driven approach towards the creation of a certificate authority [9]. However, the root certificate is not included in many popular browsers (Firefox, Chrome, Opera, and Tor Browser Bundle). When connecting over HTTPS to one of the sites that use a CACert issued certificate, these browsers return a "signed by unknown party" error message. As such, HTTPS Everywhere does not enable rules that enforce HTTPS in the rulesets of sites that employ CAcert certificates. These cases are indicated by the `platform="cacert"` attribute in the ruleset, and are not activated in browsers that have not added CACert to their root certificate. We found that only 130 domains (5.43%) out of all the domains in the rulesets are disabled because of this.

Firefox. The rulesets that set `platform` as `firefox`, will only be activated in the Firefox browser. In the release we studied, we found only 3 such rulesets.

Overall ∼74% of the domains in the rulesets are currently active on major browsers, while the rest are disabled due to the aforementioned reasons. As the Tor Browser does not disable rulesets because of mixed content, it ends up covering more domains (76%).

Site ranking. In Figure 2a we plot the distribution of the ranking of domains found in the ruleset of HTTPS Everywhere, and the domains in the rulesets that are active on Firefox. We employ the global ranking are returned by Alexa [1], and find that a surprising number of rulesets cover domains with very low ranking. Figure 2b shows the coverage obtained in each tier, with an obvious decrease across tiers, showing that more popular websites have a higher percentage of being covered by ruleset authors. The coverage of top sites is much higher compared to that of HSTS preload, which has been reported previously [20]. Naturally, we cannot calculate coverage for the last two tiers, as the overall number of websites is unknown.

HTTPS Everywhere and HSTS. We test HTTPS support of all the base domains found in the HTTPS Everywhere rulesets. Out of 21,839 domains, we are able to successfully connect to 15,525 (71.09%) domains over an encrypted connection. We show the errors for the remaining domains in Table 11. Out of those domains that support HTTPS, we found that only 2,481 (11.36%) have adopted HSTS.

Experimental evaluation. We tested all the URLs contained in our dataset, to see how many URLs would be protected if every user had installed the HTTPS Everywhere extension. As can be seen Table 12, 26.96% of the requests would be secured by HTTPS Everywhere and transmitted over a secure connection. Out of those requests, 38.54% contained HTTP cookies which would be protected from potential eavesdroppers.

[3]https://trac.torproject.org/projects/tor/ticket/8774

Table 11: HTTPS response when transmitting request over HTTPS to domains in HTTPS Everywhere rulesets.

HTTPS Response	Domains	%
SSL handshake failed error	301	1.38
Certificate error		
- Common name mismatch	1,588	7.27
- Verification failed	1,328	6.08
Others error		
- Timeout	841	3.85
- Could not resolve host	1,073	4.91
- Connection refused/closed/reset	1,183	5.42
Total	**6,314**	**28.91**
OK + No HSTS	13,044	59.73
OK + HSTS	1,857	8.50
OK + Preload HSTS	624	2.86
Total	**15,525**	**71.09**

Table 12: Handling of HTTP requests when HTTPS Everywhere is installed.

URL	Requests	%	Requests w/ Cookie	%
Modified to HTTPS	376,626,901	26.95	145,238,139	10.39
Remains on HTTPS	1,020,936,729	73.05	364,728,075	26.10

Table 13 breaks down the requests that remained over HTTP even though HTTPS Everywhere was installed. 83.18% of the requested URLs do not match any of the ruleset target hosts; Those 849,218,603 requests contain 1,395,371 unique hosts (707,188 base domains). This is either due to the domain itself not supporting HTTPS, or the domain not being covered by a ruleset despite supporting HTTPS. To quantify this, we test if those hosts are connectable over HTTPS, and found that 61.01% of the 1,395,371 unique hosts are indeed connectable. These can be added to the rulesets for increasing coverage. *No rules match* represents the cases where the URL targets a supported hosts, however there is no matching rule to modify to HTTPS, thus remaining over HTTP. We consider this large number of insecure URLs (50.2 million) to be missing from the rulesets due to insufficient coverage of the domain from the ruleset.

