Understanding Flaws in the Deployment and Implementation of Web Encryption

Suphannee Sivakorn

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2018

ABSTRACT

Understanding Flaws in the Deployment and Implementation of Web Encryption

Suphannee Sivakorn

In recent years, the web has switched from using the unencrypted HTTP protocol to using *encrypted* communications. Primarily, this resulted in increasing deployment of TLS to mitigate information leakage over the network. This development has led many web service operators to mistakenly think that migrating from HTTP to HTTPS will magically protect them from information leakage without any additional effort on their end to guarantee the desired security properties. In reality, despite the fact that there exists enough infrastructure in place and the protocols have been "tested" (by virtue of being in wide, but not ubiquitous, use for many years), deploying HTTPS is a highly challenging task due to the technical complexity of its underlying protocols (i.e., HTTP, TLS) as well as the complexity of the TLS certificate ecosystem and this of popular client applications such as web browsers. For example, we found that many websites still avoid ubiquitous encryption and force only critical functionality and sensitive data access over encrypted connections while allowing more innocuous functionality to be accessed over HTTP. In practice, this approach is prone to flaws that can expose sensitive information or functionality to third parties. Thus, it is crucial for developers to verify the correctness of their deployments and implementations.

In this dissertation, in an effort to improve users' privacy, we highlight semantic flaws in the implementations of both web servers and clients, caused by the improper deployment of web encryption protocols. First, we conduct an in-depth assessment of major websites and explore what functionality and information is exposed to attackers that have hijacked a user's HTTP cookies. We identify a recurring pattern across websites with partially deployed HTTPS, namely, that service personalization inadvertently results in the exposure of

private information. The separation of functionality across multiple cookies with different scopes and inter-dependencies further complicates matters, as imprecise access control renders restricted account functionality accessible to non-secure cookies. Our cookie hijacking study reveals a number of severe flaws; for example, attackers can obtain the user's saved address and visited websites from e.g., Google, Bing, and Yahoo allow attackers to extract the contact list and send emails from the user's account. To estimate the extent of the threat, we run measurements on a university public wireless network for a period of 30 days and detect over 282K accounts exposing the cookies required for our hijacking attacks.

Next, we explore and study security mechanisms purposed to eliminate this problem by enforcing encryption such as HSTS and HTTPS Everywhere. We evaluate each mechanism in terms of its adoption and effectiveness. 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, no mechanism can effectively protect users from cookie hijacking and information leakage.

Finally, as the security guarantees of TLS (in turn HTTPS), are critically dependent on the correct validation of X.509 server certificates, we study hostname verification, a critical component in the certificate validation process. We develop HVLearn, a novel testing framework to verify the correctness of hostname verification implementations and use HVLearn to analyze a number of popular TLS libraries and applications. To this end, we found 8 unique violations of the RFC specifications. Several of these violations are critical and can render the affected implementations vulnerable to man-in-the-middle attacks.

Table of Contents

List of Figures **vi**

List of Tables **viii**

1 Introduction **1**

1.1 Modern Web Era and Personal Information 1

1.2 Web Session, Information Leakage and Web Encryption 2

1.3 Web Encryption Deployment and Implementation 3

1.4 Thesis Statement 5

1.5 Contributions 5

1.6 What is Not Covered in this Dissertation 7

1.7 Dissertation Roadmap 8

2 Background and Related Work **9**

2.1 Web Session and User Authorization on the Web 9

2.1.1 HTTP Protocol 9

2.1.2 Web Session Management 10

2.1.3 HTTP Cookie 11

2.2 Web Session Hijacking and Unauthorized Access 12

2.2.1 Session Hijacking Attacks 12

2.2.2 Information Leakage over the Network 13

2.3 Web Encryption 15

2.3.1 HTTPS and TLS Protocols 15

2.3.2 TLS Certificate . . . 16
2.4 Caveats in Deployment of Web Encryption . . . 17
2.4.1 HTTP Cookie Scope and Integrity . . . 17
2.4.2 HTTPS Downgrading Attacks . . . 18
2.4.3 TLS Certificate Validation Implementation . . . 19
2.5 Web Encryption Enforcement . . . 19
2.5.1 HTTP Strict Transport Security . . . 20
2.5.2 Certificate Pinning . . . 21
2.5.3 HSTS and HPKP Security Implications . . . 21
2.6 Securing TLS implementations . . . 23

3 Cookie Hijacking and Exposure of Private Information 25
3.1 Overview . . . 25
3.2 Threat Model . . . 26
3.3 Uncovering Current Attack Surfaces . . . 28
3.3.1 Browser Behavior and HTTPS Redirection . . . 29
3.3.2 Mixed Content and HTTP Link . . . 30
3.3.3 Partial HSTS Deployment . . . 31
3.3.4 Persistent Cookie and Logout Invalidation . . . 32
3.4 Information Leakage Study . . . 32
3.5 Analysis of Real-world Services . . . 33
3.5.1 Real-world Privacy Leakages . . . 33
3.5.2 Collateral Cookie Exposure . . . 43
3.6 Network Traffic Study . . . 46
3.6.1 IRB . . . 47
3.6.2 Data Collection . . . 47
3.6.3 Findings . . . 48
3.7 Deanonymization Risk for Tor Users . . . 49
3.7.1 Evaluating Potential Risk . . . 49
3.8 HTTPS Deployment Guideline . . . 52
3.9 De Facto Challenges in Deploying HTTPS Ubiquitously . . . 54

3.9.1 Performance . . . 54
3.9.2 Backward Compatibility . . . 55
3.9.3 Third-party Content . . . 56
3.9.4 Infrastructure . . . 56
3.10 Ethics and Disclosure . . . 57
3.11 Conclusion . . . 57

4 Evaluating HTTPS Enforcing Mechanisms 59
4.1 Overview . . . 59
4.2 HTTPS Enforcing Mechanisms . . . 60
4.3 Server-side Mechanisms . . . 60
4.3.1 HSTS . . . 61
4.3.2 Content Security Policy . . . 62
4.4 Client-side Mechanisms . . . 64
4.4.1 HTTPS Everywhere . . . 64
4.4.2 Alternative Browser Extensions . . . 67
4.5 Measurement Setup . . . 67
4.5.1 Server-side Mechanism Testing . . . 67
4.5.2 Client-side Mechanism Testing . . . 68
4.6 Evaluation . . . 69
4.6.1 Data Collection and Statistics . . . 69
4.6.2 Analysis for HSTS . . . 71
4.6.3 Analysis for CSP . . . 74
4.6.4 Analysis for HTTPS Everywhere . . . 76
4.7 Current Deployment States (Updated Results) . . . 84
4.8 Conclusion . . . 87

5 Hostname Verification in TLS Implementations 89
5.1 Overview . . . 89
5.2 Summary of Hostname Verification in RFCs . . . 91
5.2.1 Hostname Verification Inputs . . . 92

5.2.2 Hostname Verification Rules . . . 93
5.3 Methodology . . . 96
5.3.1 Challenges in Hostname Verification Analysis . . . 96
5.3.2 HVLearn's Approach to Hostname Verification Analysis . . . 97
5.3.3 Automata Learning Algorithms . . . 99
5.4 Architecture of HVLearn . . . 102
5.4.1 System Overview . . . 102
5.4.2 Generating Certificate Templates . . . 103
5.4.3 Performing Membership Queries . . . 103
5.4.4 Automata Learning Parameters and Optimizations . . . 104
5.4.5 Analysis and Comparison of Inferred DFA Models . . . 106
5.4.6 Specification Extraction . . . 107
5.5 Evaluation . . . 108
5.5.1 Hostname Verification Test Subjects . . . 108
5.5.2 Finding RFC Violations with HVLearn . . . 109
5.5.3 Comparing Unique Differences between DFA Models . . . 111
5.5.4 Comparing Code Coverage of HVLearn and Black/Gray-box Fuzzing 112
5.5.5 Automata Learning Performance . . . 114
5.5.6 Specification Extraction . . . 118
5.6 Case Study of Bugs . . . 121
5.6.1 Wildcards within A-labels in IDN identifiers . . . 121
5.6.2 Confusing Order of Checking between CN and SAN Identifiers. . . . 122
5.6.3 Hijacking IP-based Certificates . . . 122
5.6.4 Embedded NULL Bytes in CN/SAN Identifiers . . . 124
5.7 Disclosure and Developer Responses . . . 126
5.8 Contribution . . . 127
5.9 Conclusion . . . 128

6 Conclusion **129**

6.1 Closing Remarks . . . 129
6.2 Future Directions . . . 131

6.2.1 Web Encryption 131
6.2.2 Hostname Checking in Certificate Authority 132
6.2.3 RFC Specification 132

Bibliography **134**

Appendix A Exposure of Privacy Information on Real-world Services **153**
A.1 Additional Real-world Privacy Leakages 153
A.1.1 E-commerce Websites 153
A.1.2 News Media 156
A.1.3 Ad Networks 157
A.2 Alternative Browser Extensions 157

Appendix B TLS Hostname Verification **160**
B.1 Details of Tested Hostname Verification Implementations 160
B.2 Detailed List of Discrepancies 161

List of Figures

3.1 Workflow of an HTTP cookie hijacking attack. 27
3.2 HTTP cookie sent unencrypted with HTTP request before redirect to HTTPS 29
3.3 Private information obtainable from user's Google account through HTTP cookie hijacking. 35
3.4 Extracting contact list and sending email from the victim's account in Yahoo. 37
3.5 Number of exposed accounts per services. 49
3.6 Number of encrypted and unencrypted connections per day, as seen from a freshly-deployed Tor exit node. 50

4.1 Taxonomy of HTTPS enforcing mechanism. 61
4.2 Number of domains and coverages in each ranking tier of HTTPS Everywhere 78
4.3 Histogram of Alexa Top 1 million domains for HSTS, HSTS preload, HTTPS Everywhere and CSP (upgrade-insecure requests or block-all-mixed-content policies). 87

5.1 Fields in an X.509 certificate that are used for hostname verification. 92
5.2 *Exact learning from queries:* the active learning model under which our automata learning algorithms operate. 99
5.3 Overview of learning a hostname verification implementation using HVLearn. 102
5.4 Comparison of code coverage achieved by HVLearn, gray-box fuzzing, and black-box fuzzing for OpenSSL hostname verification. 114

5.5 Number of queries required to learn an automaton with different alphabet sizes (with Wp-method depth=1 and equivalence query optimization). . . . 116
5.6 The number of queries needed to learn the DFA model of CPython certificate verification for different Wp-method depth values (without equivalence query optimization). 117
5.7 TLS implementations' DFA and intersection DFA with CN DNS: `*.a.a` and alphabet: {`a`, `dot`} 120

A.1 Obtaining information about previously purchased items from user's Amazon account. 154
A.2 Side-channel leak of user's browsing history by the Doubleclick ad network. 158

List of Tables

3.1 Browser behavior for user input in address bar. 30
3.2 Overview of the audited websites and services and the type of user information and account functionality they expose 42
3.3 The set of HTTP cookies which are required for hijacking user information. 43
3.4 Cookie exposure by popular browser extensions and apps. 44
3.5 Cookie exposure by official mobile apps. 46
3.6 Statistics of outgoing connections from a subset of our campus' public wireless network for 30 days. 48

4.1 Overview of available client-side solutions. 63
4.2 Unique domains and URLs observed over HTTP (our dataset) 70
4.3 Base domains and HSTS support. 71
4.4 Number of mis-handled HTTP requests, towards (sub)domains covered by HSTS preload. 71
4.5 HSTS preload escape domain breakdown. 72
4.6 HSTS preload domains set to `Opportunistic`. 72
4.7 HSTS preload coverage in different browsers. 73
4.8 Use of CSP directives for upgrading to HTTPS and blocking mixed content, in 100M URLs from our dataset and the top 1M Alexa sites. 75
4.9 Support of CSP directives in current version of major browsers. 75
4.10 HTTPS Everywhere ruleset statistics. 76
4.11 HTTPS response when transmitting request over HTTPS to domains in HTTPS Everywhere rulesets. 79

4.12 Handling of HTTP requests when HTTPS Everywhere is installed. 79
4.13 Cause for unmodified HTTP requests. 80
4.14 Accounts from our public wireless trace that remain exposed even with HTTPS Everywhere installed. 81
4.15 HSTS domains in Alexa Top 1M and the preload list (updated). 85
4.16 Support of CSP directives in current version of major browsers (updated). . 85
4.17 Use of CSP directive for upgrading to HTTPS and blocking mixed content in top 1M Alexa sites (updated). 86

5.1 Hostname verification functions (along with the types of supported identifiers) in TLS libraries and applications 109
5.2 A summary of RFC violations and discrepant behaviors found by HVLearn in the tested TLS libraries and applications 110
5.3 Number of unique differences between automata inferred from different TLS implementations 112
5.4 HVLearn performance for common name `*.aaa.aaa` with Wp-method depth=1 (CPython SSL implementation) 116
5.5 The number of queries needed to learn the DFA model of CPython certificate verification for different Wp-method depth values 119
5.6 Behaviors of TLS implementations for X.509 certificates with IPv4 addresses in CN/subjectAltName 123
5.7 Support for embedded null character in CN/subjectAltName in different TLS libraries 125

B.1 Sample strings accepted by the automata inferred from different hostname verification implementations 162

Acknowledgments

First and foremost, I would like to thank my advisors, Angelos D. Keromytis and Steven M. Bellovin, for giving me one of the greatest opportunities in my life and supporting me throughout my graduate studies. It was a great honor and privilege to be a part of the NSL and supervised by two excellent professors in computer security area.

I am grateful to all of those with whom I have had the pleasure to work during my time at Columbia, especially Jason Polakis, Georgios Kontaxis, Vasileios Kemerlis, and Suman Jana. Their guidance was an essential ingredient for the success of this work.

A special acknowledgment goes to my colleagues at Columbia, particularly, all members of the NSL – Marios Pomonis, George Argyros, Theofilos Petsios, Angeliki Zavou, Dimitris Mitropoulos, and Michalis Polychronakis, as well as Jill Jermyn, Kexin Pei, Vaggelis Atlidakis, and Adrian Tang. Thank you all for making Columbia a fun place to work!

With great pleasure, I would like to thanks those who served on my PhD committee: Angelos D. Keromytis, Steven M. Bellovin, Suman Jana, Roxana Geambasu, and Nicholas Weaver. Also thanks to everyone at NEC Laboratories America Inc, especially Kangkook Jee for hosting my summer internship. Thank you for the opportunity.

I would also like to acknowledge the Ministry of Science and Technology of the Royal Thai Government for the prestigious Master's - PhD scholarship to pursue my education in the United States.

Working in security area often involves breaking stuff, sometimes unintentionally. I was very fortunate to have greatly supportive and understanding people around. Besides my advisors and labmates, I would like to thank Daisy Nguyen, Director of CRF, for being very supportive and super patient! My gratitude extends to other CRF and CUIT members for their technical support throughout my projects as well as Jessica Rosa and Lester Mau for their help in departmental affairs.

I will always cherish the support and encouragement of Pawarisa and Aree, who made my PhD life so much better even from far away.

Finally, nothing would be possible without the support of my family, especially my parents, Somchai and Nalinee Sivakorn. There are no words to describe my love to you both. Thank you for being the best parents in the universe!

Chapter 1

Introduction

1.1 Modern Web Era and Personal Information

In contrast to the first generation of World Wide Web (web), where users were limited to passively viewing web content, modern web services allow users to interact, generate contents, collaborate and share information online. With aforementioned benefits and on-demand self-service property, these online web services offer multiple benefits to users and have become an important part of their daily life. A consequence of this is that a large amount of personal data e.g., user personal and sensitive information, financial information, user location, as well as user online activities and browsing history are uploaded, accessed and recorded through the web. On top of that, as these services are running on the Internet, the global system of computer networks, given how the Internet structured, the data is transferred through multiple untrusted parties and networks.

To make matters worse, over the past few years, user information has become more profitable to attackers [143], government agencies [75], and potentially Internet service providers [83]. These increased exposures of private information and activities to untrusted parties and the sophistication of modern adversaries have become a critical and pressing matter for rapidly understanding the attack scenarios and effective countermeasures. These all have raised critical concerns for ensuring the privacy of digital communications and rendered the need for robust protection mechanisms over the web.

1.2 Web Session, Information Leakage and Web Encryption

In client-server protocols, a *session* is established in order to keep different states of user actions and specific to a user. Similarly, web servers create a *web session* to identify users and keep tracking of user states. Since sensitive information can be accessed over the web, modern web applications require users to verify their identity before accessing their services. Only after successful authentication, a user is authorized to access and interact with their associated account. A *session hijacking* on the web presents high security and privacy threats since sensitive information could be revealed to attackers, something that poses a broad level of threats including information leakage, financial damages, and user de-anonymization.

There are a number of complexities behind web session management. As HTTP (the foundation of web communication) is a stateless protocol [15] – each pair of request and response is independent. In order for web servers to identify and authorize users, the websites require users (browsers) to store states mainly, session ID as an access token to the server. This mechanism also improves user experience, by avoiding requesting re-authentication, as it impacts user engagement. These session IDs, therefore, carry pure bearer tokens [197]. This means they must be secured and unpredictable – allowing only the authenticated user to access them in order to avoid session hijacking attacks. Additionally, any data transferred unencrypted is a goldmine for eavesdroppers and man-in-the-middle attackers [38, 77, 96, 124].

All these threats emphasize the necessity of encryption, especially for modern web technologies, as it provides confidentiality, integrity, and authentication for the Internet users. HTTPS (HTTP Secure) [6] has been the primary web encryption protocol. The HTTPS is an implementation of HTTP over Transport Layer Security (TLS), which are the main cryptographic protocols providing multiple security protocols including encryption and authentication [10]. Over decades, the research community has encouraged websites and modern web browsers to adopt HTTPS (e.g., [71, 97]) as well as help lower the cost deployment (e.g., free certificate [76, 194]). Recent studies from Google showed that 91% of their pages loaded on HTTPS [94]. Similarly, over 69% of all pages on Firefox are now loaded on HTTPS [122].

1.3 Web Encryption Deployment and Implementation

Even though in recent years, the web has massively switched from using the unencrypted HTTP protocol to using "encrypted" communications [80], many web content owners mistakenly think that migrating from HTTP to HTTPS will magically protect them from such attacks – i.e., that there is enough infrastructure in place and the protocols have been "tested" enough (by virtue of being in wide, but not ubiquitous, used for many years) that the switch to encrypted communication is simple. To this end, deploying and implementing web encryption have practical requirements we consider essential:

Ubiquitous HTTPS Deployment and Enforcement. A number of major websites continue to allow requests over unencrypted connections, and in turn leak user information and potentially user's session ID when they are not handled securely. Not enforcing ubiquitous encrypted connections may be attributed to various reasons, ranging from potential increases to infrastructure costs [53, 167], performance [69, 140], and loss of functionality to maintaining support for legacy clients and services [131]. In practice, HTTPS is partially adopted and websites still supporting HTTP load on both domain- and page-level e.g., deploying HTTPS only on sensitive subdomains (e.g., login, payment) [38], no server-side redirection to HTTPS pages [115], allowing static HTTP content to be loaded on HTTPS pages [132, 181]. With no HTTPS enforcement, attackers may force user information to be exchanged over plaintext as by default web browsers connect the user to HTTP [126].

A number of researchers have proposed various HTTPS enforcing mechanisms (e.g., ForceHTTPS [109], HTTPS Everywhere [72]) and browsers have included support that is designed to protect users from such attacks [16, 185]. Though, in practice, most of these approaches are almost never applied or deployed correctly [117]. Furthermore, adopting TLS on HTTP without carefully reviewing existing configurations can still lead to attacks that undermine the security of web users ranging from passive attacks e.g., exposing user information [38, 45, 96], user surveillance [77] to active attacks e.g., HTTP content injection [23], HTTP cookie injection [36, 198].

Securing TLS Implementations. Generally, vulnerabilities in TLS affect HTTPS and other applications that rely on it. The security of HTTPS thus undeniably depends on the security of TLS. Besides weak encryptions and cryptographic attacks, a majority of attacks result from implementations flaws as demonstrated by recently discovered vulnerabilities [37, 47, 56, 111, 119, 177].

One particular reason for this issue is that TLS implementations usually have a huge and complex codebase [135, 190]. They have included a number of functionalities which are necessary for performing cryptographic operations (i.e., encryption, hash functions), plus other related features e.g., certificate creation and validation, digital signature, key generation. An implementation like OpenSSL is far too complex for typical source code analyzers [192]. As a result, even the common vulnerabilities e.g., buffer overread and input validation which are usually detected by general automated analysis testing were not detected [177].

Besides the code complexity problem, the implementation must also comply with RFC specifications, which often involves numerous features and corner cases. To get a sense of the complexity, the certificate validation procedure alone is described in 8 RFCs [37]. On top of that, as the requirements are specific to TLS, typical software testing tools are hardly effective for checking against these type of requirements and identifying any violations. These challenges make maintaining, testing and reviewing the correctness of each implementation significantly harder.

Certificate Validation. In order for the HTTPS connection to be secure, ensuring the identity of the web server is necessary. The TLS (website) X.509 certificate is used to establish the trust between web client and server and presented from the server to the client during the TLS connection establishment. Certificate validation is a process during the TLS handshake process where the client must carefully verify that the server's certificate. If the certificate is not validated correctly, the server authentication guarantee of TLS does not hold, and consequently, web users might be vulnerable to man-in-the-middle attacks. The correctness of certificate validation is necessary to ensure the authentication of the server in web encryption. The certificate validation contains numerous steps e.g., chain of trust

verification, certificate revocation, certificate expiration, and hostname verification and has to comply with the composition of related RFCs.

For example, consider *hostname verification*, a critical component of the certificate validation process that verifies the remote server's identity by checking if the hostname of the server matches any of the names present in the X.509 certificate. This process ensures that the user is connecting to the correct server. The hostname verification is highly complex process due to the presence of numerous features and corner cases such as wildcards, IP addresses, international domain names. It is crucial to conduct thorough a analysis of the implementations for finding any deviation from the specification. However, regardless of how important the certificate validation is, numerous works showed that TLS libraries and applications implement this process incorrectly [37, 47, 79, 85, 144, 150, 153, 171].

1.4 Thesis Statement

Given the complexity and security-critical nature of web encryption, HTTPS and its underlying protocols (i.e., HTTP, TLS), it is crucial for developers to understand the importance of *completeness and correctness of their deployments and implementations*, especially when integrating with existing systems. This has often been overlooked and, as discussed above, hardly achieved through typical software testing tools, which do not take into account the technical complexities, specifications of the involved protocols, and a unified view when composing them.

To this end, the thesis statement of this dissertation is the following:

> *Ensuring the security guarantees of web encryption requires identifying and correcting semantic flaws in the composition of web protocols.*

1.5 Contributions

Aiming to fulfill our thesis statement and the discussed requirements, we propose two main specially designed experiments which focus on: *(i)* identifying authorized session HTTP cookie and sensitive user information leakage over HTTPS services [162] and *(ii)* verifying the correctness and finding specification discrepancies of TLS implementations, specifically

on hostname verification process of TLS certificate validation [160]. Additionally, to obtain a better understanding of the completeness of the HTTPS adoption, we evaluate the effectiveness of currently deployed HTTPS enforcement mechanisms in the wild [163]. To this end, all the testings we develop are able to identify weaknesses in web encryption deployed on the real-world applications, which can be evaluated from e.g., the sensitive information leakage we discovered and measured by the number of RFC violations and discrepancies we found.

In summary, the contribution of this dissertation can be summarized as the following:

- We explore the extent and severity of the unsafe practice followed by major services of partially adopting encrypted connections and its ramifications for user privacy. We demonstrate how HTTP cookie hijacking on HTTPS websites not only enable access to private and sensitive user information but can also circumvent authentication requirements and gain access to protected account functionality.

- We audit an in-depth assessment of 25 major websites, selected from a variety of categories as well as other online ecosystems that include browser extensions, browser search bar, and mobile applications. In each case, we analyze the use of HTTP cookies, the combination of cookies required to expose different types of information and functionality, and search for inconsistencies in how cookies are evaluated.

- We estimate the practicality of our proposed threat by conducting IRB-approved measurements on a subset of our university's public wireless network and detect a large number of accounts exposing the cookies required for our hijacking attacks. In addition, we conduct the measurements on our Tor exit node to demonstrate the practicality and pervasiveness of this threat to Tor users.

- We discovered leakage of sensitive information from our proposed threat on those major services e.g., name, email address, location, browsing history, purchase history. We disclosed our findings to the services we audited and the Tor community, in an effort to assist them in protecting their users from this significant privacy threat.

- Given the insight from our analysis, we provide a summary of HTTPS deployment guideline for web developers and administrators.
- We study and explore available HTTPS enforcing mechanisms e.g., HSTS, HTTPS Everywhere, and evaluate how they perform in practice. 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 information leakage.
- We introduce a novel black-box testing technique for analyzing TLS hostname verification implementations and applications by employing the automata learning algorithms to infer Deterministic Finite Automaton (DFA) that describes the set of hostnames that match the common name or subject alternative name of a given certificate.
- We evaluate our proposed method in terms of code coverage against other existing testing techniques, which is able to achieve significantly higher 11% more on average than existing black/gray-box fuzzing techniques.
- We present the design and implementation of HVLearn as a framework/tool for analyzing hostname verification using the described automata learning, which is publicly available online.
- We assess the effectiveness of HVLearn with 6 popular TLS libraries and 2 applications namely, OpenSSL, GnuTLS, MbedTLS, MatrixSSL, JSSE, CPython SSL, HttpClient, and cURL. Our framework discovered a number of bugs, discrepancies, and unknown RFC violations. We report our findings to developers of each affected library/application.

1.6 What is Not Covered in this Dissertation

While this dissertation intensively explores numerous vulnerabilities, attacks and severe problems on user privacy from *implementation and deployment* flaws of web encryption,

we do not discuss security attacks and vulnerabilities originating from *TLS cryptographic primitives*. These vulnerabilities include weak encryptions, hash functions, pseudo-random generators, as well as protocol-based attacks (this list is not, nor is intended to be, comprehensive). However, as demonstrated by numerous discovered attacks, implementation and deployment bugs on web encryption are often overlooked and severely cause security vulnerabilities even on well-designed web security protocols [37, 47, 56, 111, 119, 192].

1.7 Dissertation Roadmap

Chapter 2 provides important background information for understanding the goals of the thesis and summarize of the current state of related studies. Next, we will show a series of novel studies, techniques, practical tools and frameworks we proposed for identifying flaws in HTTPS deployments and hostname verification of TLS implementations. Specifically, Chapter 3 describes our experiment and discovers the unsafe practice of HTTPS deployments and experimental evaluation of user information leakage over major HTTPS websites. Chapter 4 categorizes and evaluates the effectiveness of current HTTPS enforcing mechanisms. Chapter 5 covers the design and implementation of HVLearn, our hostname verification testing framework, along with a detailed evaluation of the framework on popular TLS libraries and applications. Finally, this dissertation concludes in Chapter 6, where we present future directions addressing challenges in this line of studies.

Chapter 2

Background and Related Work

In this chapter, we provide important background for understanding the goals of the thesis and summarize of the current state of related studies. Section 2.1.3 provides an overview of what is a web session and the necessity of HTTP cookies to web sessions. Next, we explain web session hijacking attacks, HTTP unauthorized requests, and a category of web session attacks which are used to perpetrate a user session. We then present the web encryption components i.e., HTTPS and TLS certificates including TLS hostname verification in Section 2.3. Section 2.4 presents implications on web encryption such as HTTP cookie scope and integrity, HTTPS downgrade attacks, TLS certificate validation as well as related works of this area. We summarize the current HTTPS enforcement mechanisms, particularly HTTP Strict Transport Security (HSTS) and its security implications. Finally, Section 2.6 reviews previous works on securing TLS implementations.

2.1 Web Session and User Authorization on the Web

2.1.1 HTTP Protocol

The core of the web protocol communication between the client (user) and the server (service) relies on the Hypertext Transfer Protocol (HTTP protocol). The protocol was firstly standardized in RFC 1945 [4]. Over time, the protocol has been modified and improved its efficiency in order to ensure its reliability and to support new features. The protocol is stateless and primarily based on HTTP request and HTTP response messages.

HTTP Request is a message sent from a web client to a web server, designed for retrieving web resources (e.g., HTML, CSS, and image) and submitting resources (e.g., form, image) The HTTP request begins with an HTTP method (e.g., `"GET"`, `"HEAD"`, `"POST"`, `"PUT"`) followed by the request uniform resource identifier (URI) which specifies the exact resource the client is requesting. Next, the HTTP header fields allow the client to pass additional information to the server, such as a user agent string, a referrer, the encoding used and an HTTP cookie (Section 2.1.3).

HTTP Response is a response message sent from web server to the client in order to respond to the client's request. The HTTP response begins with status line indicating the protocol version and the status code corresponding to result of the request. Similarly, the HTTP response contains HTTP header fields, which allow the server to pass additional information to the client. An HTTP cookie can also be included in the response header in order to provide this piece of information to the client (which can be later used on for additional HTTP requests). Finally, the requested resource is attached to the body part of the HTTP response.

2.1.2 Web Session Management

HTTP is originally designed for viewing web contents, although, frequently when a user visits a website, a large series of HTTP requests and responses are exchanged between the user and the web server. The web contents are publicly accessible and can be retrieved by anyone. However, when it comes to the modern web era, websites often allow users to also interact and access sensitive contents. This, in turn, requires the users to authenticate themselves to the services and forces the web server to keep track of its interactions with the user, the user state, and the user activity. This tracking has been known in the literature as *session management*.

Since HTTP is stateless protocol, the protocol has no concept of session and state. Each request and response pair is unique and unrelated, therefore the web server needs some mechanism to identify requests that are submitted as part of a web session. To this end, HTTP cookies act as user session IDs.

2.1.3 HTTP Cookie

An HTTP cookie is a piece of data generated from the web server and stored on the client machine. The HTTP cookie is attached to HTTP requests to increase the user's personalization e.g., setting language preference, or other sensitive purposes e.g., identify users and states. After user logged in, services assign privileges of authentication to HTTP cookies, which are sent through HTTP responses and set on user's browser, avoiding requesting re-login unless absolutely necessary, as it impacts user engagement. Numerous works have demonstrated how stolen sensitive HTTP cookies could result in *information leakage* by allowing adversaries to send unauthorized requests to the server and performing attacks such as de-anonymizing user [75, 107], session hijacking [38, 96] and reconstructing the user's history [45, 77]. This stems from the fact that an HTTP cookie acts as a *bearer token* – any party in possession of an HTTP cookie can use it to get access to the associated resources without demonstrating possession of the cryptographic key. *Every web service and web client (browser) is required to protect the user's cookies i.e., making the cookie unguessable to avoid brute force attacks, using encryption to avoid hijacking and inception and prevent cookies accessibility over other parties that the user is interacting with.*

```
Set-Cookie: SID=XXXXX;Domain=.example.com;Path=/;Expires=Wed, 01-Jan-2020
    13:00:01 GMT
Set-Cookie: SSID=XXXXX;Domain=.example.com;Path=/;Expires=Wed, 01-Jan-2020
    13:00:01 GMT;Secure;HttpOnly
```

Listing 2.1: Example of set-cookie in HTTP response header after logging in. The cookie names in this example are `SID` and `SSID`.

The `Set-Cookie` HTTP response header is used to send cookies from web servers to web browsers. Listing 2.1 presents an example of a set-cookie header. Here we summarize important HTTP cookie attributes in `Set-Cookie` header which involve web encryption, HTTPS protocol (Section 2.3). We refer interested readers to RFC 6265 [15] for the full attribute set of an HTTP cookie.

- `cookie-name=cookie-value:` This directive represents a pair of HTTP name and value. The cookie pair is stored in the user's client (browser) and is attached to HTTP

requests matching the specified conditions in set-cookie attributes (e.g., `Domain`, `Path` and `Secure`) below.

- `Domain:` The domain that the cookie is tied to. If not specified, the current host of document location will be used.
- `Path:` the path specifies the URI path which the cookie is tied to. The default path is `"/"`, applied to any URI of the `domain`.
- `Expires:` The expiration time which the cookie will be deleted from the browser. If not specified, the cookie is a *session cookie*, which will be deleted when the browser is closed.
- `Secure:` the secure attributes indicates if the cookie is a secure cookie, meaning the cookie is *only* sent if the HTTP request is made using the HTTPS protocol. The cookie with no `Secure` attribute can be sent with both HTTP and HTTPS requests.
- `HttpOnly:` This directive directs the browsers to not allow the cookie to be accessed by javascript API (`Document.cookie`). Mainly this is designed to avoid cross-site scripting attacks (Section 2.2).

2.2 Web Session Hijacking and Unauthorized Access

Unauthorized HTTP (or HTTPS) requests play a crucial part in the web attacks, as it gives adversaries to send requests to an honest site as if those requests were legitimate and part of the victim's interaction with the honest site, levering the victim's network connectivity and the browser states, such as HTTP cookie, to disrupt the integrity of the victim's session. Obtaining user's HTTP cookies grant access privileges to user information, session hijacking on the web thus targets stealing user's HTTP cookies. Upon successful, many of malicious actions can be launched by attackers.

2.2.1 Session Hijacking Attacks

In summary, we categorize session hijacking attacks and describe each attack as follows.

- *Network Session Hijacking.* Attackers sniff traffic communication channel between victim and web server and steal session cookies [42]. This attack takes advantage of an unencrypted traffic or broken TLS encrypted sessions. Attackers are able to gain access to the victim session, retrieving victim's information and send unauthorized requests to the services on behalf of the victim, as well as inject malicious HTTP cookies to the victim's session.

- *Cross-site Request Forgery (CSRF or XSRF).* Attackers prepare links or websites and distribute to targeted users. When users click the link or open the web pages, they send the prepared malicious requests along with victim's HTTP cookies (with victim's privilege) against the services crafted by attackers [43].

- *Cross-site Scripting (XSS).* Attackers inject malicious scripts in website content, which later visited and executed with user's privileges [44]. Attackers attempt to look for a section on pages that is able to store user's content and display to other users, such as comment section, online posting, and inject JavaScript payloads. If not carefully protected, the actual user's HTTP cookies are obtainable by this type of attack.

CSRF and XSS are the attacks that execute on the client-side, while the attack payloads are sent from the web server. However, if the content sent from the server is unprotected, this also allows man-in-the-middle attackers to inject a malicious payload into the page and, in turn, potentially results in CSRF and XSS. In this dissertation, we focus on preventing a session hijacking attack over the network, as primitive protection for user confidentiality and content integrity. Next, we explore this type of attack, their defenses and summarize their related works.

2.2.2 Information Leakage over the Network

Information leakage over the network has been widely studied in many contexts including the leakage over HTTP. The publicity garnered by the Firesheep extension [38], which demonstrated how easily attackers can hijack a user's session, was a catalyst in expediting migration of critical user activity to mandatory HTTPS connections in major services [175]. Nonetheless, many major websites continue to serve content over unencrypted connections,

which exposes the users' HTTP cookies to attackers monitoring their traffic. Englehardt et al. [77] explored the feasibility of conducting mass surveillance by monitoring unencrypted traffic and inspecting third-party tracking cookies. They also identified cases of PII being exposed in unencrypted traffic, which can be leveraged for improving the clustering of user traffic and linking different requests to the same user. While their work focuses on a different attack scenario, their results also highlight the threat of unencrypted connections. Liu et al. [124] developed a novel method for detecting PII being transmitted in network traffic, without prior knowledge of the fields and form of the information transmitted by each service. Due to the very small fraction of fields that actually contain PII, the authors argue that looking for fields with values that are unique to a user results in very high false positives and false negatives. Thus, mass surveillance attacks will have to employ more advanced techniques. The evaluation of the proposed approach on a large-scale trace presented a false positive rate of 13.6%.

These approaches, however, have a limited viewpoint and can only detect information sent in clear text during the monitoring period. There exist multiple common scenarios where exposed personal information will not be detected: (i) websites are highly dynamic and content may be fetched in an obfuscated form and constructed at runtime on the client side, (ii) sensitive content may always be fetched over encrypted connections, even though HTTP cookies may (erroneously) have sufficient access privileges, (iii) certain pieces of information are only exposed after specific user actions, which may not occur during the monitoring period. Furthermore, cookie hijacking attacks can also access protected account functionalities in certain cases due to imprecise access control.

We explore the prevalence and criticality of private information and account functionality being accessible to HTTP cookies and understanding how varying components of the complicated ecosystem (from browser security mechanisms to mobile apps) affect the attack surface and feasibility of hijacking (Chapter 3). Furthermore, as the authors state [77], using the Tor likely defeats their attack scenario. On the other hand, we demonstrate that while the Tor bundle reduces the attack surface, cookie hijacking remains feasible (Section 3.7).

Castelluccia et al. [45] highlighted the problem of privacy leakage that can occur when personalized functionality is accessible to history page running on HTTP. The authors

demonstrated how adversaries could reconstruct a user's Google search history by exploiting the personalized suggestions. Google has fixed this vulnerability by moving this user's history page to be only accessible to HTTPS. In Section 3.5.1, we will demonstrate how attackers can still reveal partial user's search history.

Deanonymizing Tor Users. The necessity of web encryption also extends to the Tor users. While the Tor network is promoted to enable secure and anonymous communication, the web users on this network still critically depend on the web encryption. Tor network is a group of volunteer-operated networks designed to enhance user privacy protection against network surveillance and traffic analysis. The user traffic is encrypted and sent across different tor relays (nodes). By design, any relay in the network is able to modify and intercept the traffic. While the user traffic is encrypted during transmitting through other nodes, the encryption is terminated at the exit node (final node) and the traffic then moves to the open Internet. This renders possibility for the network attacks in the similar fashion as the open Internet. Due to this design, HTTPS is necessary for web user on Tor. [107] discussed how Tor users could be deanonymized by PII being leaked over HTTP traffic. Winter et al. [193] deployed their tool HoneyConnector for a period of 4 months and identified 27 Tor exit nodes that monitored outgoing traffic and used stolen decoy credentials. We also study the information leakage attack on the Tor exit node in Section 3.7.

2.3 Web Encryption

2.3.1 HTTPS and TLS Protocols

HTTPS and TLS are the core of web encryption. Here we provide background and related works on TLS and HTTPS.

TLS Protocol. The TLS is family of cryptographic protocols operating on application level designed to provide an end-to-end security encryption. A number of popular protocols e.g., HTTPS, SMTPS, SMTPS are implemented over TLS. Over time, TLS has been improved and deprecated in their older versions [19, 188] in order to ensure the guarantees of the protocols and keep up with discovered attacks and vulnerabilities.

HTTPS Protocol. HTTPS (Hypertext Transfer Protocol Secure) is an implementation of TLS (Secure Socket Layer or Transport Layer Security) protocol over HTTP and is the current standard practice of encryption on the web. The HTTPS protocol is mainly defined in RFC 2818 [6]. The protocol has been integrated into Netscape browser since 1994 [189]. The main purpose of this protocol is to protect users against eavesdropping and man-in-the-middle attacks.

TLS Attacks. Clark et al. [52] conducted a survey of security issues in TLS protocols and categorized their issues mainly in term of cryptographic and trust model. A number of these attacks also affect HTTPS. Since our study does not focus on TLS attacks especially from cryptographic flaws, we refer interested readers to [21, 30, 33, 35, 66, 67, 102, 121, 139, 173]. Amrutkar et al. [24, 25] studied TLS security indicators in mobile browsers.

2.3.2 TLS Certificate

While web encryption maintains the confidentiality and integrity of the HTTP exchanged between web servers and users, the user still needs to verify the identity of the web server to ensure that their information is transfer to the right party. The mechanism for website authentication in HTTPS is the user's validation of the server's X.509 public key certificate [11] presented during the TLS handshake. In contrast to regular TLS certificate validation, in HTTPS only the website side is authenticated.

The TLS certificate contains information about the website e.g., common name (website's hostname), public key, list of certificate issuers (certificate authority (CA)), expiration date. The certificate relies on public key infrastructure (PKI) [14], where a trusted CA verifies the identity of the site and signs the website's certificate. This signed certificate allows third party (e.g., web user) to validate that the public key in presented certificate belongs to the individual and ensure that the message is only readable to the server when encrypted using the public key.

Even though X.509 certificate mainly defined in RFC 5280 [11], according to Brubaker et al. [37], the validation process is "extremely complex" and described in addition of 7 RFCs. Dietz et al. [62] and Parsovs et al. [150] investigated the other direction, client certificate

authentication, which can also be applied to the implementations of server-side validation of client certificates. [61, 68, 70, 182] conducted large-scale Surveys of TLS certificates "in the wild". We explore the security issues of TLS certificate in Section 2.4.3. Vratonjic et al. [182] studied the top million sites' certificate and found that in most cases domain mismatch are the main reason causing the certificate validation failed.

TLS Hostname Verification. Hostname verification is a critical component of the certificate validation process that verifies the remote server's identity by checking if the hostname of the server matches any of the names present in the X.509 certificate. Hostname verification is a highly complex process due to the presence of numerous features and corner cases such as wildcards, IP addresses, international domain names, and so forth. For example, Kaminsky et al. [111] presented a vulnerability in several hostname verification implementations that mishandle embedded NULL characters in X.509 certificate and can be used to trick a CA into issuing a wrong subject name. However, so far prior works did not cover analyzing hostname verification in detail primarily due to the hardness of accurately modeling the implementations. We study this verification process in Chapter 5.

2.4 Caveats in Deployment of Web Encryption

As mentioned already, deploying web encryption involves various protocols and in turns, leaves numerous implications on each protocol as well as implications when they integrating, such as TLS cryptographic flaws, TLS certificate validation, HTTPS downgrading attacks. Numerous works have studied each of these implications and purposed defenses. Here we explore and summarize the works only related our study including the cookie handling problems, TLS certificate, and HTTPS enforcing.

2.4.1 HTTP Cookie Scope and Integrity

Cookie Scope and Related Domain. The related domains are domains that share a domain suffix, such as `foo.site.com` and `bar.site.com`. The work from Bortz et al. [36] pointed out that HTTP cookies do not have a strong integrity protection against related domains. For example, a request to `foo.site.com` also attaches cookies domain

`.foo.site.com` and `.site.com`. In the same way, any response to `bar.site.com` can set HTTP cookie to domain `.bar.site.com` and `.site.com`. [36, 127, 198] demonstrated attacks that make a use of no strong integrity of the domain scope in HTTP request and response, where the attacker has a control over related subdomains (e.g., `attacker.site.com`). Then the attacker trick a user to visit their control page and *inject* their malicious cookie on the domain suffix cookie (e.g., `.site.com`). Later the user visits sensitive related domains (e.g., `payment.site.com`), where the `.site.com` cookie is also attached to the user's request. These attacks are referred as *cookie tossing* or *cookie shadowing*. To avoid this problem, the public suffix domain list is introduced and adopted on major browsers in order to prevent cookies of the specified suffix domain being used in upper related domains [136].

Cookie Injection. As mentioned, HTTP cookie can be injected into the cookie from a related domain utilizing no integrity of domain suffix scope. Additionally, attackers can completely overwrite *secure* cookies over HTTP responses with their malicious cookies that have the same name, domain, and path. Zheng et al. [198] categorized cookie injection related works as well as studied cookie injection in details in terms of how the injected cookie can be controlled by attackers based on how they injected (e.g., time), and problems in CDN shared domains. To this end, similar to the purpose of this thesis, they also insisted on the principle that *websites and browsers should never issue unencrypted request regardless of how sensitive of the requested domain or URL is.*

2.4.2 HTTPS Downgrading Attacks

Although HTTPS has been integrated on browsers since 1994 [189], a large number of websites that support HTTPS still do not attempt to redirect users to HTTPS by default [172]. Additionally, even with the HTTPS redirection, attackers still can intercept before HTTPS begins and downgrade the victim's connection to HTTP. A well-known example of this type of attack is Marlinspike's SSLStrip [126]. The attack works seamlessly in practice, as the victim's browser does not show any insecure warning (connections have already been switched to HTTP).

2.4.3 TLS Certificate Validation Implementation

There is a large body of work on various attacks on the TLS certificate validation, mainly due to the incorrect implementation in TLS libraries and applications. Georgiev et al. [85] demonstrated that TLS certificate validation is completely broken in many libraries and applications. The vulnerabilities they identified such as accepting self-signed certificates, no hostname verification, extend to several critical applications e.g., banking, payment system, e-commerce. Brubaker et al. [37] studied and identified issues in implementations of TLS certificate validation on client side. They found various serious issues, for example, servers with X.509 version 1 certificate can issue fake certificates for any domains, accepting expired certificates and self-signed certificates. Fahl et al. [79] studied HTTPS content on Android apps and found that approximately 8% Android apps did not validate hostname. Nezha [153] and SymCerts [47] presented additional TLS certificate validation implementation bugs which conflict with the RFC specifications and enable man-in-the-middle attacks. Kaminsky et al. [111] demonstrated that several hostname verification implementations mishandled embedded NULL characters in X.509 certificates and can be used to trick a CA into issuing a valid leaf certificate with the wrong subject name. However, they found this issue manually and did not have any automated techniques for analyzing hostname verification implementations.

In addition to all issues and error warnings in TLS applications, users still click through browser warning and neglect the security warning [118, 170, 174]. We explore verifying the correctness of TLS implementations including the TLS certificate validation in Section 2.6.

2.5 Web Encryption Enforcement

Many security mechanisms have been proposed [16, 72, 109] for enforcing encryption in online communications, ranging from server-side mechanisms to client-side solutions. First, Jackson and Barth presented ForceHTTPS [109], a browser extension for enforcing HTTPS connections. This was reformed and standardized as HSTS (RFC 6797) [16]. Here we summarize the HSTS, their important security detail and setting.

2.5.1 HTTP Strict Transport Security

The HTTP Strict Transport Security (HSTS) mechanism enables websites to instruct browsers to only establish connections to their servers over HTTPS. HSTS is originated from Jackson et al., ForceHTTPS [109], a browser extension that designed to enforce HTTPS connections. It is later specified in RFC 6797 published in 2012 [16]. 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 85% of browsers including major browsers (e.g., Chrome, Firefox, Safari, Internet Explorer, and Opera) [41]. While this mechanism is gaining significant traction, a recent study [117] reported that only 1.1% of the top 1 million Alexa sites set an HSTS header.

```
Strict-Transport-Security: max-age=<expire_time>
Strict-Transport-Security: max-age=<expire_time>; includeSubDomains
Strict-Transport-Security: max-age=<expire_time>; preload
```

Listing 2.2: Example of HSTS response header

HSTS Header. The HSTS header in the server's response contains the following attributes. Listing 2.2 presents HSTS syntax in HTTP response header.

- `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 the 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.

2.5.1.1 HSTS Preload

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 preload. These browsers maintain a list of domains which have 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.

2.5.2 Certificate Pinning

Adversaries may create or obtain fraudulent certificates [52, 173] that allow them to impersonate websites (victim's domain) as part of man-in-the-middle attacks. Additionally, studies [141, 166], shows how governments may compel CA to issue a rogue certificate on any domain in order to intercept the targeted domain's traffic of the victim (*compelled certificate creation attack*). To prevent these attacks, websites can specify a (limited) set of hashes for certificates in the website's X.509 public key certificate chain on their HSTS preload record. HSTS preload list is also used to enforce certificate pinning (RFC 7469) [18] or HTTP Public Key Pinning (HPKP), which is purposed as an extension to HSTS. Browsers that support certificate pinning are allowed to establish a secure connection to the domain only if at least one of the predefined pinned keys matches one in the certificate chain presented.

Dynamic HPKP. The certificate pinning can also be set dynamically using HTTP header. We refer the reader to the RFC 7469 [18] for a more detailed description of dynamic HPKP header.

2.5.3 HSTS and HPKP Security Implications

Kranch and Bonneau [117] performed an extensive study on the adoption of HSTS and certificate pinning in the wild. Based on their findings, they reported a lack of understanding

by web developers on the proper use of these mechanisms, as they often use them in illogical or invalid ways. Selvi [158] demonstrated scenarios where an attacker could bypass HSTS protection. Bhargavan et al. [34] also showed how the HSTS header could be partially truncated, resulting in the expiration of the HSTS entry within seconds and HTTP connections being allowed. We study HSTS and other HTTPS enforcing mechanisms and evaluate their effectiveness in practice in Chapter 4.

HSTS Supercookie. In addition to the issue in the incorrect deployment of HSTS, the HSTS header itself also allows a website to track user even without logging in or deploy a tracking cookie. Since a domain owner can control their subdomains, a domain owner can set a specific subdomain as a tracking ID (e.g., `ID123.myusertracker.com`) that different for each user and include HSTS for that subdomain. This cause user's browsers to cache the particular domain. This information can be then tested and retrieved by different domains, for example, attempting to request `ID123.myusertracker.com` domain in HTTP, and check if the domain is switched to HTTPS by HSTS (visited). This scenario is referred as "HSTS supercookie" and even described in the RFC of HSTS [16]. This attack is also applicable to the HPKP scheme.

HPKP Lockout. There has been a number of controversies on the HPKP scheme, particularly, users could be unable to access the site due to the HPKP for various implications. For example, the pinned key is hardly guaranteed to work due to the instability in browsers and CA operations. In addition, Zadegan and Lester [196] presents abusing the HPKP scheme, where the attacker maliciously overwrites HPKP pinned key on a website being compromised. Although, later even after the websites is restored to the legitimate owner, the attackers are still able to deny any access to the websites from users. For example, the attacker can overwrite pinned certificate key value of HPKP to be invalid. Any user visiting the compromised site is not able to access as long as the duration in max-age directive due to the incorrect pinned key previously set by the attacker.

Due to these controversies, the Chrome browser developer team decided to deprecate the support of HPKP after recently deployed only two years [149]. A new scheme, "Expect-CT

header" that is designed to provide more flexibility in recovering from configuration errors (avoiding HPKP Lockout), is currently in the working draft [1].

2.6 Securing TLS implementations

The security analysis of different components of TLS implementations has been examined in a large number of projects.