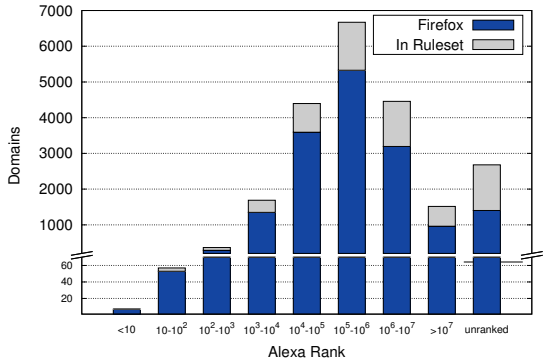
5.4.2 Ruleset Error Classification

Next, we present the different types of errors we have identified within the rulesets that impact the functionality of HTTPS Everywhere.

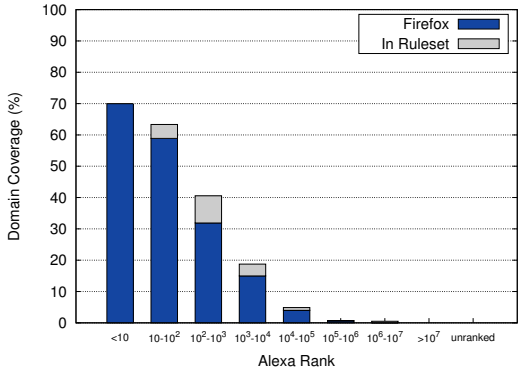
Trailing Slash. By default, Firefox (and the other major browsers) adds a trailing slash at the end of the top level domain Even if the user types the URL without the trailing slash, Firefox will append it. This modification takes place before the URL is processed by HTTPS Everywhere. Listing 2 demonstrates an example ruleset that works correctly regardless of the user adding a trailing slash.

Table 13: Cause for unmodified HTTP requests.

Cause	Requests	%
Exclusion	34,488,882	3.38
Default off	85,114,181	8.34
Mixed content	16,553,194	1.62
CA cert	1,710	0.00
No rule match	50,292,714	4.93
Host in rulesets	171,718,126	16.82
Host not in rulesets	849,218,603	83.18



(a) Number of domains.



(b) Domain coverage.

Figure 2: Number of domains and coverage in each ranking tier, for domains found in all rulesets, and domains in rulesets that are active in Firefox.

Listing 2: Rule expecting trailing slash on top level.

```
<rule from="^http://(www\.)?paypal\.com/"
      to="https://www.paypal.com/"/>
```

However, this is behavior does not extend to all cases of URLs, which can lead to rulesets with errors. Indeed, Firefox does not add a trailing slash for sublevel URLs, e.g., `http://paypal.com/accounts`. To handle such URLs, the ruleset author would have to create a rule that handles both cases, i.e., users adding a trailing slash or not. This results in rulesets with inconsistent handling of the same URL depending on the presence of a trailing slash. For example, in the ruleset in Listing 3, `http://www.google.com/analytics` gets modified to `https://`, while the version with a trailing slash (`analytics/`) does not get modified. The opposite happens for `http://support.apobox.com/system` which does not get modified to `https://`, while the presence of a trailing slash will result in correct handling.

Listing 3: Mishandling due to lack of trailing slash.

```
<rule from="^https?://(?:www\.)?google\
            .(?:com?\.)?\w{2,3}/(?=calendar|
            dictionary|foobar|ideas|partners|
            powermeter|webdesigner)"
      to="https://www.google.com/" />
```

Missing Target Hosts. As can be seen in Listing 4, `http://images.google.com` and the other regional versions are configured to be modified to `https://`. However this ruleset has a single target host (for `google.com`), and as a result, the other regional sites of `http://images.google.*` will not be protected.