Automated Analysis of TLS Implementations. Brubaker et al. [37] and subsequently Chen et al. [48] used mutation-based differential testing to find certificate validation issues. However, in their case, the hostname verification functionality of the libraries under test is disabled in order to discover other certificate validation issues and thus, they cannot uncover bugs discovered by our work. He et al. [101] used static analysis to detect incorrect usage of TLS libraries APIs. Somorovsky [169] created TLS-Attacker a tool to fuzz the TLS implementations systematically. However, TLS-Attacker focused on finding bugs in the protocol level and did not analyze the hostname verification functionalities of TLS implementations. Finally, de Ruiter and Poll [60] used automata learning algorithms to infer models of the TLS protocol and manually inspected the machines to find bugs. Contrary to our approach, where we focus on analyzing hostname verification implementations, their work focused on the TLS state machine induced by the different messages exchanged during the TLS handshake.

Certificate Validation. With the automated analysis mentioned above, Brubaker et al. [37] tested of TLS certificate validation implementations (except hostname verification) by performing differential testing of 14 TLS implementation with 8 million Frankencerts generated from the random combination of syntactically correct pieces of valid seed certificates. Frankencerts discovered 18 unique certificate validation discrepancies of 15 categories. Georgiev et al. [85] studied different ways that TLS API was abused in non-browser software. They manually identified pervasive incorrect certificate validation in different TLS implementations on which critical software rely. Petsios et al. [153] presented Nezha, a domain-independent *differential testing*, where they used this differential testing frame-

work to explore the behavioral asymmetries between TLS libraries based on their domain-independent input generation mechanisms. To this end, they discovered a higher number of differences than existing works [37, 85]. Chau [47] purposed an *adapted* symbolic execution approach to identify RFC violation in certificate validation to avoid path explosion. Using the domain-specific optimization (certificate chain validation code), their technique extracts path constraints for each accepted and rejected certificate, later cross-validate these output from different TLS libraries to find implementation flaws. They uncovered new 48 RFC noncompliance. Fahl et al. [79] investigated the incorrect usage of TLS API in Android apps.

Chapter 3

Cookie Hijacking and Exposure of Private Information

3.1 Overview

It is our belief that *the completeness of HTTPS deployment is necessary for web encryption to be effective*. In support of this claim, in this chapter, we explore the extent and severity of the unsafe practice followed by major services of partially adopting encrypted connections and its ramifications for user privacy. We demonstrate how HTTP *cookie hijacking attacks* on partially deployed HTTPS do not only enable attackers to access to private and sensitive user information but can also circumvent authentication requirements and gain access to protected account functionality.

To fully understand and evaluate the practicality and extent of the attack in the real world applications, in Section 3.5.1, we audit 25 major services, selected from a variety of categories that include search engines and e-commerce sites. In each case, we analyze the use of HTTP cookies, the combination of cookies required to expose different types of information and functionality, and search for inconsistencies in how cookies are evaluated. We identify a recurring pattern and uncover flaws in major websites that allow attackers to obtain a plethora of sensitive user information and also to access protected account functionality. We reveal a number of severe flaws; attackers can obtain the user's home and work address and visited websites from Google, Bing, and Baidu expose the user's complete

search history and Yahoo allows attackers to extract the contact list and send emails from the user's account. Furthermore, e-commerce vendors such as Amazon and Ebay expose the user's purchase history (partial and full respectively), and almost every website exposes the user's name and email address. Ad networks like Doubleclick can also reveal pages the user has visited.

To estimate the extent of the threat, in Section 3.6, we run IRB-approved measurements on a subset of our university's public wireless network for 30 days and detect over 282K accounts exposing the cookies required for our hijacking attacks. The privacy implications of these attacks become even more alarming when considering how they can be used to de-anonymize Tor users. Our measurements suggest that a significant portion of Tor users may currently be vulnerable to cookie hijacking.

These studies were conducted during June - November 2015. While the current states of some services might have changed over time, our analysis and uncovering attack surfaces are still applicable to the nature of deploying HTTPS in order to avoid session hijacking. Throughout our case studies on popular services, we demonstrated the significance of correct HTTPS deployment to the user privacy.

3.2 Threat Model

Depending on the attacker's ability and resources, a user's HTTP cookies can be hijacked through several techniques. To demonstrate the severity of the privacy leakage due to unencrypted connections, we assume the role of a weak adversary and conduct experiments through passive eavesdropping. Nonetheless, we also investigate cookie characteristics that could be exploited by active adversaries for increasing the scale of the attacks.

HTTP Cookie Hijacking. The adversary monitors the traffic of a public wireless network, e.g., that of a university campus or coffee shop. Figure 3.1 presents the workflow of a cookie hijacking attack. The user connects to the wireless network to browse the Web. The browser appends the user's HTTP cookies to the requests sent in cleartext over the unencrypted connection (❶). The traffic is being monitored by the eavesdropper who extracts the user's HTTP cookies from the network trace (❷) and connects to the vulnerable

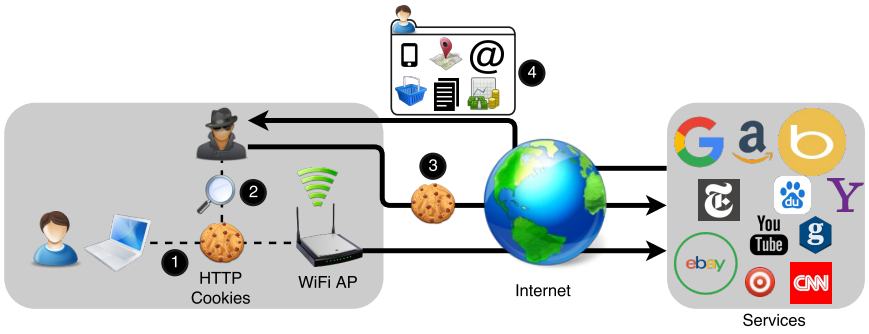


Figure 3.1: Workflow of an HTTP cookie hijacking attack. After the victim's cookies are exposed on the unencrypted connection (1) and stolen (2), the attacker can append the stolen cookies when browsing the target websites (3) and gain access to the victim's personal information and account functionality (4).

services using the stolen cookies (3). The services "identify" the user from the cookies and offer a personalized version of the website, thus, exposing the user's personal information and account functionality to the adversary (4).

Cookie Availability. These attacks require the user to have previously logged into the service, for the required cookies to be available. Having closed the browser since the previous log in does not affect the attacks, as these cookies persist across browsing sessions. Attackers are able to use these type cookies as long as they are unexpired and services have not invalidated them.

Active Adversary. Attackers can follow more active approaches, which increase the scale of the attack or remove the requirement of physical proximity to the victims, i.e., being within the range of the same WiFi access point. This also enables more invasive attacks. For example, the attacker can inject content to force the user's browser to send requests to specific vulnerable websites and expose the user's cookies, even if the user does not explicitly visit those sites. This could be achieved by compromising the wireless access point or scanning for and compromising vulnerable routers [102]. Furthermore, if the HTTP cookies targeted by the attacker do not have the `HttpOnly` flag set [59], they can be obtained

through other means, e.g., XSS attacks [44]. Users of major services can also be exposed to such attacks from affiliated ad networks [191].

State-level Adversary. In the past few years, there have been many revelations regarding mass user surveillance by intelligence agencies (e.g., the NSA [151]). Such entities could potentially deploy HTTP cookie hijacking attacks for obtaining access to users' personal information. Reports have disclosed that GCHQ and NSA have been collecting user cookies at a large scale as part of user-tracking programs [84, 168]. As we demonstrate in Section 3.6, these collected cookies could be used to amass a large amount of sensitive information that is exposed by major websites. Furthermore, in Section 3.7 we discuss how Tor users, who are known to be targeted by intelligence agencies [32], can be de-anonymized through the hijacked HTTP cookies of major services.

3.3 Uncovering Current Attack Surfaces

As mentioned in Chapter 2.2, it is a known vulnerability that cookies can be obtained by HTTP request and modified by HTTP response. This means the current protection against the attack requires that the victim's browser *never* sends any unencrypted HTTP request to a target site or any of its related domain.

To understand the current vulnerabilities and exposures for cookie hijacking attack, we set up an experimental wireless network, where we place two hosts connecting to the network. While one host behaves as a user with typical internet browsing activities (e.g., login, search, browse pages), the other host behaves as a passive attacker collecting the network traffic. We focus on the top Alexa HTTPS-adopted websites from various categories (the details of our finding of each service including types of leakage information, list of cookies which are required in the attack in Section 3.5.1). We found that although websites have deployed their services and pages majorly in HTTPS, number of requests are still currently issued via the unencrypted connection. We describe the recurring attack surfaces we found across websites as following.

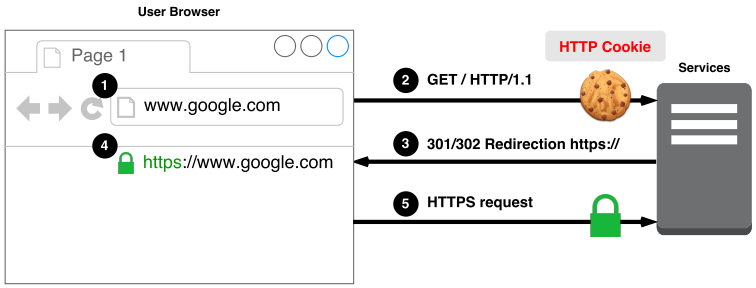


Figure 3.2: HTTP cookie sent unencrypted with HTTP request before redirect to HTTPS

3.3.1 Browser Behavior and HTTPS Redirection

As the adversary must observe an unencrypted connection to HTTPS website, which may not occur under all scenarios. However, a very typical scenario is for the victim to use the browser's address bar. Consequently, to understand the conditions under which the requirements will hold, we explore how popular browsers handle user input in the address bar (Figure 3.2). For example, the flow starts with when the user typing `google.com` in the address bar (1). The browser by default will send an HTTP request for the given URL (2). Although the user's secure (HTTPS) cookies are not sent over, as this request is over HTTP, the user's HTTP cookies (not set `secure` flag) are appended to this request. Since the server supports HTTPS, it sends an HTTP redirection (301 or 302) to its HTTPS page (3). The user's browser will receive the response from the server and automatically change `http://` to `https://` in the address bar (4). After that, the browser completes the TLS handshake and can communicate securely (5). This process seems to be very secure to users as the server and browser co-operatively redirect the user to a secure connection. However, the step 2 leaves a window of opportunity for attackers to steal the cookies.

To understand the conditions under which this occurs, we explore how popular browsers handle user input in the address bar when trying to visit `google.com`. As shown in Table 3.1, for straightforward user input, popular browsers will connect to `google.com` over HTTP, before the servers send redirection response to its HTTPS page. Additionally due to the auto-complete feature of certain browsers (e.g., Firefox v.41), even if the victim only types "`google`", the auto-complete mechanism will add "`.com`", and the browser will again connect over HTTP. Therefore, under common browsing patterns, the existing design will expose

Table 3.1: Browser behavior for user input in address bar.

Browser	Connect over HTTP
Desktop	
Chrome (v. 45)	✓
Firefox (v. 41)	✓
Safari (v. 8.0)	✓
Internet Explorer (v. 11)	✓
Opera (v. 32)	✓
Mobile	
Safari (iOS 9)	✓
Chrome (v.46, Android 5.1.1)	conditionally

*__user input:__ {`google.com`, `www.google.com`}

a user's cookie when visiting the main search engine. Interestingly, while the default iOS browser (Safari) exhibits the same behavior, Chrome on Android will connect to Google over HTTPS to securely prefetch page resources. However, if users turn this option off to improve performance[1], Android Chrome will also connect over HTTP.

3.3.2 Mixed Content and HTTP Link

Mixed content occurs when the main HTML page is loaded over HTTPS connection, but some resources on the page (e.g., images, videos, JavaScript) are loaded over an insecure HTTP connection. Websites contain mixed contents are likely to expose users to risks, as it allows man-in-the-middle attackers to change website functionality especially by modifying the HTTP request of contents of a web page's active resources e.g., script (`<script>`, `XMLHttpRequest`), CSS, fonts, and frames (`<iframe>`).

Mixed Content. While the active mixed content is blocked by popular browsers, the passive mixed content e.g., images, videos, is *not* [132]. When a website contains passive mixed content, the browser issues HTTP request to obtain the resource, which is sent over unencrypted.

[1] `https://support.google.com/chrome/answer/1385029?co=GENIE.Platform%3DAndroid&oco=1`

HTTP Link. There often times that web developers include absolute (full) HTTP URL links (e.g., `<a href="http://bank.com/login">`) instead of relative link (e.g., `<a href="/login"`) of the current HTTPS page (`https://bank.com`). HTTPS servers usually set up a redirection from HTTP to HTTPS page, therefore clicking these absolute HTTP URL eventually takes users to HTTPS secure page. As pointed out, HTTPS redirection makes users risk HTTP request to adversaries before switching to HTTPS.

We often found the mixed content and HTTP link when a website allows users to add own their HTML contents to their page (e.g., Ebay seller's item description (See Section 3.5.1 and Appendix A.1.1)).

3.3.3 Partial HSTS Deployment

As described in Section 2.5.1, setting HSTS response header lets a website force browsers to always access only with HTTPS. Major browsers (e.g., Chrome, Firefox, Safari) employ pre-loading lists for HSTS and websites deploy HSTS and HSTS preload. The incomplete HSTS deployment could be problematic, as it potentially allows some of the HTTP request and response to be sent over an insecure connection. To illustrate the situation, for example, in Listing 3.1, the preload HSTS policy of `google.com` for Chrome does *not* actually force the browser to connect to `google.com` over HTTPS. It does, however, employ certificate pinning; it requires an acceptable certificate if the browser is *already* connecting over HTTPS. As a result, users that visit the Google search engine through the address bar or access other subdomains that are protected by HSTS will most likely connect over an unencrypted channel, and their cookies will be exposed. As Firefox currently builds a custom list that is derived from the entries in Chrome's list that have `force-https` [138], Firefox users expose to this vulnerability as well. We will explore, discuss and evaluate HSTS and other current HTTPS enforcing mechanisms in detail in Chapter 4.

```
// (*.)google.com, iff using SSL, must use an acceptable certificate.
{"name": "google.com", "include_subdomains": true, "pins": "google"},

// Now we force HTTPS for subtrees of google.com.
{"name": "mail.google.com", "include_subdomains": true, "mode": "force-https",
    "pins": "google"},
{"name": "docs.google.com", "include_subdomains": true, "mode": "force-https",
    "pins": "google"},
```

Listing 3.1: Subset of rules in Chrome version 46.0's HSTS-preload file.

3.3.4 Persistent Cookie and Logout Invalidation

Our analysis also reveals that almost all sensitive HTTP cookies we found are persistent cookies with a very long period of an expiration date (Table 3.3). Those cookies do not instruct the browser to expire or delete upon exiting. Thus, attackers can maintain access to the victim's personal information and account functionality until the cookies' set expiration date which can be after several months (Google cookies expire after 2 years) to increase usability and avoid re-login.

Logout Invalidation. Invalidating cookies when a user logs out is standard practice. High-value services do so even after a short time of user inactivity. We examined whether the services also invalidate the HTTP cookies required for our hijacking attacks. We found that even if the user explicitly logs out after the attacker has stolen the cookies, almost all cookies still retain access privileges and can carry out the attack.

3.4 Information Leakage Study

In the following sections, we discuss our series of fundamental experiments designed to understand the practicality and severity of the cookie-hijacking attack. First, in Section 3.5 we conduct an in-depth analysis of real-world services to identify the type of information and specific set of cookies leaking over by the current attack surfaces covered in Section 3.3. Next, to measure the feasibility of potential impact of cookie hijacking attacks in practice, we conduct two network studies: (i) we monitor the outgoing unencrypted connections and

HTTP cookies that are exposed through our institute's wireless network, (ii) we monitor and explore evaluate potential risk of the attack on the tor network. The details of these studies can be found in Section 3.6 and Section 3.7 accordingly.

3.5 Analysis of Real-world Services

We conduct experiments on the ramifications of HTTP cookie hijacking attacks in real websites. We first aim to understand how prevalent HTTPS is, and how web developers separate information accessibility and functionality for encrypted and unencrypted connections. This entails what privileges are granted to HTTP cookies without requiring user authentication. Guided by these questions, we audit the top 25 Alexa websites from a varied collection of categories using test accounts (or our personal when necessary). We set up an experiment machine which connected to our experiment network similar to our threat model described in Section 3.2 and collect HTTP traffic in the network while perform automate browsing on a different host, over the targeted websites with Firefox and Chrome browsers in experiment machine using Selenium [2], such as logging in, clicking same-domain links and submitting a form. All targeted websites are tested with the default setting of browsers. Additionally, we audit browser components (i.e., browser extensions and apps) as well as the official iOS and Android apps from the targeted services. The detail of this analysis can be found in Section 3.5.2. Afterward, from all cookies we collect, we set the subsets of the cookies on each service in order to obtain which subset of cookies are necessary to reveal user information.

3.5.1 Real-world Privacy Leakages

This section describes the result of our analysis of each service. Table 3.2 presents an overview of the services and our results. We provide details on the private information and account functionality we are able to access with stolen cookies for certain websites and describe other classes of attacks that become feasible. Some certain services are described

[2] http://www.seleniumhq.org/

in Appendix A.1. Table 3.3 presents the set of required cookies for each service we analyze to perform cookie hijacking attacks and obtain the private information described.

3.5.1.1 Google and Google-related Services

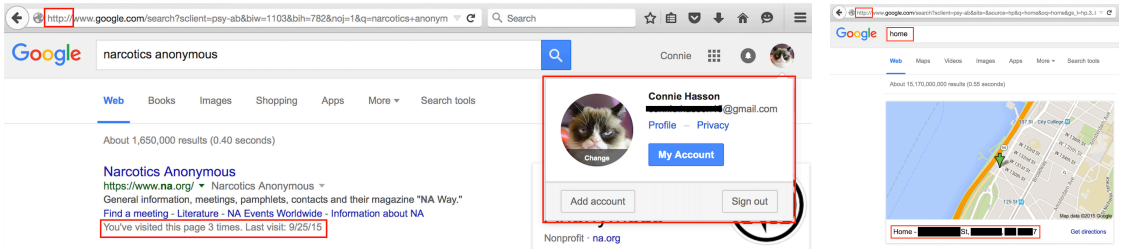
Cookie Hijacking. Typically, the adversary can steal the victim's HTTP cookie for Google by observing a connection to any page hosted on `google.com` for which encryption is not enforced. Additionally, Google automatically redirects users connecting over HTTP to `google.com` to HTTPS, to protect their searches from eavesdropping. However, upon the initial request, before being redirected and enforcing encrypted communication, the browser will send the HTTP cookies. Furthermore, the user can also use the address bar for visiting Google services; e.g., the user can type `"www.google.com/maps"` to visit Google Maps. Under these usage scenarios the browser will again expose the user's HTTP cookies, and if an adversary is monitoring the traffic, she can hijack them.

HSTS Preload. As described in Section 3.3, Google does not fully deploy HSTS on all of their domains. The main `google.com` does not enforce HTTPS on the domain itself and its subdomains. This is applied to all local country-based variations of Google's search engine (e.g., `google.co.uk`). On the other hand, only critical Google subdomains support HSTS preload and are explicitly forced to connect over HTTPS (see Figure 3.1).

Information Leakage. If the adversary simply visits `google.com` using the stolen cookie, no sensitive information will be accessible as the browser is redirected to HTTPS. However, if the adversary "forces" the browser to visit Google over HTTP, sensitive information can be accessed. During our auditing, we have identified the following.

Personal Information. Due to the cookie, Google considers the victim logged-in, resulting in personal information being leaked. As can be seen in Figure 3.3(a), we gain access to the user's name and surname, Gmail address, and profile picture.

Location. Google Maps allows users to set their Home and Work addresses, for easily obtaining directions to/from other destinations. While Google Maps requires HTTPS, which prevents us from acquiring any information, if the adversary connects to Google over HTTP and searches for `"home"` or `"work"`, the search results will contain a widget of



(a) Profile and History (b) Location

Figure 3.3: Private information obtainable from user's Google account through HTTP cookie hijacking.

Google Maps revealing the respective address. An example can be seen in Figure 3.3(b). Accessibility to location information can expose the user to physical threats [99, 154].

Browsing History. Using the stolen cookie, the adversary can start issuing Google searches for various terms of interest. If the search results contain links that the user has previously visited through the search engine, Google will reveal how many times the page has been visited and the date of the last visit. Users can opt-out of historical information being included in their search results, however, this option is enabled by default. If enabled, the adversary can search for a variety of terms and infer sensitive data about the user. Figure 3.3(a) shows an example scenario where the adversary obtains such information. Depending on the attacker's goal, she could employ a precompiled dictionary of sensitive keywords for finding sensitive web activity, or a dictionary of the most popular Google search terms for recovering parts of the user's web visiting history. While previous work demonstrated that unencrypted sessions could enable attackers to reconstruct a user's Google search history [45], this is the first, to our knowledge, attack that discovers web page *visited* by the user through Google. As users' browsing history are tied with these hijacked cookies, attackers could use these cookies in a similar fashion and bypass even more stringent safeguards that require extensive browsing history, e.g., bypassing the Google's noCAPTCHA reCAPTCHA challenges [161].

Exploiting Search Optimization. Google search may return results that have been personalized for the user, either by inserting specific entries or changing the rank of specific

results. Previous work has demonstrated a methodology for measuring personalization in Google search results [100]. By adopting this technique, the adversary can extract entries from the search results that have been returned based on characteristics of the victim's profile.

Pollution Attacks. If the attacker issues search queries using the stolen cookies, the search terms are treated as if originating from the user and added to the search history. This allows the adversary to affect the victim's contextual and persistent search personalization through pollution attacks [195].

Youtube. Youtube exhibits a strange behavior that we did not come across in other services. If the victim is logged in, the stolen cookie does not reveal any information. However, if the victim is not logged in, the cookie that is exposed gives access to the user's recommended channels and videos, which can be changed through pollution attacks. Furthermore, information about the user's music interests can be used to infer private attributes [46].

3.5.1.2 Bing

By default, all connections are served over HTTP, i.e., all searches are sent in clear-text. Users have to explicitly type `https` in the browser's address bar to be protected from eavesdropping.

Personal Information. Bing will expose the user's first name and profile photo. The profile photo can be used to obtain more information about the user through face recognition and publicly available data in other websites [20]. Additionally, if the victim has saved any locations on Bing Maps they are also exposed. Apart from the work or home addresses, this may include other locations the user has visited in the past (e.g., bars, health clinics).

Search and Browsing History. Once the adversary steals the cookie, she can retrieve the user's search history, including those in the images and videos categories. Apart from a widget displaying the users most recent and most frequent search queries, the search history page also reveals the page that the user visited from each search.

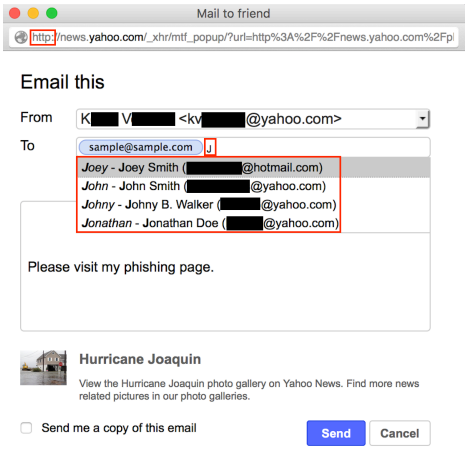


Figure 3.4: Extracting contact list and sending email from the victim's account in Yahoo.

3.5.1.3 Yahoo

We have identified three main HTTP cookies (Table 3.3) (assigned to `yahoo.com` domain) that are functionally significant and exposed to eavesdroppers. Due to cookie domain scope problem mentioned in Chapter 2.4.1, we also found that these cookies allow attackers to obtain other information presenting in other Yahoo services presenting in different subdomains of `yahoo.com`.

Personal Information. The cookies set for `yahoo.com` allow the attacker to obtain the user's first name. The full last name and email address can also be obtained, as we explain below.

Yahoo Mail. To facilitate sharing posts with friends, articles in Yahoo contain an "Email to friends" button, which presents a popup window in which the adversary can add an arbitrary message, as shown in Figure 3.4. Furthermore, the `Sender` field has auto-complete functionality, which allows us to obtain the victim's complete contact list. These features combined can be leveraged for deploying effective phishing or spam campaigns. The widget also contains the user's full name and email address. Extracting the contacts requires all three cookies set for the main domain while sending the email requires them for the `news.yahoo.com` or the `finance.yahoo.com` subdomain depending on which section the article is located in.

If the user hovers over or clicks on the mail notification button, the attacker can also access the incoming mail widget, which reveals the `Sender` and partial `Subject` (up to 21 characters) of the 8 most recent incoming emails. This is due to a cookie being attributed an "authenticated" status. This lasts approximately one hour, after which it cannot access the widget. If at any point the user accesses the notification button again, the hijacked cookie is re-authorized.

Yahoo Search. Having acquired the main domain and search subdomain cookies, the adversary can gain access to the victim's complete search history. Apart from viewing the searched terms, these cookies allow editing the history and removing previous searches. However, Yahoo explicitly states that even if past searches are deleted, user search data is still logged. This enables *stealthy* pollution attacks; after issuing search queries for influencing the personalization profile of the user, the adversary can then delete all issued searches and remove traces of the attack.

Other Yahoo Services. Upon auditing Yahoo, we found that the victim's HTTP cookie allows partial control over the account; the adversary is able to ask or answer questions (either eponymously or anonymously) in Yahoo Answers, and also to view and edit previous questions and answers posted by the victim. Thus, the adversary can effectively "deanonymize" posts and obtain potentially sensitive information about the victim, which was posted under the assumption of anonymity.

3.5.1.4 E-commerce Websites

We analyze the impact of hijacking over a number of popular e-commerce websites. We also found a recurring pattern across these services. The detail on other e-commerce services (i.e., Target, Walmart and Ebay) are described in Appendix A.1.1. As an example, we describe Amazon as an example.

Amazon. The homepage follows the common approach of redirecting to HTTPS if connected to over HTTP. However, product pages are served over HTTP and, as a result, users' cookies will be exposed during their browsing sessions.

Personal information. The adversary can obtain the information used by the victim for logging in; this includes the victim's username, email address and/or cell phone number. Furthermore, when proceeding to checkout items in the cart, Amazon also reveals the user's full name and city (used for shipping). Viewing and changing the user's profile picture is also permitted. Amazon also allows users to post their reviews under a pseudonym, which is not connected to the user's name. However, the adversary can view the user's reviews (which may include sensitive items), thus, breaking the pseudonymous nature of those reviews. Previous work has demonstrated the privacy risks of recommender systems and experiments in Amazon indicated that sensitive purchases can be inferred from the user's review history [40].

Account history. The user's HTTP cookie is sufficient for accessing private information regarding previous actions. Specifically, the adversary can obtain information regarding recently viewed items, and recommendations that are based on the user's browsing and purchase history. The wish-lists where the user has added items of interest are also accessible. Furthermore, the adversary can obtain information regarding previously purchased items either through the recommendation page or through product pages (which depict date of purchase). In an extensive study on privacy-related aspects of online purchasing behavior [180], users rated the creation of a detailed profile from their purchase history and other personal information as one of the most troubling scenarios.

Shopping cart. The user's cart is also accessible, and the adversary can see the items currently in the user's cart. Additionally, the cart can be modified, and existing items can be removed, and other items can be added.

Vendor-assisted spam. We also found that the cookie exposes functionality that can be leveraged for deploying spam campaigns to promote specific items that are presented as "endorsed" by the victim. The widget has an auto-complete feature that reveals the contacts that the user has emailed in the past. The attacker can either send emails about a specific item or a wish-list and can add text to the email's body. URLs can be included; while the email is sent as simple text, email providers such as Gmail render it as a clickable link. Since the emails are actually sent by Amazon (`no-reply@amazon.com`), they are most likely to pass any spam detection heuristics. Furthermore, the `From` field contains

the victim's username, further strengthening the personalized nature of the spam/phishing email.

Extortion scams. Previous work has revealed how scammers extorted money from users through *One Click Fraud* scams by threatening to reveal "embarrassing" details about the users' online activities [49]. In a similar vein, the attacker can employ two different scam scenarios. In the first case, if the attacker identifies potentially embarrassing item(s) in the user's viewing or purchase history, she can send an email to the user disclosing knowledge about the item(s), and other personal information obtained about the user, and request money to not share that information with the user's contacts (even if no contact information has been collected). In the second scenario, the attacker can send an email blackmailing the user to pay money otherwise she will send an email to the victim's friends and family with information about his cart that is full of embarrassing items. Subsequently, the attacker will add such items to the user's cart or wish list, and send the corresponding email through Amazon to the victim's own email address as proof of her capabilities.

3.5.1.5 News Media

Information acquired from media outlets can reveal characteristics and traits of the user (e.g., political inclination), and demographic information [87]. We audited the websites of several of the most popular print or broadcast news organizations (see Appendix A.1.2).

3.5.1.6 Indirect Information Exposure - Ad Networks

We explore the impact of hijacking ad network cookies. Online ads account for a significant portion of website real estate, and their ubiquitous nature has been discussed extensively in the context of user tracking (e.g., [120, 129, 157]). Here we focus on Doubleclick, as it is the most prevalent advertising network with a presence on 80% of the websites that provide advertisements [86]. While the symbiotic nature of service providers and data aggregators are complicated, the ads presented while browsing with stolen user cookies from ad networks can be used to infer sensitive information. An interesting aspect of hijacking ad-network cookies is that they result in *side-channel information leakage*. We describe a scenario we discover during our experiment which leaks browsing history and behavior.

Attack Scenario. User `U` is browsing through an e-commerce site `E`, which uses the ad network `A` to advertise its products on other websites. `U` searches for items that belong to a specific category `C`, and after the site returns a list of relevant products, `U` clicks on a link and views the page of product `P`. A short time later, the attacker visits an unrelated website that is known to show various ads, and appends `U`'s stolen HTTP cookie for the ad network `A`. The attacker is then presented with several ads relevant to `U`'s browsing history. Some are more generic and expose information about `U`'s gender, while others explicitly refer to category `C` and even depict the specific item `P`.

Information Leakage. We conducted a small number of experiments for identifying cases of personal information being leaked by Doubleclick. Previous work has shown that ads presented to users may be personalized based on the user's profile characteristics [31].

Here we describe one of our experiments. We browsed maternity clothes on a popular e-commerce website and visited the page of a few returned products. We then browsed other sites from a different machine connected to a different subnet and appended the Doubleclick HTTP cookie from the previous browsing session. We were presented with ads from the e-commerce website advertising women's clothing. Several ads even advertised a specific maternity product whose page we had visited (see screenshots in Appendix A.1.3). Depending on the time lapsed between the user browsing the e-commerce site and the attacker browsing with hijacked cookies, there is a decrease in the frequency of ads that contain the viewed product. However, we found that even after several hours received ads that continued to promote the exact product and women's clothing ads even after several days.

Table 3.2: Overview of the audited websites and services, the feasibility of cookie hijacking attacks, and the type of user information and account functionality they expose.

Service	HTTPS Adoption	Cookie Hijacking	XSS Cookie Hijacking	Information and Account Functionality Exposed
Google	partial	✓	✗	Full name, username, email, profile picture, home and work address, search optimization, visited websites returned in search results
Baidu	partial	✓	✓	Username, email, profile picture, full search history, saved locations
Bing	partial	✓	✓	Name, profile picture, view/edit search history (incl. images and videos), links clicked from search results, frequent search terms, saved locations, view/edit interest manager
Yahoo	partial	✓	✓	Full name, username, email, view/edit search history, view/edit/post answers and questions in Yahoo Answers (anonymous or eponymous), view/edit finance portfolio, view subject and sender of latest incoming emails, extract contact list and send email as user
Youtube	partial	✓	✗	View/edit (through pollution attacks) recommended videos and channels
Amazon	partial	✓	✓	User credential (username, email or mobile number), delivery name and city, profile picture, view user's activities (user's wish lists, recommended items, recently bought items, browsed items, user's review (even anonymous)), view/edit items in cart, view current balance, send email of products or wish list on behalf of user, view emails of previously emailed contacts
Ebay	partial	✓	✓	Delivery name and address, view/edit user's activities (items in cart, purchase history, watch list and wish lists), view items for sale, previous bids, user's messages
MSN	partial	✓	✓	Full name, email, profile picture
Walmart	partial	✓	✓	Name, email, view/edit items in cart, view delivery postcode, write product review
Target	partial	✓	✓	Name, email, view recently browsed items, view/edit items in cart and wish list, send email about products on behalf of user
CNN	partial	✓	✓	View/edit profile (full name, postal address, email address, phone number, profile picture, linked Facebook account), write/delete article comments, recently read article on iReport
New York Times	partial	✓	✓	Username, email, view/edit profile (display name, location, personal website, bio, profile picture) view/edit list of saved articles, share article via email on behalf of user
Huffington Post	partial	✓	partial	View/edit (login name, profile photo, email, biography, postal code, location, subscriptions, fans, comments and followings), change account password, delete account
The Guardian	partial	✓	✓	Username, profile picture, interests, comments, replies, tags and categories of read articles, post comments on articles as user
Doubleclick	partial	✓	✓	Ads show content targeted to user's profile characteristics or recently viewed content
Skype	partial*	✗	✗	-
LinkedIn	partial*	✗	✗	-
Craigslist	partial*	✗	✗	-
Chase Bank	partial*	✗	✗	-
Bank of America	partial*	✗	✗	-
Facebook	full	✗	✗	N/A
Twitter	full	✗	✗	N/A
Live (Hotmail)	full	✗	✗	N/A
Gmail	full	✗	✗	N/A
Paypal	full	✗	✗	N/A

*While these services do not have ubiquitous HTTPS, no personalization is offered over HTTP pages.

Table 3.3: The set of HTTP cookies (name and value pairs) which are required for hijacking user information as described in Table 3.2. The expiration duration is the duration that the cookie stays valid from the creation time (usually login time). "SESSION" indicates the cookie is a session cookie (not set expiration date) which remains on the user's browser until the browser is closed.

Service	Required HTTP Cookies	Expiration Duration (days)
Amazon	x-main	7305 (20y)
Bing	_U	14
	WLS	SESSION
Baidu	BDUSS	3000 (8y)
CNN	CNNid	365 (1y)
	authid	365 (1y)
Doubleclick	id	731 (2y)
Ebay	cid	365 (1y)
	nonsession	365 (1y)
Google	HSID	731 (2y)
	SID	731 (2y)
Guardian	GU_U	90
HuffingtonPost	huffpost_s	365 (1y)
	huffpost_user	365 (1y)
	huffpost_user_id	365 (1y)
	last_login_username	365 (1y)
MSN	MSNRPSAuth	SESSION
New York Times	NYT-S	365 (1y)
Target	WC_PERSISTENT	30
	guestDisplayName	90
	UserLocation	90
Walmart	customer	731 (2y)
	CID	731 (2y)
Yahoo	F	366 (1y)
	T	366 (1y)
	Y	366 (1y)
Youtube	VISITOR_INFO1_LIVE	243

3.5.2 Collateral Cookie Exposure

In this section, we explore and analyze other means by which a user's HTTP cookies may be exposed.

Table 3.4: Cookie exposure by popular browser extensions and apps.

Name	Type	Browser	#	Cookie leaked
Google Maps	app	Chrome	N/A	✓
Google Search	app	Chrome	N/A	✓
Google News	app	Chrome	1.0M	✓
Amazon Assistant	extension	Chrome	1.1M	✓
Bing Rewards	extension	Chrome	74K	✓
eBay for Chrome	extension	Chrome	325K	✓
Google Dictionary	extension	Chrome	2.7M	✓
Google Hangouts	extension	Chrome	6.4M	✗
Google Image Search	extension	Chrome	1.0M	✗
Google Mail Checker	extension	Chrome	4.2M	✗
Google Translate	extension	Chrome	5.5M	✗
Yahoo Mail Notification	extension	Chrome	1.2M	✗
Amazon	default search bar	Firefox	N/A	✓
Bing	default search bar	Firefox	N/A	✗
Ebay	default search bar	Firefox	N/A	✓
Google	default search bar	Firefox	N/A	✗
Yahoo	default search bar	Firefox	N/A	✗
Amazon 1Button	extension	Firefox	157K	✓
Bing Search	extension (unofficial)	Firefox	28K	✓
eBay Sidebar	extension	Firefox	36K	✓
Google Image Search	extension	Firefox	48K	✓
Google Translator	extension (unofficial)	Firefox	794K	✓
Yahoo Toolbar	extension	Firefox	31K	✓

3.5.2.1 Browser Components

According to a manifest file analysis of over 30K Chrome extensions [112], a higher number of extensions requested permission for connecting to Google over HTTP compared to HTTPS. The same was true for wildcarded (`http://*/*`) permission requests. This indicates that a considerable number of extensions may be weakening security by connecting over unencrypted connections to websites that also support encrypted connections. To that end, we explore whether browser components expose users to cookie hijacking attacks.

We analyze a selection of the most popular browser components, for Chrome and Firefox, that have been released by major vendors we have audited. Our aim is not to conduct an exhaustive evaluation, but to obtain an understanding of the implementation practices for browser components and assert whether they also suffer from a limited use of encryp-

tion. While we experiment with a relatively small number of components, we consider any discovered exposure indicative of general practices, as official extensions from major vendors are likely to adhere to certain quality standards. As Google has discontinued the development of extensions for Firefox, we cannot do a direct cross-browser comparison for most of its components.

Table 3.4 lists the web components we have evaluated, their reported number of downloads if available, and if they leak the cookies required for our hijacking attacks. Our experiments yield a number of surprising findings. The 3 Chrome apps released by Google we tested expose the HTTP cookies, while their extensions present mixed results with 4 out of 9 leaking the cookie. As one of those is Google Dictionary, with over 2.7 million downloads, a significant number of Chrome users is vulnerable to considerable risk.

Every Firefox extension we tested, along with two of the default search bars, actually expose the required HTTP cookies over unencrypted connections. Interestingly, Google's Search by Image extension is secure for Chrome but not for Firefox. As there is no official Bing app for Firefox, we test the most popular one, and we also audit a popular unofficial Google translator extension with over 794K users, both of which turn out to be vulnerable. Overall, these findings highlight the privacy threats that millions of users face due to browser components.

3.5.2.2 Mobile Devices

Mobile devices have become ubiquitous, and account for a large part of the time users spend online. To explore the feasibility of our HTTP cookie hijacking attacks against users on mobile devices, we audited the official iOS and Android apps for the most popular services that we found to expose private information and account functionality.

The overview of our results is shown in Table 3.5. Once again Yahoo follows poor security practices as 3 out of 4 iOS apps leak the user's cookies. As expected both versions of Gmail protect the cookies, while iOS Amazon apps prior to version 5.3.2 expose the cookie. Furthermore, both Amazon iOS apps contain cookies that reveal information about the user's device and mobile carrier (details in Appendix A.1.1). For both platforms, the Ebay app will expose the cookies under certain conditions. First, Ebay sellers are allowed

Table 3.5: Cookie exposure by official mobile apps.

Application	Platform	Version	#	Cookie leaked
Amazon	iOS	5.3.2	N/A	✗
Amazon	iOS	5.2.1	N/A	✓
Amazon	Android	28.10.15	10-50M	✗
Bing Search	iOS	5.7	N/A	✓
Bing Search	Android	5.5.25151078	1-5M	✓
Spotlight (Bing)	iOS	iOS9.1	N/A	conditionally
Siri (Bing)	iOS	iOS9.1	N/A	✗
Ebay	iOS	4.1.0	N/A	conditionally
Ebay	Android	4.1.0.22	100-500M	conditionally
Google	iOS	9.0	N/A	✗
Google	Android	5.4.28.19	1B+	✗
Gmail	iOS	4.1	N/A	✗
Gmail	Android	5.6.103338659	1-5B	✗
Google Search Bar	Android	5.4.28.19	N/A	✗
Yahoo Mail	iOS	4.0.0	N/A	conditionally
Yahoo Mail	Android	4.9.2	100-500M	✗
Yahoo News	iOS	6.3.0	N/A	✓
Yahoo News	Android	18.10.15	10-50M	✗
Yahoo Search	iOS	4.0.2	N/A	✗
Yahoo Search	Android	4.0.2	1-5M	✗
Yahoo Sports	iOS	5.7.4	N/A	✓
Yahoo Sports	Android	5.6.3	5-10M	✗

to customize their item pages and often add links to other items they are selling; if the seller has added an HTTP Ebay link to those items, the cookie will be exposed if a link is clicked by the user. Empirically we found that these HTTP links are common. The other scenario is if the user clicks on the "Customer Support" menu.

3.6 Network Traffic Study

The feasibility of cookie hijacking attacks by eavesdroppers is dependant on the browsing behavior of users when connected to public wireless networks. Our goal is to understand the browsing patterns of users connecting to public wireless networks and measure the feasibility of exploring the potential impact of cookie hijacking attacks in practice. We conduct an exploratory study of the traffic passing through the public wireless network of

our university's campus. Specifically, we monitor the outgoing unencrypted connections and HTTP cookies that are exposed by users.

3.6.1 IRB

Before conducting any experiments, we submitted a request to our Institutional Review Board that clearly described our research goals, collection methodology, and the type of data to be collected. Once the request was approved, we worked closely with the Network Security team of our university's IT department for conducting the data collection and analysis in a secure and privacy-preserving manner.

3.6.2 Data Collection

In order to collect the data, we set up a logging module on a network tap that received traffic from multiple wireless access points positioned across our campus. The RSPAN [51] was filtered to only forward outgoing traffic destined to TCP ports 80 and 443, and had a throughput of 40-50 Mb/s, covering approximately 15% of the public wireless outgoing traffic. Our data collection lasted for 30 days. We used the number of TCP SYN packets to calculate the number of connections. When the connection is over HTTP or HTTPS, we capture the destination domain name through the HTTP host header and the TLS SNI extension respectively. For each HTTP request, we log the destination domain, and the name of any HTTP cookies appended (e.g., SID). We also calculated a HMAC of the cookie's `value` (the random key was discarded after data collection). The cookie names allow us to verify that users are logged in and susceptible to cookie hijacking for each service, as we have explored the role of each cookie and also identified the subset required for the complete attack (described in Section 3.5.1).

While we do not log the cookie value for privacy reasons, the keyed hash value allows us to distinguish the same user within a service to obtain a more accurate estimation of the number of exposed accounts. We must note that our approach has limitations, as the numbers we estimate may be higher than the actual numbers; a user's cookie value may have changed over the course of the monitoring period or the user may use multiple devices (e.g., laptop and smartphone). However, some services employ user-identifier cookies, which we

leverage for differentiating users even if the other cookie values have changed. Furthermore, we cannot correlate the same user across services as we do not collect source IP addresses or other identifying information; thus, we refer to vulnerable *accounts*. Nonetheless, we consider this to be a small trade-off for preserving users' privacy and consider our approximation accurate enough to highlight the extent of users being exposed when browsing popular services.

3.6.3 Findings

Table 3.6 presents the aggregated numbers from the data collected during our study. During our monitoring, we observed more than 29 million requests towards the services that we have found to be vulnerable. This resulted in 282,459 accounts exposing the HTTP cookies required for carrying out the cookie hijacking attacks and gaining access to both their private information and account functionality. Figure 3.5 breaks the numbers down per service. Search engines tend to expose many logged in users, with 67,201 Google accounts being exposed during our experiment. Every category of services that we looked at has at least one very popular service that exposes over ten thousand users during the monitoring period. Ad networks also pose a significant risk, as they do not require users to login and ads are shown across a vast number of different websites, which results in Doubleclick exposing more than 124K users to privacy leakage.

Table 3.6: Statistics of outgoing connections from a subset of our campus' public wireless network for 30 days.

Protocol	Connections	Requests	Vulnerable Requests*	Exposed Accounts
HTTP	685,500,365	1,398,044,178	29,908,099	282,459
HTTPS	772,562,024	--	--	--

*HTTP requests to domains that we have audited and found to be vulnerable.

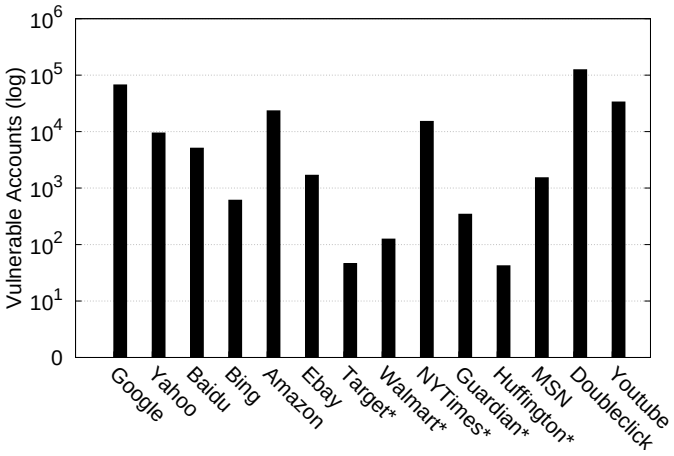


Figure 3.5: Number of exposed accounts per service. Services marked with "*" have an explicit userID cookie (or field) that allows us to differentiate users.

3.7 Deanonymization Risk for Tor Users

In this section, we investigate if more privacy-conscious users are protected against our presented cookie hijacking attacks. Specifically, we explore how users employing the Tor bundle (Tor Browser with pre-installed extensions) can be deanonymized by adversaries. The Tor bundle offers significant protection against a variety of attacks including HTTPS Everywhere [72], a browser extension which is designed to helps reduce number of HTTP connection by rewriting HTTP requests to HTTPS (Note that we will explore the effectiveness of HTTPS Everywhere and other HTTPS enforcement mechanisms in detail in Chapter 4). However, we found that its effectiveness in mitigating cookie hijacking attacks varies greatly depending on each website's implementation. In this case, we consider a variation of the threat model from the previous sections; the adversary monitors Tor exit nodes instead of public wireless access points.

3.7.1 Evaluating Potential Risk

We want to explore whether privacy-conscious users actually visit these major websites over the Tor network, or if they avoid them due to the lack of ubiquitous encryption.

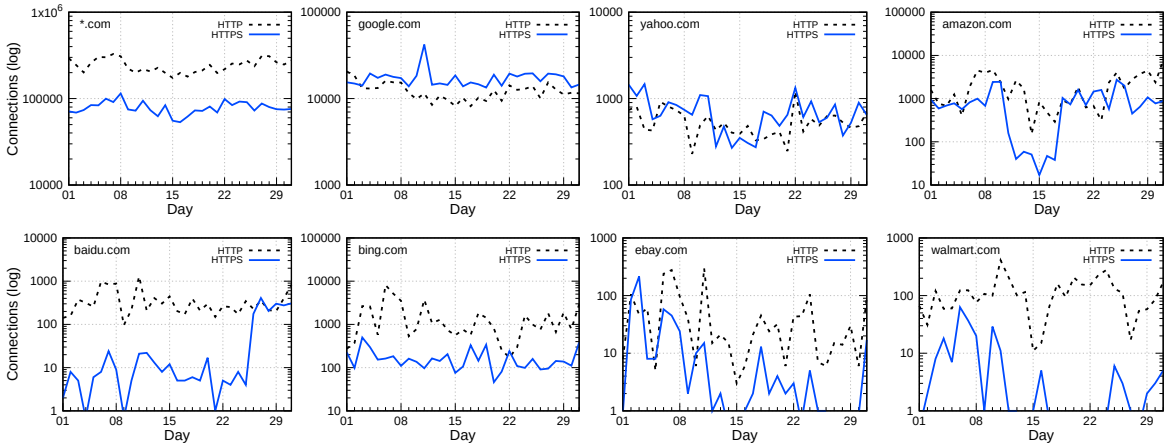


Figure 3.6: Number of encrypted and unencrypted connections per day, as seen from a freshly-deployed Tor exit node.

3.7.1.1 Ethics

Again, we obtained IRB approval for our experiments. However, due to our ethical considerations for the Tor users (as they are not members of our university nor connecting to our public wireless network), we do not replicate the data collection we followed in our experiment from Section 3.6. We opt for a coarse-grained non-invasive measurement and only count the total connections towards the websites we audited in Section 3.5.1, using the port number to differentiate between HTTP and HTTPS. *We do not log other information, inspect any part of the content, or attempt to deanonymize any users.* Furthermore, *all data was deleted* after calculating the number of connections. Since we do not look at the name of the cookies sent in the HTTP connections, we cannot accurately estimate the number of users that are susceptible to cookie hijacking attacks. Our goal is to obtain a rough approximation of the number and respective ratio of encrypted and unencrypted connections to these popular websites. Based on the measurements from our university's wireless trace, we can deduce the potential extent of the deanonymization risk that Tor users face. We consider this an acceptable risk-benefit trade-off, as the bulk statistics we collect do not endanger users in any way, and we can inform the Tor community of a potentially significant threat they might already be facing. This will allow them to seek countermeasures for protecting their users.

3.7.1.2 Tor Exit Node

We deployed a fresh exit node for this experiment. The number of outgoing connections was measured over 1 month, on a fresh exit node with a default reduced exit policy[3] and bandwidth limited to 300 KB/s.

3.7.1.3 Measurements

Figure 3.6 presents the number of total TCP connections and broken down for some services. The number of TCP connections over HTTP accounts for 75.4% of all the connections we saw, with an average of 10,152 HTTP and 3,300 HTTPS connections per hour. For most of the services, the unencrypted connections completely dominate the outgoing traffic to the respective domains. On the other hand, for Google, we observe an average of 508 HTTP connections per hour as opposed to 705 HTTPS connections. Similarly, we logged 23 unencrypted connections to Yahoo per hour and 36 encrypted connections. We do not consider the Doubleclick side channel leakage attack for Tor, as the double key session cookies employed by the Tor browser affect third-party cookies and their ability to track users across domains.