Listing 4: Example of missing target host in ruleset.

```
<target host="images.google.com" />
<rule from="^http://images\.google\.((?:com?\.)
           ?\w{2,3})/"
      to="https://images.google.\$1/" />
```

To identify how many rulesets are effected by this type of error, we extracted all the URLs from the test tags and checked if they match the target hosts of the ruleset they belong to. We found that 76 rulesets (from 292 test URLs) failed to match to the host. Next, we created a simple fuzzing tool that extracted the regular expressions from the rules (`<rule from="...." />`), and created a random string that matched the regular expression. While these URLs are obviously invalid to the server, they should nonetheless be caught and modified by HTTPS Everywhere (and our system). This allowed us to detect 440 rulesets, from 492 rules, that were not modified because they failed to match to any target host in the ruleset and, thus, the modification defined by the rule was never enforced. In total, we found 487 rulesets with this type of error. The automated rule validation tool employed by HTTPS Everywhere (see Section 3.1.3) does not capture this error.

Rule Coverage. As shown in Table 13 rulesets miss certain URL patterns, even for domains that are covered. This shortcoming is expected to a degree, as the rules are created manually by the community and many domains have complicated structures and HTTPS support. This occurs even for critical sites, such as Google, where for example services accessed through the `www.google.com/service` do not get modified (e.g., `http://www.google.com/maps`). This also means that HTTPS Everywhere does not handle any error URLs (`http://www.google.com/notavailable`). While HTTPS Everywhere could potentially specify rules that cover non-existing URLs (`google.com/.*`), such an approach is too risky, since other URLs that do not support HTTPS might match the rule and break the user's browsing experience.

6. RELATED WORK

By sniffing unencrypted network traffic adversaries can learn which websites are visiting, as well as obtain sensitive personal user information. A recent study by Hilts and Parsons [17] revealed that sites that contain a large number of trackers often fail to employ HTTPS when transmitting those identifiers [17]. Englehardt et al. [16] argued about how this can be used for mass user surveillance across different services. Even on websites that support HTTPS, client-side state such as that stored in cookies is leaked in unencrypted connections. These leaked cookies can be hijacked, leading to attacks with severe implications [27]. As demonstrated by Zheng et al. [34], adversaries that deploy man-in-the-middle attacks can take advantage of HTTP connections for injecting secure cookies, which can even result in account hijacking. As pointed out Sivakorn et al. [27] and Mundata et al. [22], cookie-based authentication has become increas-

ingly complicated rendering access control flaws a common incident, as website administrators struggle to correctly separate access permissions to different parts of their web services across a large number of inter-connected cookies.

ForceHTTPS [19] was proposed in 2008 by Jackson and Barth in an attempt to offer an effective mechanism for enforcing HTTPS. The work was later revised and resulted in the HSTS standard. Kranch and Bonneau performed an extensive study on the deployment of HSTS (preload) and certificate pinning in practice [20]. While their work offers an extensive analysis of these security mechanisms, our work offers a more comprehensive study on the mechanisms that enforce HTTPS, as we also explore the relevant directives of CSP, as well as client-side mechanisms such as HTTPS Everywhere.

7. DISCUSSION AND FUTURE WORK

Kranch and Bonneau reported that out of the Alexa top million websites that have adopted HSTS, a surprising 59.5% had misconfigurations in their deployment [20]. When taking our analysis into consideration, it becomes apparent that developers struggle when it comes to correctly handling the nuances of existing security mechanisms, rendering hybrid support of both HTTP and HTTPS risky and error-prone (as demonstrated in [27]). As such, these findings highlight the necessity to streamline the deployment of ubiquitous encryption.

Services should also fully deploy HSTS and set the "`secure`" attribute in all cookies. Cookies are not only exposed due to the lack of encryption, but the absence of a strong origin concept [8] coupled with their integrity and scope issues, can lead to cookie injection and shadowing attacks as shown in the work from Zheng and et al. [34].

Our experiments show that the relevant CSP directives are also quite uncommon in practice. However, one should keep in mind that CSP is not designed to enforce HTTPS when loading a page, but instead focuses on the loading of secure content. This mitigation is different from that of the other mechanisms we study, and should be employed for reducing insecure requests within specific usage scenarios.

HTTPS Everywhere is the most effective client-side mechanism that we have found available. Our tests against the network dataset indicates that the number of supported hosts and domains is limited. Even for hosts that have been selected by the ruleset authors, we find URLs that are not covered. We also found that their is significant room for improvement when it comes to the automated evaluation of rulesets prior to their incorporation to the extension.