3.7.1.4 Susceptible Population

We see that there is a significant amount of HTTP traffic exiting Tor and connecting to popular websites that expose a vast collection of private user information. While the ratio of unencrypted connections is even higher than that of our university's network, possibly fewer users will be logged in when using Tor. More experienced users may be aware of the shortcomings of this mechanism and avoid the pages and subdomains that are not protected when connecting over untrusted connections. Nonetheless, we expect that many users will exhibit normal browsing patterns, thus, exposing their cookies to attackers. Furthermore, even though we can not know how many of the users are indeed logged in and susceptible to cookie hijacking (that would require looking at the cookie names), for some websites observing encrypted connections is an almost definitive sign that we are also observing HTTP

[3] `https://trac.torproject.org/projects/tor/wiki/doc/ReducedExitPolicy`

traffic of logged in users; due to functionality breaking and the corresponding exceptions in the HTTPS Everywhere rule-sets, HTTPS traffic for Amazon and Baidu signifies account-related functionality that requires users to be logged in (e.g., Amazon checkout) and is accompanied by HTTP traffic (Amazon products pages). Thus, we believe that a considerable number of Tor users may be facing the risk of deanonymization through hijacked cookies.

User Bias. As this is a newly deployed exit node, the population of users connecting to it may be biased towards inexperienced users, as more privacy-conscious ones may avoid exiting from such nodes. Thus, our observed ratio of encrypted connections or the websites which users connect to, may present differences to other exit nodes. Nonetheless, adversaries could already own exit nodes with long uptimes, or be able to monitor the outgoing traffic from legitimate exit nodes, which is a common adversarial model for Tor related research [64, 107]. Thus, we believe this to be a credible and severe threat to Tor users that want to maintain their anonymity while browsing (popular) websites.

3.8 HTTPS Deployment Guideline

In this section, given the insight from our analysis and attack results, we provide a summary of HTTPS deployment guideline particularly for web developers and administrators as a take away for this dissertation.

HTTPS Ubiquitous Deployment. Oftentimes, we observe websites deploy HTTPS only on some sensitive pages (e.g., login, payment, setting). Given the fact that HTTP cookie does not have a strong integrity property, attackers could potentially steal HTTP cookie to reveal users' information (Section 3.5.1) perform a cookie injection attacks (Section 2.4.1) on some of the websites' related domains (or subdomains).

HTTPS Redirection. As described in Section 3.3, major browsers by default attempt to connect web users to `http://` if the users do not specify `http://`. Specifically, typing `"www.example.com"` or `"example.com"` on their address bar, browsers open connection to

`"http://www.example.com"`, `"https://example.com"`, respectively. Without the help of server HTTPS redirection, users remain on HTTP pages.

In addition, redirection should be done with 301 redirection. This ensures search engines recognize the site permanently move to a new page. The HTTPS redirection is mandatory for the websites that deploy HSTS as the HSTS header is only set from HTTPS header.

HTTP Cookie. In addition to the integrity implications of HTTP cookie, numerous services still do not set `Secure` directive on HTTPS. Setting `Secure` directive ensures the cookie only sent through HTTPS requests. However, in Section 3.5.1, our experiment also revealed that major services maintain sensitive cookies (cookies that provide accessibility to user sensitive information) on cookie non-`secure` set.

Although some cookies might not be sensitive (e.g., language preferences) and is believed to be "ok" to not have the `secure` directive, this still opens feasibilities of user tracking when integrated with other website cookies, as discussed in the related work chapter (Chapter 2). Furthermore, with a combination of different cookies provides different access control to user information, carelessly setting `secure` on some cookies possibly leak some of the user information.

Therefore, we recommend services to set `"secure"` to all of their cookies. Services might also consider to limit the expiration times and invalidate when user logout (Section 3.3).

HTTP Strict Transport Security. Deploying HTTPS with HTTPS redirection by default are still vulnerable to HTTPS downgrade attacks (See Section 2.4). The Strict-Transport-Security (HSTS) is necessary as presented in Section 2.5.1. First, the setup of HSTS *must* be done in HTTP header on HTTPS *all* responses when possible. Setting HSTS header on HTTP potentially allow to intercept and remove the HSTS header. Due to this reason, Firefox does not honor the HSTS header setting on HTTP [133].

includeSubdomains. Services should also set `includeSubdomains` directive on their *base domain*. This enforces their all subdomains to always connect to HTTPS, in turns, reduce the feasibilities of cookie injection and cookie hijack on their related domains.

max-age. The `max-age` directive indicates the duration when the HSTS policy is expired. According to the HSTS RFC specification, user agents must honor and update

to the freshest information receive on HSTS. Therefore, the value in `max-age` is continually updated from every new HSTS in HTTP response. It is recommended that developers should set the `max-age` to be between six months (`max-age=15768000`) to two years (`max-age=63072000`) [137]. The HSTS `max-age` less than that increases the risk of HTTP downgrade attack due to likely chances of exposing of HTTP requests and responses.

HSTS Preload. HSTS Preload is designed to reduce the chance of being attacks in the initial connection (Chapter 2.5.1). We also recommend deploying HSTS Preload when possible. In addition to setting dynamic HSTS header, developers need to follow other requirements from HSTS Preload [103].

Mixed Content. In Section 3.3, we pointed out that passive mixed content could potentially leak user's cookies as the request is sent over HTTP. In Section 4.2, we also studied two content security policies (i.e., upgrade-insecure-requests and block-all-mixed-content) designed to eliminate retrieving content over HTTP and therefore eliminate the HTTP requests. These policies are not designed to enforce HTTPS, rather quick modify content's URLs which could still point to HTTP resources. Therefore they do not replace the need of HSTS.

3.9 De Facto Challenges in Deploying HTTPS Ubiquitously

As discussed, the incompleteness of HTTPS deployment and the need to support HTTP can potentially lead to user information leaks. In this section, we attempt to shed light on de facto challenges and difficulties in migrating to HTTPS ubiquitously, i.e., not only support HTTPS but insist on using it.

3.9.1 Performance

Performance is always one of the biggest concerns in deploying HTTPS as a default, given the fact that connecting with HTTPS requires additional network round trips, which doubles the delay in connecting to the servers when compared to traditional HTTP. These additional round trips are caused by the TLS handshake that is needed in order to use HTTP on TLS. In addition, deploying HTTPS ubiquitously requires deploying HTTPS on

every subdomain and page. To make matters worse, the network latency introduced from longer distances between clients and servers significantly increases this delay and is expected to impact user engagement, since one-second delay could cost 1.6 billion dollars in sales, as reported from Amazon [69]. In addition, there are also delays caused by performing necessary cryptographic computations e.g., encryption, decryption, hash, and certificate validation.

Full and Abbreviated TLS Handshake. While it is mandatory to perform the TLS handshake and end up adding up a delay to overall latencies of HTTPS connection, the delay imposed by this process can be optimized by opting for abbreviated TLS handshake whenever possible using session resumption [9]. Both full and abbreviated TLS handshakes are part of TLS handshake standard [10]. A study from CloudFlare showed that using session resumption is able to reduce over 55% latency from full TLS handshake including reducing latency caused by CPU consumption from client [123]. Currently, major services such as Facebook, Amazon deploy the abbreviated TLS handshake [78, 110]. Furthermore, the new version of TLS, TLS 1.3, is also designed to optimize the handshake process by reducing the number of handshake round trips.

3.9.2 Backward Compatibility

While major web browsers already support HTTPS, HSTS [41], and the majority of web users are capable to connect with HTTPS, in this chapter we showed that web services still continue to support HTTP and not enforce HTTPS connections, especially due to loss of functionality to maintaining support for legacy clients and services. For example, some services (e.g., Google) opt to deploy HSTS or HSTS preload only on some sensitive subdomains (e.g., Gmail) [131]. Particularly, deploying HTTPS ubiquitously (TLD+1 level) is challenging given the fact that deploying HSTS at the TLD+1 requires *all pages and subdomains to fully support HTTPS*. In other words, the developers need to ensure HTTPS support for all services and resources they provide and by default require HTTPS connections. This also extends to all application program interfaces (API) deployed on any of their subdomains as well as other third-party resources. The continuity to support HTTP for backward compat-

ibility (e.g., allowing some users' functionalities and personalizations) means maintaining non-secure accessibility on HTTP cookies. For example, although Google is fully aware of this problem, the service still prefers not to deploy HSTS (enforcing HTTPS) on their main pages [95].

3.9.3 Third-party Content

The necessity to support third-party contents on HTTPS pages also presents challenges to developers. For example, since the business model of modern websites often involve serving advertisements to users generated via third-party services (e.g., affiliates and partners) [130], services often require loading passive contents (e.g., images, videos) executing active contents (e.g., JavaScript, frame) including monitoring user behaviors and preferences in order to serve ads tailored to the user [157]. As the active mixed content is blocked from major browsers (Section 3.3) and not all ad networks provide contents on HTTPS, migrating to HTTPS affects the websites' revenues. This extends to other third-party contents (e.g., social networking, web analytics) on all pages to be loaded over HTTPS.

3.9.4 Infrastructure

As with any security solution, HTTPS does not come for free. We explore the additional infrastructure costs in deploying HTTPS.

TLS Certificate. A valid certificate is a basic requirement of deploying HTTPS. Depending on the complexity of the services and features, obtaining a certificate from traditional certificate authorities could be costly ($1,999 per year) [63]. Deploying HTTPS ubiquitously requires acquiring a certificate that covers all service's hostnames (including all subdomains). Thus, wildcard certificate (e.g., `*.example.org`) feature is necessary. The developers can opt for Let's Encrypt [76] which provides free certificates.

Content Delivery Network. As mentioned earlier, any additional round trips affect users, especially when users' network latencies are already noticeable. Deploying content delivery networks (CDN) closer to users eventually reduces round-trip time, and in turn,

reduces the total overhead that is caused by the encryption process [78]. With multiple servers, deploying a session resumption requires an extra effort in handling session IDs or session tickets [98, 178]. Currently, several CDN vendors provide support for this [108].

3.10 Ethics and Disclosure

To ensure the ethical nature of our research, we provided a detailed description of our data collection and analysis process to Columbia University's IRB, and obtained approval for both our experiments with the public wireless network (Protocol number IRB-AAAQ4105, titled "Measuring unencrypted connections in public wireless network") and the Tor network (Protocol number IRB-AAAQ7089, titled "Measuring unencrypted connections in Tor network"). Furthermore, all captured data was destroyed after the end of our evaluation measurements.

Disclosing attacks against popular services raises ethical issues as, one might argue, adversaries may have previously lacked the know-how to conduct these attacks. However, the practicality of cookie hijacking suggests that such attacks could soon happen in the wild (if not happening already). To that end, we have already contacted all the audited websites security contact point or vulnerability disclosure portal, and other services through their privacy policy contact points. We disclose our findings in detail. We believe that by shedding light on this significant privacy threat, we can incentivize services to streamline support for ubiquitous encryption. Furthermore, we must alert users of the privacy risks they face when connecting to public wireless networks or browsing through Tor, and educate them on the extent of protection offered by existing mechanisms.

3.11 Conclusion

In this chapter, we presented our extensive in-depth study on the privacy threats that users face when attackers steal their HTTP cookies. We audited a wide range of major services and found that cookie hijacking attacks are not limited to a specific type of websites, but pose a widespread threat to any website that does not enforce ubiquitous encryption. Our study revealed numerous instances of major services exposing private information and pro-

tected account functionality to non-authenticated cookies. This threat is not restricted to websites, as users' cookies are also exposed by official browser extensions, search bars, and mobile apps. To obtain a better understanding of the risk posed by passive eavesdroppers in practice, we conducted an IRB-approved measurement study and detected that a large portion of the outgoing traffic in public wireless networks remains unencrypted, thus, exposing a significant amount of users to cookie hijacking attacks. We also evaluated the protection offered by popular browser-supported security mechanisms and found that they can reduce the attack surface but cannot protect users if websites do not *support* ubiquitous encryption. The practicality and pervasiveness of these attacks, also renders them a significant threat to Tor users, as they can be deanonymized by adversaries monitoring the outgoing traffic of exit nodes.

In the next chapter, we analyze and evaluate the current HTTPS enforcement mechanisms in term of practicality in deployment and effectiveness for protecting users' information leakage via unencrypted connection as explained and evaluated in this chapter.

Chapter 4

Evaluating HTTPS Enforcing Mechanisms

4.1 Overview

In Chapter 3, we studied significant threats when web services fail to enforce ubiquitous encryption, explored this phenomenon in the majority of top services and demonstrated how users are exposed to cookie hijacking attacks with severe privacy implications: vulnerable to surveillance, information leakage through non-secure cookies, as well as exposed account functionality and potential account takeover. As shown, migrating to HTTPS is a daunting task with multifaceted challenges, which has resulted in a tangled web of partial support of encryption across websites and flawed access control.

To eliminate this problem, as mentioned, many security mechanisms have been proposed [16, 72, 109] 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 [16]. While HSTS is gaining traction, this technology is still in a relatively early stage of adoption, with a recent study also showing that many sites deploy the protocol incorrectly [117]. 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 [72], 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.

The main goals of this chapter are to *(i) explore available security mechanisms and defenses that are already deployed by web services or can be deployed by end users with and without requiring server modification, (ii) evaluate their effectiveness in enforcing HTTPS and preventing cookie hijacking attacks in practice.*

Specifically, our study focuses on HSTS and HTTPS Everywhere as they are the most 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. We design our testing framework Section 4.5 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 traffics observed (traffics collected from Section 3.6 and connecting to Alexa top million domains), and conduct an in-depth analysis of unencrypted connections towards domains that have adopted mechanisms for enforcing HTTPS.

4.2 HTTPS Enforcing Mechanisms

We begin by exploring and categorizing current encryption enforcing mechanisms. Figure 4.1 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.

4.3 Server-side Mechanisms

Here we provide an overview of the mechanisms at the disposal of servers.

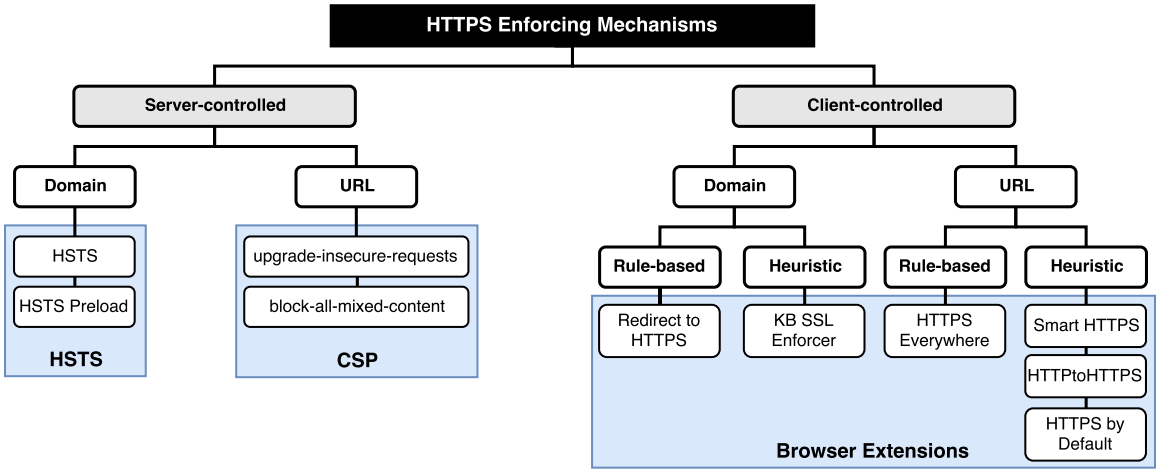


Figure 4.1: Taxonomy of HTTPS enforcing mechanism (including HTTP blocking mechanisms).

4.3.1 HSTS

While technically HSTS and HSTS preload lie on the client-side and is enforced by the browser, the server has to fulfill a set of requirements (e.g., include HSTS header) and apply for inclusion within the list for HSTS preload. Thus, we categorize this mechanism as a server-controlled solution. Here we describe additional HSTS preload detail on Chrome and Firefox.

Chrome Preload. The Chromium project is in charge of maintaining the preload list which is shipped with the Chrome browser [50]. 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.

To be added and remain on the HSTS preload list websites must satisfy a set of requirements set by the Chromium project [103], 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 fewer 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 a 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 [138].

4.3.2 Content Security Policy

The Content Security Policy (CSP) [183] 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 (XSS) 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 [184]. 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 web page, and not used to create a policy for the entire domain.

Though CSP is not mainly designed for enforcing HTTPS, two directives are recently

Table 4.1: Overview of available client-side solutions.

Extension	Browser Support	#Users*	Last Update
HTTPS Everywhere	Firefox, Chrome, Opera, Tor, Firefox for Android	3.3M	01/2018
Redirect to HTTPS	Opera	123.5K	03/2011
KB SSL Enforcer	Chrome	35.7K	11/2016
Smart HTTPS	Firefox, Chrome, Opera	14.6K	09/2017
HTTPtoHTTPS	Firefox	1.7K	01/2018
HTTPS by Default	Firefox	1.5K	01/2017

*Total downloads/users across all supported browsers.

proposed to reduce or block HTTP connections including `upgrade-insecure-requests` and `block-all-mixed-content` [187].

Upgrade Insecure Requests The CSP header in an HTTP response can contain the `upgrade-insecure-requests` [186] directive to instruct the browser to "upgrade" all HTTP requests to HTTPS before the fetching request is transmitted. This can, therefore, mitigate threats by preventing insecure requests from being transmitted over the network. However, as web pages 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.

Blocking Mixed Content. As mentioned in Section 3.3, mixed active content is currently blocked in major browsers by default, while mixed passive content is allowed but accompanied by visual warnings [181]. Although with the inclusion of insecure references to non-active (display) resources (such as images, audio, video), adversaries are not able to modify critical functionality, the user's HTTP cookies can be exposed to hijacking attacks.

4.4 Client-side 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 4.1 provides general some information for these extensions, including browser support and their number of downloads.

4.4.1 HTTPS Everywhere

HTTPS Everywhere is a browser extension that was developed by the Tor Project and the Electronic Frontier Foundation [72]. The extension operated through rulesets that contain a collection of rules for each domain, which 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.

4.4.1.1 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 4.1, along with the relevant attributes.

```
<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>
```

Listing 4.1: Example of HTTPS Everywhere ruleset structure

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 (`<exclusion pattern="" />`) 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. It is 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 [74]. 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.

4.4.1.2 Adding Rulesets

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

4.4.1.3 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.

4.4.1.4 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.

4.4.1.5 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.

4.4.2 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. As two of the mechanisms are severely outdated and don't support recent browser versions, we omit them from our evaluation. We present these extensions and our analysis including their ineffectiveness in Appendix A.2.

4.5 Measurement 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*.

4.5.1 Server-side Mechanism Testing

The *online* module focuses on testing mechanisms that lie on the server-side. We use `curl`[1] 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

[1] `https://curl.haxx.se/`

the HTTP response headers and body (HTML tag and content) that is relevant to the mechanism we are studying in each experiment.

4.5.1.1 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.

4.5.1.2 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.

4.5.2 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.

4.5.2.1 HSTS Preload Module

This module is designed to check if a URL's domain 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 4.6.1).

4.5.2.2 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 4.6.1. 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 [142], giving us the ability to execute the JavaScript without the need for a browser, rendering our experimentation lightweight and efficient.

4.6 Evaluation

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

4.6.1 Data Collection and Statistics

Datasets. The main goal of our evaluation on each HTTPS enforcing mechanisms is to answer the following questions: *(i) how much the mechanism is deployed in the real-world?*, *(ii) how effective the mechanism is in reducing HTTP requests on HTTPS web pages?*, and *(iii) does the mechanism have different coverage across browsers?*, To this end, we utilized

Table 4.2: Unique domains and URLs observed over HTTP (our dataset)

	Records	%
URLs	599,034,558	100.00
Base domains	699,873	100.00
Base domains support TLS	409,026	58.44

two data sources including URLs that we extracted from HTTP requests we collected from our university wireless network experiment (Section 3.6) and domains from the Alexa top 1 million sites (accessed on May 2016).

As described in Section 4.2, existing mechanisms operate on both URL-level (page-level) and domain-level. Both sources, therefore, allow us to perform evaluations on both types of mechanisms. Additionally, while the HTTP traffic data source allows us to measure the current deployment and effectiveness in reducing HTTP connection in practice, the top million domains allows us to measure these on the top-sites domains (and each site's landing page). Although similar to our experiment, Kranch et al. [117] studied and evaluated HSTS using the domains from the top 1 million Alexa sites, we, aim to evaluate this dataset with other mechanisms (e.g., CSP, HTTPS Everywhere) missing from their works. We refer to their study and compare their result with ours.

Terminology. We use the term *"domain"* to refer to fully-qualified domain name e.g., `www.example.com`, `www.example.co.uk`, and *"base domains"* to refer to the highest-level non-public domain e.g., `example.com`, `example.co.uk`. To handle TLDs with two labels (e.g., `co.uk`) and to classify them correctly as TLD, we cross-check with Mozilla's public suffix list [136].

Finally, our HTTP request dataset contains approximately 1.4 billion, with over 500 million requests (36.48%) containing at least one HTTP cookie. Table 4.2 shows the number of unique URLs and base domains of this dataset.

Table 4.3: Base domains and HSTS support.

	Unique Base Domains	%
Support HTTPS	409,026	100.00
Support HSTS	9,297	2.27
HSTS + `includeSubdomains`	1,418	0.35
HSTS + `preload`	921	0.23

Table 4.4: 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

4.6.2 Analysis for HSTS

HSTS Header. Using the URLs extracted from our dataset, we study the coverage and effectiveness of HSTS in practice, as detailed in Section 4.5. Specifically, in Table 4.3, we show how many of the unique base domains from our dataset are connectable over HTTPS. Out of those, only 9,297 (2.27%) 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.

HSTS Preload. We use the preload list released May 2016, which contains a list of 12,602 domains (12,233 base domains). Table 4.4 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 cookie hijacking attacks. This is mainly to due to out-of-date browsers. In Table 4.5 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 (4.31%) 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.

Table 4.5: 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 4.6: HSTS preload domains set to `Opportunistic`.

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

HSTS Adoption: *From all HTTPS-supported base domains in our dataset, only 2.27% of domains support HSTS and 0.23% of domains support HSTS preload. Kranch and Bonneau [117] also presented a similar result, where their evaluation revealed HSTS low adoption rate on the top Alexa 10k domains.*

Opportunistic Security. There are 259 (2.06%) domains on HSTS preload that are `opportunistic` (Table 4.6), i.e., do not set `force-https` in the Chrome preload list. The vast majority of those domains belong to Google (250 domains), with 218 of those covering google.com and Google's regional search engines. As demonstrated in the previous chapter, 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.

Partial Security. In the latest release of the preload list that we evaluate, we found that 156 (1.24%) domains do not set `include_subdomains`, with 89 not specifying any directive

Table 4.7: 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

for the subdomains while 97 explicitly set it to `false`. Surprisingly, 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 [117, 162].

HSTS Effectiveness: *From our experiment, we still observed HTTP requests from HSTS deployed domains. While HSTS preload is effective when deployed correctly, as shown only 0.05% of HTTP requests toward domains that are covered by the HSTS preload list (escape preload list), mainly due to out-of-date browsers. However, some (sub)domains in preload list still allow HTTP requests due to setting in opportunistic mode or not include subdomain directive.*

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 4.7, while Chrome's list prevents almost 15 million requests from being issued over an unencrypted connection higher than Safari's. 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.

HSTS Coverage Across Browsers: *Safari HSTS preload list contains fewer entries than Chrome and Firefox (only 3.52% of the Chrome's preload entries). From our dataset, Chrome's list prevents almost 15 million requests from being issued over an unencrypted connection higher than Safari's.*

4.6.3 Analysis for CSP

Below we discuss our findings regarding the use of CSP for upgrading connections to HTTPS or blocking unencrypted connections. Table 4.8 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.

Our 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 4.8, 27,565 (∼0.03%) of the URLs upgrade the insecure requests, while only 557 blocked all mixed content.

Top-site Dataset Analysis. We also study the use of CSP in the top-site dataset to obtain a better 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 pages. 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 4.8: 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 Top 1M Domains	%
upgrade-insecure-requests	HTTP header	27,565	0.03	250	0.03
	HTML meta tag	259	0.00	40	0.00
	HTTP header/HTML meta tag	27,824	0.03	290	0.03
block-all-mixed-content	HTTP header	557	0.00	36	0.00
	HTML meta tag	0	0.00	0	0.00
	HTTP header/HTML meta tag	557	0.00	36	0.00

Table 4.9: Support of CSP directives in current version of major browsers.

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

CSP Adoption: *CSP upgrade-insecure-requests and block-all-mixed-content adoption rates are close to 0% on URLs of our dataset and landing page of top-site dataset.*

Coverage Across Browsers. Our experiments reveal that current versions of Chrome, Firefox, Opera and Safari do not support `block-all-mixed-content` CSP mechanism (Table 4.9), while `upgrade-insecure-requests` is currently only supported in Chrome, Firefox and Opera.

CSP Coverage Across Browsers: *All major browsers have not supported block-all-mixed-content. While other major browsers have supported upgrade-insecure-requests, Safari has not.*

Table 4.10: 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

4.6.4 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 3.3 million installations.

4.6.4.1 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 4.10.

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 4.4.1.3) 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 con-

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

tained 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 general case, any unencrypted static content in an encrypted page will be blocked. This is done in most major browsers (e.g. Chrome, Firefox, Opera). However, 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 [73, 179].

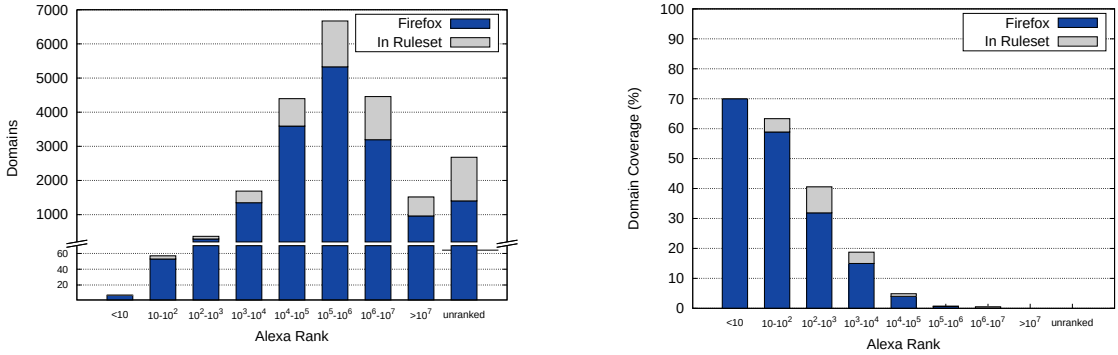
CACert. CACert is a community-driven approach towards the creation of a certificate authority [39]. However, the root certificate is not included in many popular browsers (Firefox, Chrome, Opera, and Tor Browser Bundle). When connecting over HTTPS to one of the sites that use a CACert issued certificate, 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%).

4.6.4.2 Adoption and Coverage

HTTPS Everywhere extension gains over 3.3 million downloads across all major browsers (Table 4.1). The extension is also by-default included in Tor browser [152]. Although the extension gains popularity, we also want to understand how effective of them in practice.



(a) Number of domains

(b) Domain coverage

Figure 4.2: Number of domains and coverages in each ranking tier, for domains found in all rulesets, and domains in rulesets that are active in Firefox.

> ***HTTPS Everywhere Adoption:*** *HTTPS Everywhere gains over 3.3 million downloads and is the most adopted client-side HTTPS enforcing mechanism. However, this adoption rate is still very low when compared to all internet users.*

Site Ranking. In Figure 4.2(a) 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 top-site dataset and found that 13,812 base-domains covered by HTTPS Everywhere ranked in the top 1 million sites. Figure 4.2(b) 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 (751 domains) and still higher HSTS (12,593 domains), which has been reported previously [117]. 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

Table 4.11: 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 + HSTS Preload	624	2.86
Total	**15,525**	**71.09**

Table 4.12: Handling of HTTP requests when HTTPS Everywhere is installed.

URL	Requests (million)	%	Requests (million) with cookie	%
Modified to HTTPS	376.6	26.95	145.2	10.39
Remains on HTTP	1,020.9	73.05	364.7	26.10

the remaining domains in Table 4.11. Out of those domains that support HTTPS, we found that only 2,481 (11.36%) have adopted HSTS.

Quantifying Impact. We tested all the URLs contained in our HTTP request dataset, to see how many URLs would be protected if every user had installed the HTTPS Everywhere extension. As can be seen Table 4.12, 27.96% 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.

Table 4.13 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

Table 4.13: Cause for unmodified HTTP requests.

Reason	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
Domain in rulesets	171,718,126	16.82
Domain not in rulesets	849,218,603	83.18

target hosts. Those 849.2 million requests contain 1.39 million unique domains (707,188 unique 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 domains are connectable over HTTPS, and found that 61.01% of the 1.39 million unique domains are indeed connectable (341,520 unique base domains). Although adding these domains in the rulesets helps to reduce HTTP connections, we also have to consider (i) the feasibility of adding all these domains in the rulesets and (ii) does the lack of coverage actually matter in practice.

To estimate the feasibility of covering these domains, 334,039 based domains need to be added to HTTPS Everywhere rulesets (approximately 15x multiplier of current coverage domains). This might affect the performance of HTTPS Everywhere. To see if the lack of coverage matters, we cross-check these missing domains with the Alexa top million sites and found that only 88,252 domains (26.42%) in the top million sites. To this end, while these domains can be added to the rulesets for increasing coverage starting from top million sites, it requires a lot of effort to maintain 88k rulesets manually.

No rules match represents the cases where the URL targets supported domains 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.

Table 4.14: Accounts from our public wireless trace (Section 3.5.1) that remain exposed even with HTTPS Everywhere installed.

Service	Exposed Accounts	Reduction
Google	31,729	53.12%
Yahoo	5,320	43.55%
Baidu	4,858	4.63%
Bing	378	38.03%
Amazon	22,040	5.68%
Ebay	1,685	0%
Target	46	0%
Walmart	97	23.62%
NYTimes	15,190	0%
Guardian	343	0.29%
Huffington	42	0%
MSN	927	39.25%
Doubleclick	124,352	0%
Youtube	264	99.21%
Total	207,271	26.62%

HTTPS Everywhere Coverage: *Over the top 1 million sites, the ruleset of HTTPS Everywhere covers a higher number of domains in HSTS and HSTS preload. Our experiment also revealed that HTTPS Everywhere is able to reduce the number of HTTP requests in our dataset 10.4% if the extension is installed in all browsers.*

4.6.4.3 Effectiveness

Cookie Hijacking. To simulate the potential impact of HTTPS Everywhere in reducing cookie leakage, we use the network trace collected from our campus' public WiFi and calculate the number of accounts that would remain exposed due to URLs not handled by HTTPS Everywhere rulesets. Due to those 73% connections remaining on HTTP (Table 4.12), 207,271 accounts remain exposed to our cookie hijacking attacks. Table 4.14 breaks down the numbers per targeted service. The largest impact is seen on Youtube where less than 1% of the users remain exposed while Ebay, Doubleclick and numerous

news sites are not impacted at all. Surprisingly, even though Google's main page is protected, over 46% of the users remain exposed when visiting a Google service. For the remaining search engines, the impact has a varying degree, with over 95% of the Baidu users remaining susceptible to cookie hijacking.

While the HTTPS Everywhere offers reducing HTTP connections, its effectiveness in mitigating cookie hijacking attacks varies greatly depending on each website's implementation. Even with all protection mechanisms enabled, users still face the risk of deanonymization when visiting popular sites.

4.6.4.4 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 4.2 demonstrates an example ruleset that works correctly regardless of the user adding a trailing slash.

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

Listing 4.2: Rule expecting trailing slash on top level.

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 sub-level 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 inconsistently handling of the same URL depending on the presence of a trailing slash. For example, in the ruleset in Listing 4.3, `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.

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

Listing 4.3: Mishandling due to lack of trailing slash.

Missing Target Hosts. As can be seen in Listing 4.4, `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.

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

Listing 4.4: Example of missing target host in ruleset.

To identify how many rulesets are affected 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 4.4.1.3) does not capture this error.

Rule Coverage. As shown in Table 4.13 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.

HTTPS Everywhere Effectiveness: *HTTPS Everywhere reduced 26.62% from the original number of exposed accounts. However, as the extension highly depends on their community-written rulesets, any incorrect or missing rulesets impact their domain and URL coverage and consequently affect their overall effectiveness.*

4.7 Current Deployment States (Updated Results)

Our evaluations were carried out during November 2015 - June 2016. To understand the changes in the current states of deploying HTTPS enforcing mechanisms, we repeat some of the experiments, including evaluating HSTS and CSP adoption, and HTTPS Everywhere coverage using our frameworks towards the Alexa top million sites (retrieved on February 2018). We highlight and describe the changes as follows.

HSTS and HSTS Preload. Over the period of 2-3 years (from our experiments in 2016 Section 4.6.2 and the survey from Kranch and Bonneau in 2015 [117]), we observe higher HSTS and HSTS preload adoptions. Table 4.15 shows the breakdown of the number of HSTS and HSTS preload adopted domains from our latest evaluation. As shown in the table, now over 11.29% of Alexa top million sites adopted HSTS (approx. 9.0x multiplier from the 2015 study). Similar to HSTS, the number of domains in the preload increases to 44,907 (approx. 3.5x multiplier of our previous measurement). However, only 27.45% of those domains are `includeSubdomains`, this setup could potentially leak users' cookies in their subdomains as pointed out in Chapter 2.4.1 and Chapter 3.3. Additionally, there are a number of HSTS domains that are still set `max-age` to be 0 and extremely low `max-age`

(less than a day). Setting the `max-age=0` instructs the browsers to remove from their known HSTS hosts, including includeSubdomains directive [16], therefore there will be no HTTPS enforced in these domains. Although this setting seems to be a mistake from the admin, it can be intended for being in the *knockout entry*, where the admin wants to remove their domains from HSTS preload [103].

Opportunistic. We still observe a number of domains in preload that are set to be in opportunistic (ForceHTTPS) nearly to our experiment in 2016. The majority of domains still belongs to Google and its regional domains and vulnerable to HTTP cookie hijack attacks.

Table 4.15: HSTS domains in Alexa Top 1M and the preload list (updated).

	Alexa Top 1M		**Preload Domains**	
	Domains	**%**	**Domains**	**%**
Attempt to set HSTS	112,906	--	44,907	--
ForceHTTPS	--	--	44,645	99.42
ForceHTTPS + includeSubdomains	30,991	27.45	44,400	98.87
max-age=0	16,226	14.37	--	--
0 < max-age ≤ 1 day	4,764	4.22	--	--

CSP. The upgrade-insecure-requests and block-all-mixed-content are now supported in all major browsers (Table 4.16). To see if affects the current adoption, we repeat our experiment on both CSPs on the landing page of domains in Alexa top million sites. Table 4.17 presents the breakdown of both CSPs adoption. To this end, only 16,208 (1.68%) landing pages on top million sites set either of these CSP mechanisms.

Table 4.16: Support of CSP directives in current version of major browsers (updated).

Browser	**upgrade-insecure-requests**	**block-all-mixed-content**
Chrome 64.0	✓	✓
Firefox 58.0	✓	✓
Safari	✓	✓
Opera	✓	✓

Table 4.17: Use of CSP directive for upgrading to HTTPS and blocking mixed content in top 1M Alexa sites (updated).

Content Security Policy	Setting	Alexa Top 1M	%
upgrade-insecure-requests	HTTP header	14,517	1.45
	HTML meta tag	676	0.07
	HTTP header/HTML meta tag	15,170	1.52
block-all-mixed-content	HTTP header	1,206	0.12
	HTML meta tag	147	0.01
	HTTP header/HTML meta tag	1,340	0.13

HTTPS Everywhere. At the time of writing, the current release of HTTPS Everywhere rulesets (January 1, 2018) contains 23,574 ruleset files (19.02% increasing from 2016 presented in Section 4.6.4). Interestingly the number of target hosts increases over 2.5x multiplier (142,330 hosts) from our study in 2016. This is due to adding missing target host which was also identified by our analysis (Section 4.6.4.4) [106]. Finally, we found that 39,636 domains in Alexa top million sites covered by HTTPS Everywhere.

Site Ranking. Figure 4.3 illustrates the adoption of our studied HTTP enforcing mechanisms. The current adoptions of these popular domains increase to 90%, 40%, 70% on the top 10 domains for HSTS, HSTS preload, and HTTPS Everywhere, respectively. The only site in top 10 domains that does not employ HSTS is `qq.com`. While `baidu.com` (also in top 10) deploys HSTS but does not redirect users to HTTPS by default. In top 100, 68 domains adopted HSTS. Although we observe the larger number of sites adopts HSTS preload, only 12 domains from top 100 domains are in the preload list. HTTPS Everywhere still has a larger domain coverage HSTS and HSTS preload (71 domains), but the rulesets that operate on page-level could potentially expose user information on some uncovered pages.

Both upgrade-insecure-requests and block-all-mixed-content show the lowest support on Alexa top million sites. None of landing pages in top 10 domains deploy these CSP mechanisms. However as mentioned, over 90% already support HSTS. However as these

mechanisms operate on page-level, we likely to observe higher adoption rates using URL-based datasets.

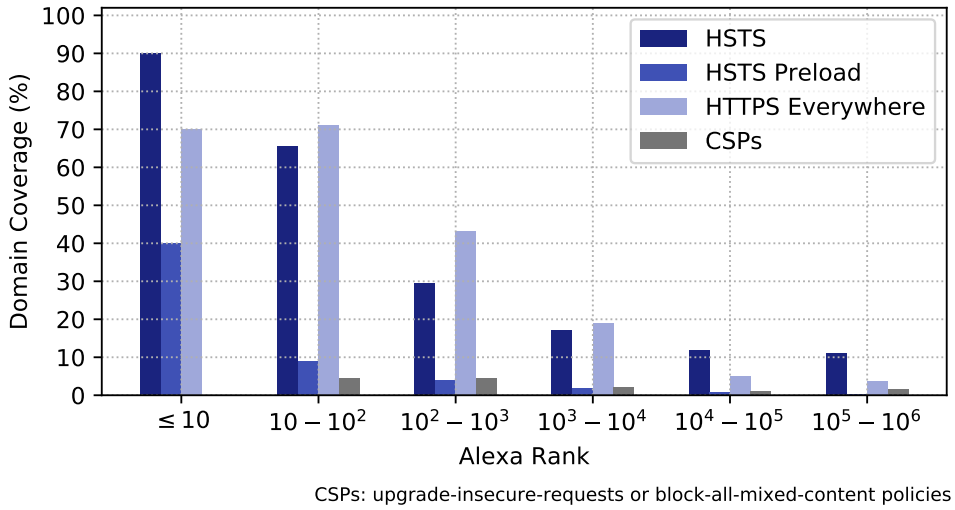


Figure 4.3: Histogram of Alexa Top 1 million domains for HSTS, HSTS preload, HTTPS Everywhere and CSP (upgrade-insecure requests or block-all-mixed-content policies).

4.8 Conclusion

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 [117]. 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 the previous chapter). As such, these findings highlight the necessity to streamline the deployment of ubiquitous encryption.

Our experiments showed 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 studied and should be employed for reducing insecure requests within specific usage scenarios.

While HTTPS Everywhere is the most effective client-side mechanism that we have found available, our tests against the network dataset indicated 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 there is significant room for improvement when it comes to the automated evaluation of rulesets prior to their incorporation to the extension.

Our updated result on Alexa top million sites shows a higher HSTS and CSP adoption rate and domain coverage of HTTPS Everywhere. Specifically approximately 11% of the top million attempts to HSTS. The number of target hosts in HTTPS Everywhere expands to over 2.5x multiplier from our previous survey, whereas the CSP adoption is still very low. However, these number of adoptions are still far from ubiquitous encryption.

In conclusion, all of 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.

Chapter 5

Hostname Verification in TLS Implementations

5.1 Overview

Previous chapters explore the importance of complete deployment of HTTPS and evaluate the effectiveness of existing HTTPS enforcing mechanisms. As a step towards enhancing the security of web encryption, we are now studying TLS, the family protocols for securing network communications, which is a sub-layer under regular HTTPS. The security guarantees of web encryption are thus critically dependent on the TLS protocols and consequently the correctness of TLS implementations.

Similar to HTTPS, TLS protocols comprise multiple RFCs and integrate multiple components. We opt to study the *TLS certificate validation* implementations, particularly the *hostname verification* process, as HTTPS and TLS protocols are critically dependent on correct validation of X.509 digital certificates presented by the servers during the TLS handshake phase.

The Importance of TLS Hostname Verification. When a client communicates using TLS, it references some notion of the server's identity (e.g., "the website at `example.com`") while attempting to establish secure communication. The certificate validation depends on hostname verification for verifying the identity of TLS entities (including the web server).

Specifically, hostname verification verifies that the hostname (i.e., fully qualified domain name, IP address, and so forth) of the server matches one of the identifiers in the "SubjectAltName" extension or the "Common Name" (CN) attribute of the presented leaf certificate. Therefore, any mistake in the implementation of hostname verification could completely undermine the security and privacy guarantees of TLS as well as HTTPS which is built around it.

Hostname verification is a complex process due to the presence of numerous special cases (e.g., wildcards, IP addresses, international domain names, etc.). For example, a wildcard character (`'*'`) is only allowed in the left-most part (separated by `'.'`) of a hostname. To get a sense of the complexities involved in the hostname verification process, consider the fact that different parts of its specifications are described in five different RFCs [6, 8, 11, 14, 17]. Given the complexity and security-critical nature of the hostname verification process, it is crucial to perform automated analysis of the implementations for finding any deviation from the specification.

However, despite the critical nature of the hostname verification process, none of the prior research projects dealing with adversarial testing of TLS certificate validation [37, 48, 79, 85], support detailed automated testing of hostname verification implementations. The prior projects either completely ignore testing of the hostname verification process or simply check whether the hostname verification process is enabled or not. Therefore, they cannot detect any subtle bugs where the hostname verification implementations are enabled but deviate subtly from the specifications. The key problem behind automated adversarial testing of hostname verification implementations is that the inputs (i.e., hostnames and certificate identifiers like common names) are highly structured, sparse strings and therefore makes it very hard for existing black/gray-box fuzz testing techniques to achieve high test coverage or generate inputs triggering the corner cases. Heavily language/platform-dependent white-box testing techniques are also hard to apply for testing hostname verification implementations due to the language/platform diversity of TLS implementations.

In this chapter, we present HVLearn, a novel black-box testing framework for analyzing TLS hostname verification implementations, which is based on automata learning algorithms. HVLearn utilizes a number of certificate templates, i.e., certificates with a common

name set to a specific pattern, in order to test different rules from the corresponding specification. For each certificate template, HVLearn uses automata learning algorithms to infer a Deterministic Finite Automaton (DFA) that describes the set of all hostnames that match the CN of a given certificate. Once a model is inferred for a certificate template, HVLearn checks the model for bugs by finding discrepancies with the inferred models from other implementations or by checking against regular-expression-based rules derived from the specification. The key insight behind our approach is that the acceptable hostnames for a given certificate template form a regular language. Therefore, we can leverage automata learning techniques to efficiently infer DFA models that accept the corresponding regular language.

We use HVLearn to analyze the hostname verification implementations in a number of popular TLS libraries and applications written in a diverse set of languages like C, Python, and Java. We demonstrate that HVLearn can achieve on average 11.21% higher code coverage than existing black/gray-box fuzzing techniques. By comparing the DFA models inferred by HVLearn, we found 8 unique violations of the RFC specifications in the tested hostname verification implementations. Several of these violations are critical and can render the affected implementations vulnerable to active man-in-the-middle attacks.

5.2 Summary of Hostname Verification in RFCs

As part of the hostname verification process, the TLS client must check that the hostname of the server matches either the "common name" attribute in the certificate or one of the names in the "subjectAltName" extension in the certificate [11]. Note that even though the process is called hostname verification, it also supports verification of IP addresses or email addresses.

In this section, we first provide a brief summary of the hostname format and specifications that describe the format of the common name attribute and subjectAltName extension formats in X.509 certificate. Figure 5.1 provides a high-level summary of the relevant parts of an X.509 certificate. Next, we describe different parts of the hostname verification process (e.g., domain name restrictions, wildcard characters, and so forth) in detail.

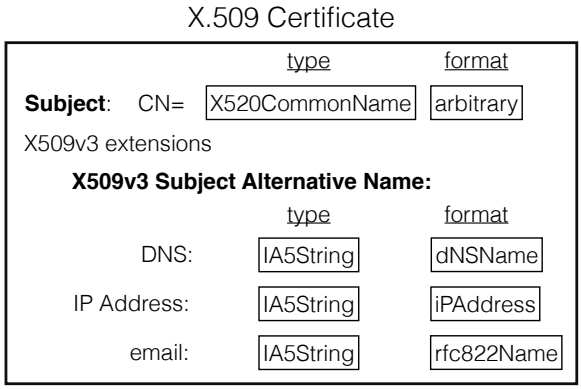


Figure 5.1: Fields in an X.509 certificate that are used for hostname verification.

5.2.1 Hostname Verification Inputs

Hostname Format. Hostnames are usually either a fully qualified domain name or a single string without any `'.'` characters. Several TLS implementations (i.e., OpenSSL) also support IP addresses and email addresses to be passed as the hostname to the corresponding hostname verification implementation.

A domain name consists of multiple "labels", each separated by a `'.'` character. The domain name labels can only contain letters `a-z` or `A-Z` (in a case-insensitive manner), digits `0-9` and the hyphen character `'-'` [2]. Each label can be up to 63 characters long. The total length of a domain name can be up to 255 characters. Earlier specifications required that the labels must begin with letters [11]. However, subsequent revisions have allowed labels that begin with digits [3].

Common Names in X.509 Certificates. The common name (CN) is an attribute of the "subject distinguished name" field in an X.509 certificate. The common name in a server certificate is used for validating the hostname of the server as part of the certificate verification process. A common name usually contains a fully qualified domain name, but it can also contain a string with arbitrary ASCII and UTF-8 characters describing a service (e.g., `'CN=Sample Service'`). The only restriction on the common name string is that it should follow the X520CommonName standard (e.g., should not repeat the substring `'CN='`) [11]. Note that this is different from the hostname specifications that are very strictly defined and only allow certain characters and digits as described above.

SubjectAltName in X.509 Certificates. Subject alternative name (subjectAltName) is an X.509 extension that can be used to store different types of identity information like fully qualified domain names, IP addresses, URI strings, email addresses, and so forth. Each of these types has different restrictions on allowed formats. For example, dNSName(DNS) and uniformResourceIdentifier(URI) must be valid IA5String strings, a subset of ASCII strings [11]. We refer interested readers to Section 4.1.2.6 of RFC 5280 for further reading.

5.2.2 Hostname Verification Rules

Matching Order. RFC 6125 recommends TLS implementations to use subjectAltName extensions, if present in a certificate, over common names as the common name is not strongly tied to an identity and can be an arbitrary string as mentioned earlier [14]. If multiple identifiers are present in a subjectAltName, the TLS implementations should try to match DNS, SRV, URI, or any other identifier type supported by the implementation and must not match the hostname against the common name of the certificate [14]. The Certificate Authorities (CAs) are also supposed to use the dNSName instead of common name for storing the identity information while issuing certificates [6].

Wildcard in Common Name/SubjectAltName. If a server certificate contains a wildcard character `'*'`, a TLS implementation should match hostname against them using the rules described in RFC 6125 [14]. We provide a summary of the rules below.

A wildcard character is only allowed in the left-most label. If the presented identifier contains a wildcard character in any label other then the left-most label (e.g., `www.*.example.com` and `www.foo*.example.com`), the TLS implementations should reject the certificate. A wildcard character is allowed to be present anywhere in the left-most label, i.e., a wildcard does not have to be the only character in the left-most label. For example, identifiers like `bar*.example.com`, `*bar.example.com`, or `f*bar.example.com` valid.

While matching hostnames against the identifiers present in a certificate, a wildcard character in an identifier should only apply to one subdomain (one label) and a TLS implementation should not compare against anything but the left-most label of the hostname

(e.g., `*.example.com` should match `foo.example.com` but not `bar.foo.example.com` or `example.com`).

Several special cases involving the wildcards are allowed in the RFC 6125 only for backward compatibility of existing TLS implementations as they tend to differ from the specifications in these cases. RFC 6125 clearly notes that these cases often lead to overly complex hostname verification code and might lead to potentially exploitable vulnerabilities. Therefore, new TLS implementations are discouraged from supporting such cases. We summarize some of them: (i) a wildcard is all or part of a label that identifies a public suffix (e.g., `*.com` and `*.info`), (ii) multiple wildcards are present in a label (e.g., `f*b*r.example.com`), and (iii) wildcards are included as all or part of multiple labels (e.g., `*.*.example.com`).

International Domain Name (IDN). IDNs can contain characters from a language-specific alphabet like Arabic or Chinese. An IDN is encoded as a string of unicode characters. A domain name label is categorized as a U-label if it contains at least one non-ASCII character (e.g., UTF-8). RFC 6125 specifies that any U-labels in IDNs must be converted to A-labels domain before performing hostname verification [14]. U-label strings are converted to A-labels, an ASCII-compatible encoding, by adding the prefix `'xn--'` and appending the output of a Punycode transformation applied to the corresponding U-label string as described in RFC 3492 [7]. Both U-labels and A-labels still must satisfy the standard length bound on the domain names (i.e., up to 255 bytes).

IDN in SubjectAltName. As indicated in RFC 5280, any IDN in X.509 subjectAltName extension must be defined as type IA5String which is limited only to a subset of ASCII characters [11]. Any U-label in an IDN must be converted to A-label before adding it to the subjectAltName. Email addresses involving IDNs must also be converted to A-labels before.

IDNs in Common Name. Unlike IDNs in subjectAltName, IDNs in common names are allowed to contain a PrintableString (`A-Z`, `a-z`, `0-9`, special characters `' = ( ) + , - . / : ?`, and space) as well as UTF-8 characters [11].

Wildcard and IDN. There is no specification defining how a wildcard character may be embedded within A-labels or U-labels of an IDN [13]. As a result, RFC 6125 [14] recommends that TLS implementations should not match a presented identifier in a certificate where the wildcard is embedded within an A-label or U-label of an IDN (e.g., `xn--kcry6tjko*.example.com`). However, TLS implementations should match a wildcard character in an IDN as long as the wildcard character occupies the entire left-most label of the IDN (e.g., `*.xn--kcry6tjko.example.com`).