Future work. Currently, major browsers will attempt to connect over HTTP, when a specific scheme is not specified by the user (unless an HSTS policy exists for the specific domain). A different approach would be to attempt to connect over HTTPS by default, and keep HTTP as a fallback option, which is the approach of some of the mechanisms we explored in Section 3. However, this approach poses several interesting challenges, which we plan on exploring in the future. While cases where HTTPS is not supported can trivially be handled by falling back to HTTP, instances of partial support of HTTPS where certain resources or types of functionality fail silently are challenging to detect (e.g., during our experiments we found that accessing product pages in Amazon over HTTPS broke some functionality and items could not be added to the cart[4]; this has also been previously reported by HTTPS Everywhere users [4]). Such scenarios can be detected by ruleset authors and handled in HTTPS Everywhere; however, the manual nature of this process can lead to errors, as we demonstrated, and is inherently non-scalable.

8. CONCLUSION

The lack of support for ubiquitous encryption in web services poses a serious threat to the security and privacy of our online communications. Apart from personal information being leaked as it travels in cleartext, recent work has shown how cookie-enabled attacks can lead to the exposure of sensitive user information and account functionality. Albeit a known problem, the majority of popular websites have failed to tackle this issue. A number of different mechanisms have been proposed that attempt to enforce encrypted connections and prevent such attacks. Server controlled mechanisms like HSTS (preload) are becoming increasingly deployed, while client-side options like HTTPS Everywhere have attracted the attention of end users and have been integrated in privacy oriented solutions like the Tor browser.

However, our extensive analysis of these mechanisms, which we conducted with our testing framework and a large dataset with real-world traffic, revealed a series of implementation flaws and deployment issues in all the widely available mechanisms. As such, we argue that unless websites strive to offer ubiquitous encryption across their entire domains, and take full advantage of the security mechanisms at hand, existing practices of partial deployment and best-effort approaches will continue to expose users to significant threats.

9. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their feedback. We would also like to thank the CUIT team of Joel Rosenblatt and the CRF team of Bach-Thuoc (Daisy) Nguyen at Columbia University, for their technical support throughout this project. Finally we would like to thank Georgios Kontaxis, for informative discussions and feedback. This work was supported by the NSF under grant CNS-13-18415. Author Suphannee Sivakorn is also partially supported by the Ministry of Science and Technology of the Royal Thai Government. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government or the NSF.

10. REFERENCES

[1] Alexa Top Sites. http://www.alexa.com/.
[2] HSTS Preload. https://hstspreload.appspot.com/.
[3] HTTPS by default. https://addons.mozilla.org/firefox/addon/https-by-default/.
[4] HTTPS-Everywhere Mailing List. https://lists.eff.org/pipermail/https-everywhere/2010-October/000211.html.
[5] KB SSL Enforcer. https://github.com/kbitdk/kbsslenforcer.
[6] Smart HTTPS. https://addons.mozilla.org/firefox/addon/smart-https/.

[4]This was fixed recently, when Amazon significantly increased HTTPS support across their site.

[7] A. Bittau, M. Hamburg, M. Handley, D. Mazières, and D. Boneh. The Case for Ubiquitous Transport-level Encryption. In *Proceedings of the 19th USENIX Conference on Security Symposium*, 2010.

[8] A. Bortz, A. Barth, and A. Czeskis. Origin cookies: Session integrity for web applications. In *Proceedings of 2011 Web 2.0 Security and Privacy*, 2011.

[9] CACert. https://www.cacert.org/.

[10] Can I Use. Strict Transport Security. http://caniuse.com/stricttransportsecurity. (Accessed on 05/2016).

[11] Chromium. HSTS Preloaded list. https://cs.chromium.org/chromium/src/net/http/transport_security_state_static.json.

[12] David Keeler. Preloading HSTS. https://blog.mozilla.org/security/2012/11/01/preloading-hsts/, November 2012.

[13] EFF. HTTPS Everywhere. https://www.eff.org/https-everywhere.