IP Address. IP addresses can be part of either the common name attribute or the subjectAltName extension (with an 'IP:' prefix) in a certificate. Section 3.1.3.2 of RFC 6125 specifies that an IP address must be converted to network byte order octet string before performing certificate verification [14]. TLS implementations should compare this octet string with the common name or subjectAltName identifiers. The length of the octet string must be 4 bytes and 18 bytes for IPv4 and IPv6 respectively. The hostname verification should succeed only if both octet strings are identical. Therefore, wildcard characters are not allowed in IP address identifiers, and the TLS implementations should not attempt to match wildcards.

Email. Email can be embedded in common name as the emailAddress attribute in legacy TLS implementations. The attribute is not case sensitive. However, new implementations must add email addresses in rfc822Name format to subject alternative name extension instead of the common name attribute [11].

Internationalized Email. As similar to IDNs in subjectAltName extensions, an internationalized email must be converted into the ASCII representation before verification. RFC 5321 also specifies that network administrators must not define mailboxes (`local-part@host-part`/address-literal) with non-ASCII characters and ASCII control characters. Email addresses are considered to match if the *local-part and host-part are exact matches using a case-sensitive and case-insensitive ASCII comparison respectively* (e.g., an identifier, `MYEMAIL@example.com` does not match `myemail@example.com` but match

`MYEMAIL@EXAMPLE.COM`) [11]. Note that this specification contradicts that of the email addresses embedded in the common name that is supposed to be completely case-insensitive.

Email with IP Address in the Host Part. RFCs 5280 and 6125 do not specify any special treatment for IP address in the host part of emails and only allow email in rfc822Name format. The rfc822Name format supports both IPv4 and IPv6 addresses in the host part. Therefore, an email with an IP address in the host part is allowed to be present in a certificate [12].

Wildcard in Email. There is no specification that wildcard should be interpreted and attempted to match when they are part of an email address in a certificate.

Other Identifiers in SubjectAltName. There are other identifiers that can be used to perform identity checks e.g., UniformResourceIdentifier(URI), SRVName, and otherName. However, most popular TLS libraries do not support checking these identifiers and leave it up to the applications.

5.3 Methodology

In this section, we describe the challenges behind automated testing of hostname verification implementations. Albeit small in size, the diversity of these implementations and the subtleties in the hostname verification process make these implementations difficult to test. We then proceed to describe an overview of our methodology for testing hostname verification implementations using automata learning algorithms. We also provide a brief summary of the basic setting under which automata learning algorithms operate.

5.3.1 Challenges in Hostname Verification Analysis

We believe that any methodology for automatically analyzing hostname verification functionality should address the following challenges:

- **Ill-defined Informal Specifications.** As discussed in Section 5.2, although the relevant RFCs provide some examples/rules defining the hostname verification process,

many corner cases are left unspecified. Therefore, it is necessary for any hostname verification implementation analysis to take into account the behaviors of other popular implementations to discover discrepancies that could lead to security/compatibility flaws.

- **Complexity of Name Checking Functionality.** Hostname verification is significantly more complex than a simple string comparison due to the presence of numerous corner cases and special characters. Therefore, any automated analysis must be able to explore these corner cases. We observe that the format of the certificate identifier as well as the matching rules closely resemble a regular expression matching problem. In fact, we find that the set of accepted hostnames for each given certificate identifier form a regular language.

- **Diversity of Implementations.** The importance and popularity of the TLS protocol resulted in a large number of different TLS implementations. Therefore, hostname verification logic is often implemented in a number of different programming languages such as C/C++, Java, Python, and so forth. Furthermore, some of these implementations might be only accessible remotely without any access to their source code. Therefore, we argue that a black-box analysis algorithm is the most suitable technique for testing a large variety of different hostname verification implementations.

5.3.2 HVLearn's Approach to Hostname Verification Analysis

Motivated by the challenges described above, we now present our methodology for analyzing hostname verification routines in TLS libraries and applications.

The main idea behind our HVLearn system is the following: For different rules in the RFCs as well as for ambiguous rules which are not well defined in the RFC, we generate "template certificates" with common names which are specifically designed in order to check a specific rule. Afterward, we use automata learning algorithms in order to extract a DFA which describes the set of all hostname strings which are matching the common name in our template certificate. For example, the inferred DFA from an implementation for the identifier template `"aaa.*.aaa.com"` can be used to test conformance with the rule in RFC

6125 prohibiting wildcard characters from appearing in any other label than the leftmost label of the common name.

Once a DFA model is generated by the learning algorithm, we check the model for violations of any RFC rules or for other suspicious behavior. HVLearn offers two methods to check an inferred DFA model:

Regular-expression-based Rules. The first option allows the user to provide a regular expression that specifies a set of invalid strings. HVLearn can ensure that the inferred DFAs do not accept any of those strings. For example, RFC 1035 states that only characters in the set [`A-Za-z0-9`] and the characters `'-'` and `'.'` should be used in hostname identifiers. Users, therefore, can construct a simple regular expression that can be used by HVLearn to check whether any of the tested implementations accept a hostname with a character outside the given set.

Differential Testing. The second option offered by HVLearn is to perform a differential testing between the inferred model and models inferred from other implementations for the same certificate template. Given two inferred DFA models, HVLearn generates a set of *unique differences* between the two models using an algorithm which we discuss in Section 5.4.5. This option is especially useful for finding bugs in corner cases which are not well defined in the RFCs.

We summarize the advantages of our approach below:

- Adopting a black-box learning approach ensures that our analysis method is language independent and we can easily test a variety of different implementations. Our only requirement is the ability to query the target library/application with a certificate and a hostname of our choice and find whether the hostname is matching the given identifier in the certificate.

- As pointed out in the previous section, hostname verification is similar to regular expression matching. Given that regular expressions can be represented as DFAs, adopting an automata-based learning algorithm for representing the inferred models for each certificate template is a natural and effective choice.

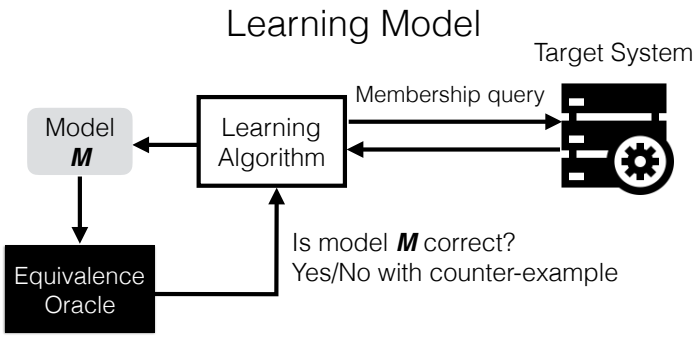


Figure 5.2: *Exact learning from queries:* the active learning model under which our automata learning algorithms operate.

- Finally, an additional advantage of having DFA models is that we can efficiently compare two inferred models and enumerate all differences between them. This property is very important for differential testing as it helps us in analyzing the ambiguous rules in the specifications.

Limitations. A natural trade-off of choosing to implement our system as a black-box analysis method is that we cannot guarantee completeness or soundness of our models. However, each difference inferred by HVLearn can be easily verified by querying the corresponding implementations. Moreover, since our system will find all differences among implementations, it will not report a bug that is common among all implementations unless a rule is explicitly specified for it, as described above. Finally, we point out that not all discrepancies among systems are necessarily security vulnerabilities; they may represent equally acceptable design choices for ambiguous parts of the RFCs.

5.3.3 Automata Learning Algorithms

We will now describe the automata learning algorithms that allow us to realize our automata-based analysis framework.

Learning Model. We utilize learning algorithms that work in an active learning model which is called *exact learning from queries*. Traditional supervised learning algorithms, such as those used to train deep neural networks, work on a given set of labeled examples. In

contrast, active learning algorithms in our model work by adaptively selecting inputs that they use to query a target system and obtain the correct label.

Figure 5.2 presents an overview of our learning model. A learning algorithm attempts to learn a model of a target system by querying the target system with inputs of its choice. Eventually, by querying the target system multiple times, the learning algorithm infers a model of the target system. This model is then checked for correctness through an *equivalence oracle*, an oracle that checks whether the inferred model correctly summarizes the behavior of the target system. If the model is correct, i.e., it agrees with the target system on all inputs, then the learning algorithm will output the generated model and terminate. On the other hand, if the model is incorrect, the equivalence oracle will produce a *counterexample*, i.e., an input under which the target system and the model produce different outputs. The learning algorithm then uses the counterexample to refine the inferred model. This process iterates until the learning algorithm produces a correct model.

To summarize, a learning algorithm in the exact learning model is able to interact with the target system using two types of queries:

- **Membership queries:** The input to this type of query is a string s and the output is `Accept` or `Reject` depending on whether the string s is accepted by the target system or not.

- **Equivalence queries:** The input to an equivalence query is a model M and the output of the query is either $True$, if the model M is equivalent to the target system on all inputs, or a counterexample input under which the model and target system produce different outputs.

Automata Learning in Practice. The first algorithm for inferring DFA models in the exact learning from queries model was developed by Angluin [26] and was followed by a large number of optimizations and variations in the following years. In our system, we use the Kearns-Vazirani (KV) algorithm [114]. The KV algorithm utilizes a data structure called the discrimination tree and it is in practice more efficient in terms of the amount of queries it requires to infer a DFA model.

The most significant challenge that one should address in order to use the KV algorithm

and other automata learning algorithms in practice, is how to implement an efficient and accurate equivalence oracle in order to simulate the equivalence queries performed by the learning algorithm. Since we only have black-box access to the target system, any method for implementing equivalence queries is necessarily incomplete.

In HVLearn, we use the Wp-method [82], for implementing equivalence queries. The Wp-method checks the equivalence between an inferred DFA and a target system using only black-box queries to the target system. Essentially, the Wp-method approximates an equivalence oracle by using multiple membership queries. The algorithm is given as input the DFA to be checked and an upper bound on the number of states in the target system when modeled as a DFA, a parameter which we call *depth*. Then, the algorithm creates a set of test inputs S, which are then submitted to the target system. If the target system agrees with the DFA model on all inputs in the test set S, then the DFA and the target system are proved equivalent under the assumption that the upper bound on the number of states of the target system is correct.

In theory, one can set the depth parameter of the Wp-method to a very large value in order to design an equivalence oracle which is, in practice, complete. However, the size of the set of test inputs produced by the Wp-method is on the order of $O(n^2|\Sigma|^{m-n+1})$ where Σ is the input alphabet for the DFA, m is the upper bound on the number of states of the target system and n is the number of states in the input DFA. Therefore, using the Wp-method with a large depth (i.e., upper bound on the number of states of the target system) is impractical. Note that, the bound on the number of test inputs produced by the Wp-method is not a worst case bound; on the contrary, the number of test inputs produced is usually of that order.

Consequently, it is essential for the efficiency of our system to maintain a small alphabet for our DFAs and also set a small upper bound (depth) on the number of states of the target system while using the Wp-method. We address both of these issues in the next section.

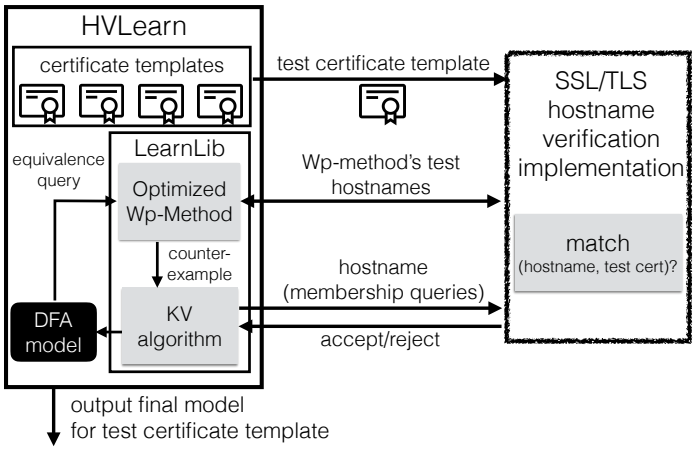


Figure 5.3: Overview of learning a hostname verification implementation using HVLearn.

5.4 Architecture of HVLearn

In this section, we describe the design and implementation of our system, HVLearn, based on automata learning techniques. Specifically, we describe the technical challenges that arise when we attempt to use automata learning algorithms in practice. We also summarize the optimizations that HVLearn implements to address these challenges and efficiently learn DFA models of hostname verification implementations.

5.4.1 System Overview

Figure 5.3 presents an overview of how HVLearn is used to analyze the hostname verification functionality of an TLS library. To use HVLearn, the user provides HVLearn access to the hostname verification function that takes an X.509 certificate and a hostname as input and returns `accept/reject` depending on whether the provided hostname is matching the identifier in the certificate. We describe how we implement this interface in Section 5.4.3. Our system includes a number of certificate templates, which are certificates designed to test the TLS implementation on a number of different rules as described in Section 5.4.2. For each such template, HVLearn will learn a DFA model describing the set of hostnames accepted by a given implementation for the given certificate template. To produce a DFA model, HVLearn utilizes the LearnLib [156] library which contains implementations of both the KV algorithm and the Wp-method. To avoid setting the maximum depth of the Wp-

method to impractically high values, we optimize the equivalence oracle as described in Section 5.4.4.

Once a model is generated, our system proceeds to analyze the model as described in Section 5.4.5. The results of our analysis, both the inferred models and the differences between models are then saved for reuse. Optionally, HVLearn can also utilize the inferred models for a certificate template to extract a formal specification for the corresponding certificate template as described in Section 5.4.6.

5.4.2 Generating Certificate Templates

To cover all different rules and ambiguous practices in hostname verification, we created a set of 23 certificates with different identifier templates, where each certificate is designed to test a specific rule from the specification. These certificates are selected to cover all the rules we described in Section 5.2. For example, a certificate with common name `"xn--a*.aaa"` will test if the implementation allows wildcards as part of an A-label in an IDN, something which is explicitly forbidden by RFC 6125. Our template certificates are self-signed X.509 v3 certificates generated using the GnuTLS library. We choose to use GnuTLS for certificate generation because it allows identifiers with embedded NULL characters in both subject common name and SAN. The template identifier to be tested is placed in either Subject CN and/or SAN (as dNSName, iPAddress, or email). We provide all of our generated certificate templates at `https://github.com/HVLearn/HVLearn/tree/master/CERT_TEMPLATES`.

5.4.3 Performing Membership Queries

In order to utilize the learning algorithms in LearnLib (including the Wp-method), we implement a *membership query* function that performs all queries to the target system. This function accepts input as a string and returns a binary value. In our system, we use the hostname verification function from the target TLS implementation. We note here that, since LearnLib is written in Java while many of our tested TLS implementations are written in C/C++/Python, we utilized the Java Native Interface (JNI) [147] to efficiently perform membership queries to the target in such cases.

5.4.4 Automata Learning Parameters and Optimizations

In this section, we describe the architectural decisions and optimizations that we implemented to efficiently scale the KV algorithm for testing complex real-world TLS hostname verification implementations.

Alphabet Size. The first important decision we have to make to utilize the KV algorithm is to select an alphabet that will be used by the algorithm. The alphabet refers to the set of symbols that the learning algorithm will test.

A straightforward approach is to use a very general set of characters such as the set of ASCII characters. However, this will impose an unnecessary overhead in our system's performance since the performance of both the KV algorithm and the Wp-method rely heavily on the underlying alphabet size. Our main insight is that we can reduce the alphabet to a small set of representative characters that will thoroughly test all different aspects of hostname verification. In particular we select the set $\Sigma = \{$ `a, 1, dot, \s, @, A, =, *, x, n, -, \u4F60, NULL`$\}$ as an input alphabet in our experiments. In the presented alphabet, `'dot'` denotes the `'.'` character, `\s` denotes the space character (ASCII value 32), NULL denotes the zero byte character, and `\u4F60` denotes the unicode character with hexadecimal value `4F60`.

Note that this set of symbols is adequate for analyzing hostname verification implementations since it includes characters from all different categories such as lowercase, uppercase, digits, unicode, etc., as well as special characters like the NULL character. The lowercase characters `'x'`, `'n'` in conjunction with the `'-'` character are necessary in order to encode IDN hostnames. Finally, the inclusion of some non-alphanumeric characters such as the `'='` character allows us to detect violations where an implementation accepts invalid hostnames.

Note that, even though the hostnames generated using this alphabet set will often not resolve to a real IP address when processed as DNS names, it does not affect the accuracy of our analysis in any way. This is a side-effect the fact that the hostname verification routines are not responsible for resolving the provided DNS name to an IP address. It simply checks whether the given hostname matches the identifier in the provided certificate.

Caching Membership Queries. To avoid the communication cost of repeated querying of the TLS implementations with same inputs, we utilize LearnLib's `DFALearningCache` class to cache the results of the membership queries. The cache is checked on each new query, and a cached result is used whenever found. This optimization is particularly useful for cutting down the overhead of the repeated queries generated by the Wp-method across multiple equivalence queries.

Optimizing Equivalence Queries. In practice, the first model generated by the learning algorithm is usually just single state DFA which rejects all hostnames. The reason is that the learning algorithm is not able to generate any accepting hostname and thus cannot distinguish between the initial state and any other state in the target system. Sometimes, to force the KV algorithm to produce an accepting hostname using the Wp-method, a very large depth is required. This may cause efficiency issues in the system. However, if we supply the model with an accepting hostname, then trivial models will be improved quickly without having to utilize excessive depth parameters in the Wp-method.

Recall here that the exponential term in the Wp-method is dependent on the difference between the number of states in the model and the provided depth. Therefore, once we discover an accepting state in the target system, the Wp-method with a much smaller depth will still be able to explore many different aspects of the hostname verification implementation.

In order to generate an accepting hostname, we perform the following test during an equivalence query and before calling the Wp-method. First, we search for any wildcard characters (*) in the provided common name and replace them with random characters from our alphabet to obtain a concrete hostname. Next, we check that the generated model and the target hostname verification implementation agree on a set of hostnames generated using this method. If not, we return the hostname for which they differ as a counterexample. The main advantage of this heuristic is that it allows us to quickly produce accepting hostnames that uncover new states in the target system without invoking the Wp-method with very large depth values. Once these states are uncovered, and the quality of the inferred models

improve, the Wp-method, with a small depth parameter, is utilized to discover additional states in the target system.

5.4.5 Analysis and Comparison of Inferred DFA Models

After HVLearn outputs a model, the next task for our system is to analyze the produced model for RFC violations or, confusing/ambiguous rules in the RFC, to compare different inferred models and analyze any discrepancies found between different implementations.

Analyzing a Single DFA Model. In the case of a single model, we would like to determine whether the model is accepting invalid hostnames prohibited by the RFC specification. If the specification is unclear, our analysis can still be used in order to manually inspect the behavior of the implementation on the specific certificate template besides the differential analysis described below.

Our system offers two options for performing analysis of a single model. First, our system generates inputs that will exercise all simple paths (i.e., paths without loops) that lead to accepting states, in the inferred model. Intuitively, these inputs are a small set of inputs that describe all different flavors of hostnames that will be accepted for the given certificate template. By inspecting these certificates, we can determine if the implementation is accepting invalid hostnames. Second, HVLearn allows the user to specify a regular expression rule to be checked against the inferred model. In this case, the user specifies a regular expression and HVLearn verifies that the regular expression and the inferred model does not share any common strings. This option allows to easily check certain RFC violations by utilizing simple regular expression rules. For example, consider the rule specifying that no non-alphanumeric characters should be part of a matching hostname. By specifying the regular expression rule `"(.)*=(.)*"` we can check whether there exists any matching hostname that contains the `'='` character in the inferred model.

Comparing Unique Differences between DFA Models. For analyzing certain corner cases which are not specified in the RFC, testing a single model may not be enough. Instead, we compare the inferred models for different TLS implementations and find inputs under which the implementations behave differently. To perform this analysis, we utilize the

difference enumeration algorithm from [29]. In a nutshell, this algorithm computes the product DFA between two, or more, given models and then finds all simple paths to states in which the DFAs are producing different output.

5.4.6 Specification Extraction

As we discussed already, the RFC specifications leave certain aspects of hostname verification up to the implementations by not specifying the correct behavior in all cases. In these cases imposing specific restrictions in the implementations is challenging since we have to be careful to avoid breaking compatibility with existing implementations and valid certificates. In this section, we describe how the inferred DFA models for the different certificate templates can be used to infer a formal specification, which is compatible with existing implementations, for the cases where RFC specifications are vague.

Our main insight is the following: *For each certificate template, we can use the DFA accepting the set of hostnames accepted by all TLS implementations as a formal specification of the corresponding rule template.* The intuition behind this choice is that this specification is avoiding small idiosyncrasies of each library and it is thus very compact. On the other hand, if a vulnerability exists in this specification then this vulnerability must also exist in all tested implementations. Since each implementation is audited independently, our choice gives us confidence that our specification is secure from simple vulnerabilities while maintaining backward compatibility with the tested implementations.

Computing the Specification. In order to compute the corresponding specification for each certificate template, we proceed as follows: First, we obtain DFA models for all hostname verification implementations under test using HVLearn. Next, we compute the product DFA for all the inferred models. The product DFA accepts the intersection of the regular languages of each DFA. We compute the product DFA using standard automata algorithms [159]. The inferred formal specification for our set of implementations is represented by the product DFA of each DFA model. This product DFA can be then converted back to a regular expression to improve readability.

Finally, we would like to point out that computing the intersection of k DFAs have a

worst case time complexity of $O(n^k)$ where n is the number of states in each DFA [116]. However, in our case, the inferred DFAs are mostly similar and thus, the product construction is very efficient because intersecting two DFAs is not adding a significant number of states in the resulting product DFA. We provide more evidence supporting this hypothesis in Section 5.5.

5.5 Evaluation

The main goals of our evaluation of HVLearn are to answer the following questions: *(i) how effective HVLearn is in finding RFC violations in real-world hostname verification implementations?*, *(ii) how much do our optimizations help in improving the performance of HVLearn?*, *(iii) how does HVLearn perform compare to existing black-box or coverage-guided gray-box techniques*, and *(iv) can HVLearn infer backward-compatible specifications from the inferred DFAs of real-world hostname verification implementations.*

5.5.1 Hostname Verification Test Subjects

We use HVLearn to test hostname verification implementations in six popular open-source TLS implementations, namely OpenSSL, GnuTLS, MbedTLS (PolarSSL), MatrixSSL, JSSE, and CPython SSL, as well as in two popular TLS applications: cURL and HttpClient. Note that as several libraries like OpenSSL versions prior to 1.0.1 do not provide support for hostname verification and leave it up to the application developer to implement it. Therefore, applications like cURL/HttpClient that support different libraries are often forced to write their own implementations of hostname verification.

Among the libraries that support hostname verification, some like OpenSSL provide separate API functions for matching each type of identifier (i.e., domain name, IP addresses, email, etc.) and leave it up to application to select the appropriate one depending on the setting. In contrast, others like MatrixSSL combine all supported types of identifiers in one function and figure out the appropriate by inspecting the input string. Table 5.1 shows the hostname verification function/class names for all implementations that we tested and the types of identifier(s) that each of them supports. The last column shows physical source lines

Table 5.1: Hostname verification functions (along with the types of supported identifiers) in TLS libraries and applications

TLS Libs/Apps	Version	Supported Identifier(s)	Hostname Matching Function/Class Name	Approx. SLOC
OpenSSL	$\leqslant$ 1.0.1	–	–	–
OpenSSL	$\geqslant$ 1.0.2	CN/DNS	`X509_check_host`	314
		IP	`X509_check_ip`	308
		IP	`X509_check_ip_asc`	417
		EMAIL	`X509_check_email`	314
GnuTLS	3.5.3	CN/DNS/IP	`gnutls_x509_crt_check_hostname`, `gnutls_x509_crt_check_hostname2`	195
		EMAIL	`gnutls_x509_crt_check_email`	149
MbedTLS	2.3.0	CN/DNS	`mbedtls_x509_crt_verify`, `mbedtls_x509_crt_verify_with_profile`	193
MatrixSSL	3.8.4	CN/DNS/IP/ EMAIL	`matrixValidateCerts`	130
JSSE	1.8	CN/DNS/IP	`HostnameChecker`	202
CPython SSL	3.5.2	CN/DNS/IP	`match_hostname`	59
HttpClient	4.5.2	CN/DNS/IP	`DefaultHostnameVerifier`	257
cURL	7.50.3	CN/DNS/IP	`verifyhost`, `Curl_verifyhost`	300

of code (SLOC) for each host matching function/class as reported by the SLOCCount [164] tool. Note that the shown SLOC only count the parts of the code that perform hostname matching.

5.5.2 Finding RFC Violations with HVLearn

We use HVLearn to produce DFA models for each distinct certificate template corresponding to different patterns from the RFCs. Afterward, we detect potentially buggy behavior by both performing differential testing of output DFAs as well as checking individual DFAs for violations of regular-expression-based rules that we created manually as described in Section 5.4.5.