[14] EFF. Ruleset coverage requirements. https://github.com/EFForg/https-everywhere/blob/master/ruleset-testing.md, Jan 2016.

[15] J. el van Bergen. Google Developers – Fixing Mixed Content. https://developers.google.com/web/fundamentals/security/prevent-mixed-content/.

[16] S. Englehardt, D. Reisman, C. Eubank, P. Zimmerman, J. Mayer, A. Narayanan, and E. W. Felten. Cookies That Give You Away: The Surveillance Implications of Web Tracking. In *Proceedings of the 24th International Conference on World Wide Web*, 2015.

[17] A. Hilts and C. Parsons. Half Baked: The Opportunity to Secure Cookie-based Identifiers from Passive Surveillance. In *Proceedings of the 5th USENIX Workshop on Free and Open Communications on the Internet (FOCI 15)*, 2015.

[18] IETF. RFC 6797 - HTTP Strict Transport Security (HSTS) . https://tools.ietf.org/html/rfc6797, November 2012.

[19] C. Jackson and A. Barth. ForceHTTPS: Protecting high-security web sites from network attacks. In *Proceedings of the 17th International World Wide Web Conference*, 2008.

[20] M. Kranch and J. Bonneau. Upgrading HTTPS in Mid-Air: An Empirical Study of Strict Transport Security and Key Pinning. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium*, 2015.

[21] MDN. Mixed content - web security. https://developer.mozilla.org/en-US/docs/Web/Security/Mixed_content, June 2016.

[22] Y. Mundada, N. Feamster, and B. Krishnamurthy. Half-Baked Cookies: Hardening Cookie-Based Authentication for the Modern Web. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, 2016.

[23] D. Naylor, A. Finamore, I. Leontiadis, Y. Grunenberger, M. Mellia, M. Munafò, K. Papagiannaki, and P. Steenkiste. Cost of the "S" in HTTPS. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies, CoNEXT 2014*, 2014.

[24] Node.js Foundation. Node.js. https://nodejs.org/.

[25] C. Palmer, C. Evans, and R. Sleevi. Certificate Pinning Extension for HSTS. RFC 7469, 2015.

[26] K. Singh, A. Moshchuk, H. J. Wang, and W. Lee. On the Incoherencies in Web Browser Access Control Policies. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP)*, 2010.

[27] S. Sivakorn, I. Polakis, and A. D. Keromytis. The Cracked Cookie Jar: HTTP Cookie Hijacking and the Exposure of Private Information. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP)*, 2016.

[28] M. Stevens, A. Sotirov, J. Appelbaum, A. Lenstra, D. Molnar, D. A. Osvik, and B. De Weger. Short Chosen-prefix Collisions for MD5 and the Creation of a Rogue CA Certificate. In *Proceedings of the 29th Annual International Cryptology Conference on Advances in Cryptology*. 2009.

[29] Trustworthy Internet Movement. SSL Pulse – Survey of the SSL Implementation of the Most Popular Web Sites. https://www.trustworthyinternet.org/ssl-pulse/. (Accessed on 05/09/2016).

[30] W3C. Content Security Policy Level 2. https://www.w3.org/TR/CSP2/, July 2015.

[31] W3C. Content Security Policy Level 2. https://www.w3.org/TR/CSP2/#delivery-html-meta-element, July 2015.

[32] W3C. Mixed content. https://www.w3.org/TR/mixed-content, October 2015.

[33] W3C. Upgrade Insecure Requests. https://www.w3.org/TR/upgrade-insecure-requests/, October 2015.

[34] X. Zheng, J. Jiang, J. Liang, H. Duan, S. Chen, T. Wan, and N. Weaver. Cookies Lack Integrity: Real-World Implications. In *Proceedings of the 24th USENIX Conference on Security Symposium*, 2015.

APPENDIX

A. APPENDIX

Table 14: Popular sites that do not cover their subdomains in HSTS preload.

Sites	Alexa rank
facebook.com	3
twitter.com	9
paypal.com	43
wordpress.com	45
airbnb.com	365
usaa.com	478
united.com	695
bitbucket.org	798
mega.co.nz	915
dropboxusercontent.com	1489