Table 5.2: A summary of RFC violations and discrepant behaviors found by HVLearn in the tested TLS libraries and applications

RFC Violations	RFC	OpenSSL	GnuTLS	MbedTLS	MatrixSSL	JSSE	CPython SSL	cURL	HttpClient	HttpClient*
Invalid hostname character										
Only alphanumeric and `'-'` matches in hostname	1035	✗	✗	✗	✓	✗	✗	✗	✗	✗
Case-insensitive hostname										
Match CN in case-insensitive manner	5280, 6125	✓	✓	✓	✓	✓	✓	✓	✗	✗
Wildcard										
Not attempt to match wildcard not in left-most label (CN/DNS: `aaa.*.aaa`)	6125	✓	✓	✓	✓	✗	✓	✓	✗	✓
IDN and wildcard										
Not attempt to match wildcard fragment in IDN (`xn--a*.aaa`)	6125	✓	✓	✓	✓	✗	✓	✓	✗	✓
Common name and subjectAltName										
No CN checked when DNS presents	6125	✓	✓	✓	✗	✓	✓	✓	✓	✓
No CN checked when any SAN ID presents	6125	–	✗	–	✗	✗	✓	✓	✗	✗
Email-based certificate										
Case-sensitive on local-part of email attribute in SAN	5280	✓	✓	–	✗	–	–	–	–	–
IP address-based certificate										
Not attempt to match IP address with DNS (DNS: `1.1.1.1`)	1123	–	✗	✗	✗	✓	✓	✓	✓	✓
Discrepancies										
Wildcard										
Attempt to match wildcard with empty label (hostname: `.aaa.aaa` with CN/DNS: `*.aaa.aaa`)	–	✓	✓	✗	✗	✗	✗	✗	✓	✓
Attempt to match wildcard in public suffix (CN/DNS: `*.co.uk`)	6125	✓	✗	✓	✓	✓	✓	✓	✓	✗
Embedded NULL character										
Allowed NULL character in CN	–	✓	✓	✓	✗	✓	✓	✓	✓	✓
Allowed NULL character in SAN	–	✓	✓	✗	✗	✓	✓	✓	✓	✓
Match NULL character hostname: `b.b\0.a.a`, CN/DNS: `b.b\0.a.a`	–	✗	✗	✗	✗	✓	✓	✗	✓	✓
Other invalid hostname										
Partially match suffix (hostname: `.a` with CN/DNS: `a.a`, `a.a.a`)	1035	✓	✗	✗	✗	✗	✗	✗	✗	✗
Match trailing dot (hostname: `aaa.aaa` with CN/DNS: `aaa.aaa`)	–	✗	✗	✗	✗	✗	✗	✓	✗	✗

HttpClient*: HttpClient with `PublicSuffixMatcher`

For RFC Violation: ✓= OK, ✗= RFC violate, – = libraries/applications do not support • For Discrepancies: ✓= Accept, ✗= Reject

Table 5.2 presents the results of our experiments. We evaluated a diverse set of rules from four different RFCs [2, 3, 11, 14]. We found that every rule that we tested is violated by at least one implementation, while on average each implementation is violating three RFC rules. Several of these violations have severe security implications (e.g., mishandling wildcard characters in international domain names, confusing IP addresses as domain names etc.). We describe these cases along with their security implications in detail in Section 5.6.

Note that the library with the most violations is JSSE (four violations), while HttpClient is the application with the most violations (five violations). OpenSSL, MbedTLS, and CPython SSL only have two violations each, having common the violation of matching invalid hostnames. The interested reader can find an extended description of our results in the Appendix (Table B.1).

5.5.3 Comparing Unique Differences between DFA Models

In order to evaluate the discrepancies between all different hostname verification implementations, we computed the number of differences for each pair of hostname verification implementations in our test set. Recall that for two given DFA models we define the number of differences as the number of simple paths in the product DFA which lead to a different output being produced by the two models [29].

Table 5.3 presents the results of our experiment. For example, OpenSSL and GnuTLS have 95 discrepancies in total. This is obtained by summing up the number of unique paths that are different between the inferred DFAs for each common name in Table B.1. Note that all pairs of implementations contain a large number of unique cases under which they produce a different output. As seen in Table 5.3, each pair of tested implementation has 127 unique differences on average between them. We note that some differences only imply ambiguous RFC rules while some reveal the potential invalid hostnames or RFC violation bugs. The interested reader can find a more detailed list of the unique strings that each implementation is accepting in Table B.1 in the Appendix. In any case, we find the fact that all implementations of such a security critical component of the TLS protocol present such a larger number of discrepancies to be an alarming issue since it signifies either a poor

Table 5.3: Number of unique differences between automata inferred from different TLS implementations

	OpenSSL	GnuTLS	MbedTLS	MatrixSSL	JSSE	CPython	HttpClient	cURL
OpenSSL	--	95	98	99	282	92	482	187
GnuTLS	--	--	6	38	127	34	214	56
MbedTLS	--	--	--	44	97	28	220	50
MatrixSSL	--	--	--	--	37	25	58	94
JSSE	--	--	--	--	--	69	177	110
CPython	--	--	--	--	--	--	108	54
HttpClient	--	--	--	--	--	--	--	414
cURL	--	--	--	--	--	--	--	--

implementation of the specification or vagueness in the specification itself. Our analysis suggests that both cases are present in practice.

5.5.4 Comparing Code Coverage of HVLearn and Black/Gray-box Fuzzing

In order to compare HVLearn's effectiveness in finding bugs with that of black/gray-box fuzzing, we investigate the following research question:

RQ.1: How HVLearn's code coverage differ from black/gray-box fuzzing techniques?

We compare the code coverage of the tested hostname verification implementations achieved by HVLearn and two other techniques, black-box fuzzing, and coverage-guided gray-box fuzzing. We describe our testing setup briefly below.

HVLearn: HVLearn leverages automata learning that invokes the hostname verification matching routine with a predefined certificate template and alphabet set. HVLearn adaptively refines a DFA corresponding to the test hostname verification implementation by querying the implementation with new hostname strings. We measure the code coverage achieved during the learning process until it finishes. We also monitor the total number of queries NQ, which comes from both the membership and the equivalence queries.

Black-box Fuzzing: With the same alphabet and certificate template used by HVLearn, we randomly generate NQ strings and query the target TLS hostname verification function with the same certificate template. Note that the black-box fuzzer generates independent random strings without any sort of guidance.

Coverage-guided Gray-box Fuzzing: Unlike black-box fuzzing, coverage-guided gray-box fuzzing tries to generate more interesting inputs by using evolutionary techniques to the input generation process. In each generation, a new batch of inputs are generated from the previous generation through mutation/cross-over and only the inputs that increase code coverage are kept for further changes. Coverage-guided gray-box fuzzing is a popular technique for finding bugs in large real-world programs [22, 125].

To make it a fair comparison with HVLearn, we implemented our own coverage-guided gray-box fuzzer as existing tools like AFL do not provide an easy way of restricting the mutation outputs within a given alphabet. With the same alphabet set, we initialize the fuzzer with a set of strings of varying lengths as the seeds maintained in a queue Q. The seeds are then used by the fuzzer to query the target hostname verification implementation. After finishing querying, using the seeds, the fuzzer gets the string $S = dequeue(Q)$. It randomly mutates one character within S and obtains S'. Then it uses the mutated S' to query the target. If the mutated string S' increased code coverage, we store it in the queue for further mutation, i.e., $enqueue(S', Q)$. Otherwise, we throw it away. The fuzzer is thus guided to always mutate on the strings that have better code coverage. The fuzzer iteratively performs this enqueue/dequeue operations for NQ rounds, and we obtain the final code coverage COV_{randmu} of each functions TLS implementations. Note that we keep the test certificate template fixed during the entire test.

We use the percentage of lines executed, which are extracted by Gcov [88], as the indicator for the code coverage. Considering that hostname verification is a small part of an TLS implementation, we do not compute the percentage of lines covered with respect to the total number of lines. Instead, we calculate the percentage of line coverage within each function and only take into account the functions that are related to hostname verification.

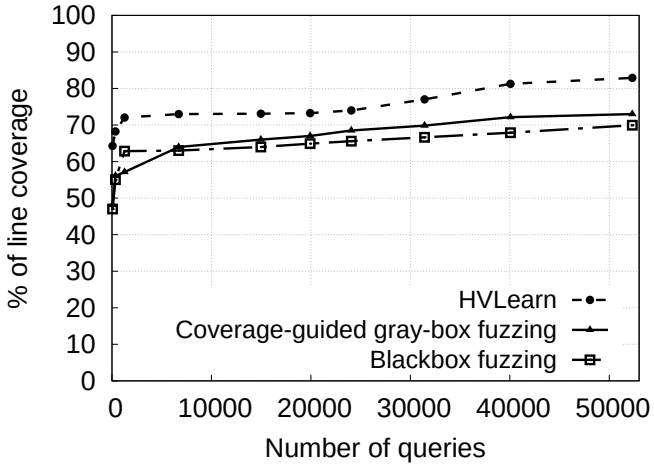


Figure 5.4: Comparison of code coverage achieved by HVLearn, gray-box fuzzing, and black-box fuzzing for OpenSSL hostname verification.

Result 1: *HVLearn achieves 11.21% increase in code coverage on average when comparing to the black/gray-box fuzzing techniques.*

Therefore, let LE(f) be the number of lines executed of function f in the SI and L(f) be the total number of lines of f, the code coverage can be defined in the following equation: $coverage = \frac{\sum_{i=1}^{m} LE(f_i)}{\sum_{i=1}^{m} L(f_i)}$, where $f_1, f_2, \cdots, f_m$ are the functions that are relevant to hostname verification. Figure 5.4 illustrates the code coverage comparison, which shows that HVLearn achieves significantly better code coverage compared to the black/gray-box fuzzing techniques.

5.5.5 Automata Learning Performance

HVLearn is largely based on the KV algorithm and the Wp-method in order to perform its analysis. It is therefore crucial to thoroughly evaluate the different parameters of these algorithms and their impact on the performance of HVLearn. We will now evaluate the effect of each different parameter of the learning algorithms in the overall performance of HVLearn.

RQ.2: How does the alphabet size affect HVLearn's performance in practice?

As discussed in Section 5.3.3, the alphabet size impacts the performance of our system. In theory, the performance of both the KV algorithm and the Wp-method, depends on the size of the input alphabet. We perform two experiments for evaluating the extent to which the alphabet size affects the performance of our learning algorithm component in practice. In the first experiment, we evaluate the effect of increasing the size of the alphabet in real world DNS names. For this experiment, we used our system in the default configuration with all optimizations (e.g., query cache and EQ optimizations) enabled and we set the Wp-method depth to 1. We used the CPython's SSL implementation as the hostname verification function for these experiments.

Figure 5.5 shows the results of our experiment. Notice that, starting from an alphabet size of 9, each additional character we include in the alphabet will cause the learning algorithm to perform at least 10% more queries in order to produce a model, for both DNS names, while this percentage is only increasing when in larger alphabet sizes.

We also measure the effect of increasing the alphabet size on the overall running time of our system. To perform this experiment we used the same setup as our previous experiment and evaluated the performance of HVLearn with a certificate containing the common name `"*.aaa.aaa"`. Table 5.4 shows the results of this experiment. We notice that the increase in the membership queries directly translates in an increased running time. Specifically, by adding 5 additional characters in the alphabet (from 2 to 5), we notice that the running time increases 7 times. Similar results can be observed when we add more characters in the alphabet set.

> ***Result 2:*** *Adding just one symbol in the alphabet set incurs at least 10% increase in the number of queries. Thus, the succinct alphabet set utilized by HVLearn is crucial for the system's performance.*

RQ.3: Does membership cache improve the performance of HVLearn?

Table 5.4 presents the number of queries required to infer a model for the certificate template with common name `"*.aaa.aaa"` with and without utilizing a membership query cache over different alphabet sizes. We notice that the cache is consistently helping to

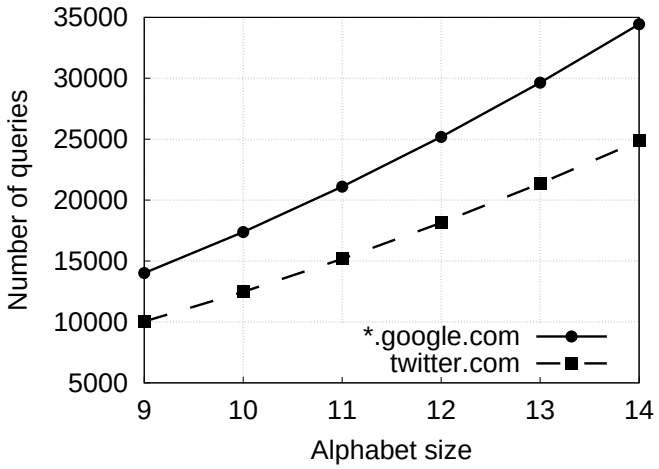


Figure 5.5: Number of queries required to learn an automaton with different alphabet sizes (with Wp-method depth=1 and equivalence query optimization).

Table 5.4: HVLearn performance for common name `*.aaa.aaa` with Wp-method depth=1 (CPython SSL implementation)

Alphabet Size	W/o Cache	With Cache				
	#Queries	#Queries				Average Time (sec)
	Total	Total	Membership	Equivalence		
				Counterexample	Membership	
2	883	226	136	2	90	3.10
5	3,049	1,582	436	2	1,146	21.61
7	5,163	3,156	636	2	2,520	42.24
10	9,339	6,522	936	2	5,586	86.92
15	18,979	14,812	1,436	2	13,376	196.35

reduce the number of membership queries required to infer a model. Overall, the cache is reducing the number of queries by 42%, thus significantly improving the efficiency of our system. Therefore, for the rest of the experiments in this section, we utilize our system with the membership query cache enabled.

Result 3: *Membership cache is offering, on average, a 42% decrease on the number of membership queries made by the learning algorithm.*

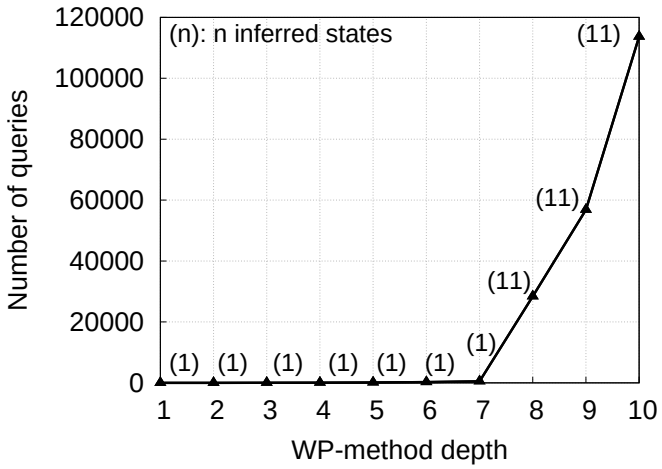


Figure 5.6: The number of queries needed to learn the DFA model of CPython certificate verification for different Wp-method depth values (without equivalence query optimization).

RQ.4: How does Wp-method's depth parameter affect HVLearn's performance and accuracy?

As discussed in Section 5.4.4, the number of queries performed by the Wp-method is exponential on the customizable depth parameter. We evaluated how this exponential term is affecting the number of queries in practice and moreover, what is the effect of different values of the depth parameter on the correctness of the models inferred by HVLearn.

For our first experiment, we explore the correlation between the overall number of membership queries and the corresponding depth parameter. The results of this experiment are presented in Figure 5.6 and Table 5.5. In order to ensure that the experiment finishes within a reasonable time, we further reduced the alphabet size only to two symbols. the results clearly show that the dependence between the depth parameter and the overall number of queries performed by the learning algorithm is clearly exponential, and in fact exactly matches the $O(|\Sigma|^d)$ bound where d is the depth parameter as discussed in Section 5.4.4. Notice that when the depth parameter of the Wp-method is set to a value less than 8, HVLearn fails to infer any aspect of the target implementation and outputs a single state DFA model that rejects all hostnames as shown in Table 5.5.

Result 4: *Large values of the Wp-method depth parameter result in impractical running times while small values result in incomplete models.*

RQ.5: How much improvement is offered by the equivalence query optimization in HVLearn?

The previous experiment clearly demonstrates that the Wp-method alone is not efficient enough to accurately analyze a variety of different templates with HVLearn. Using our full alphabet, inferring a complete model for the common name `"*.aaa.aaa"` requires the depth parameter to be ≥ 8 as shown in Table 5.5. With our full alphabet of 13 symbols this would require around 2^{30} queries based on the query complexity of the algorithm. We find that even running the algorithm with a depth of 6, which is still not able to infer a complete model, results in more than 68 million queries.

Therefore, our equivalence query optimization is a crucial component of HVLearn that allows it to produce accurate DFA models that can be used to evaluate the security and correctness of the implementations. As we can see from Table 5.5, using our equivalence query optimization and a depth parameter of just 1, our system is able to produce a complete model for a given certificate template. Running the same experiment with the alphabet size 15, we found that HVLearn infers a correct model using only 14,812 queries as shown in Table 5.4.

Result 5: *EQ optimization is providing, in some cases, over one order of magnitude improvement on the number of queries required to infer a complete DFA model.*

5.5.6 Specification Extraction

Let us now examine how we can utilize HVLearn's specification extraction functionality in order to infer a practical specification for the rule corresponding to the common name `"*.a.a"`. This rule corresponds to the basic wildcard certificate case where a wildcard is found in the leftmost label of the identifier. Nevertheless, Figure 5.7 demonstrates that even for this simple rule, the corresponding DFA models for different implementations present

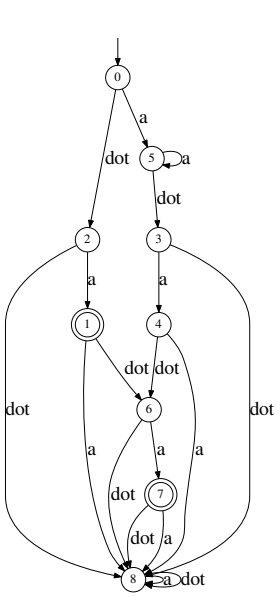
Table 5.5: The number of queries needed to learn the DFA model of CPython certificate verification for different Wp-method depth values

Wp.	W/o EQ Optimization			With EQ Optimization		
Depth	#Queries	#States	Complete?	#Queries	#States	Complete?
1	7	1	✗	226	11	✓
2	15	1	✗	448	11	✓
3	31	1	✗	890	11	✓
4	63	1	✗	1,778	11	✓
5	127	1	✗	3,554	11	✓
6	255	1	✗	7,104	11	✓
7	511	1	✗	14,207	11	✓
8	28,415	11	✓	28,415	11	✓
9	56,831	11	✓	56,831	11	✓
10	113,663	11	✓	113,663	11	✓

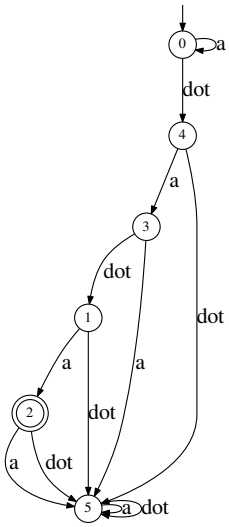
obvious discrepancies. For example, DFA model (a) accepts the hostname `".a"`, model (b) accepts the hostname `".a.a"`, while model (d) accepts the hostname `"a.a.a."`. Only model (c) perform the most intuitive matching by only accepting hostnames matching the regular expression `"a+.a.a"` (here `'+'` denotes one or more repetitions of the character `'a'`).

By computing the intersection between all DFA models, we obtain the intersection DFA model (e). Our first observation is that the intersection DFA has only 6 states and it is thus very compact as discussed in Section 5.4.6. Furthermore, we notice that the intersection DFA is the same as DFA (c) that corresponds to the most natural implementation of the corresponding rule. More importantly, even if we compute the intersection without including model (c), we will still infer the same specification. Thus, we conclude that computing the intersection of DFA models, even from implementations which fail in different ways, can often produce compact and natural specifications.

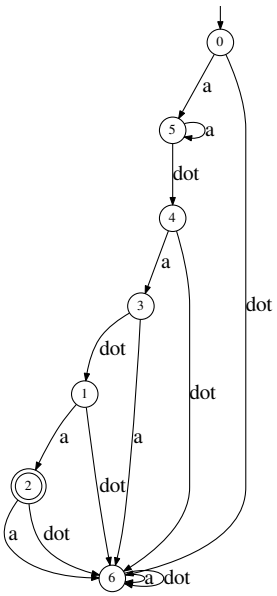
Size of Inferred Models. In general, the actual size of the inferred models is heavily dependent on the implementation details of the tested system. However, we expect that the DFA models inferred by our system will have around $l + 2$ states, where l is the length of the common name in the certificate template. Indeed, if we consider the inferred DFAs in Figure 5.7 we can notice that, for the common name `"*.a.a"` with length $l = 5$, the



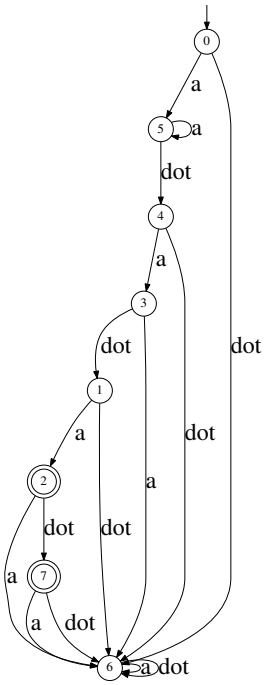
(a) OpenSSL



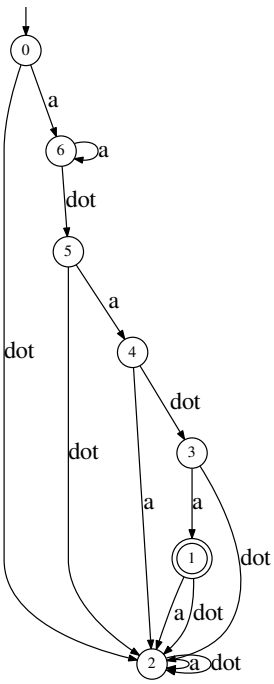
(b) GnuTLS, JSSE, and HttpClient



(c) MbedTLS, MatrixSSL, and CPython



(d) cURL



(e) Intersection

Figure 5.7: TLS implementations' DFA and intersection DFA with CN DNS: `*.a.a` and alphabet: `{a, dot}`

average number of states is 6.9, which is very close to the expected 7 states. Intuitively, the reasoning behind this size is that a DFA for matching a string of length l is expected to have $l + 2$ states in general where l states are moving the DFA forward towards the accepting state while the additional 2 states include the initial state and a *sink* state where the DFA goes when no match is found.

5.6 Case Study of Bugs

The goal of our study aims at understanding the severity of potential exploitation by incorrect or unclear hostname check in certificate verification. We are also interested in finding any inconsistency of TLS implementations' hostname checks with what RFC specifies. In this section, we present some interesting cases we achieved from the result of our experiment or corner cases we found.

5.6.1 Wildcards within A-labels in IDN identifiers

RFC 6125 strictly prohibits matching a certificate with an identifier containing wildcards embedded within an A-label of an IDN. For a certificate with an identifier of the form `"xn--aa*"`, it is very difficult to predict the set of unicode strings that will be matched after they are transformed into the punycode format due to the complexity of the transformation process. This inability to easily predict the set of hostnames which match an A-label with an embedded wildcard often present avenues for man-in-the-middle attacks.

Hostname verification implementations which match identifiers with wildcards embedded within A-labels have been found recently in the Ruby OpenSSL extension [58] and the NSS library used by Mozilla Firefox [57]. These issues were identified as security vulnerabilities by the developers of the corresponding products.

Using our framework, we identified that both JSSE and HttpClient (without using `PublicSuffixMatcher` [27] in constructor) were also vulnerable to this issue. Our tool also reported that the other tested libraries/applications were not affected.

5.6.2 Confusing Order of Checking between CN and SAN Identifiers.

RFC 6125 explicitly specifies that applications should not attempt to match the hostname with the subject CN when any subjectAltName identifiers are present, regardless of whether there is a match in subjectAltName as shown in Section 5.2). We found a number of violations of that rule using HVLearn as described in Table 5.2. We also found that MatrixSSL exhibits an interesting behavior in such cases.

More specifically, MatrixSSL matches the CN identifier before attempting to match any identifiers in the SAN even if they are present in the certificate. Note here that the CN does not have any strong restrictions on its content and may even contain non-FQDN characters (e.g., UTF-8).

Therefore, it is possible that certain certificate authorities, following the instructions in RFC 6125, will not check the CN in the presence of SAN identifiers and will issue a certificate regardless of the value in the CN as long as the user is successfully identified as the owner of the domains in the SAN identifier. Albeit natural, this choice will render applications using MatrixSSL vulnerable to a simple man-in-the-middle attack.

Specifically, an attacker can generate a signed certificate with a SAN identifier for a domain owned by the attacker, say `"www.attacker.com"` and have the CN field set to the victim domain, say `"www.bank.com"`. MatrixSSL will first check the CN and omit to check the SAN identifiers. Therefore, MatrixSSL will allow the attacker to hijack any domain which is present in the CN field (e.g., `www.bank.com`). Due to this implication, Google Chrome recently (from version 58) removed the support for common name matching in certificate [134].

5.6.3 Hijacking IP-based Certificates

Section 2.3.1 of domain names implementation and specification in RFC [2] dictates that the preferred name (label) should only begin with a letter character. However, RFC [3] changed this restriction to allow the first character to be a letter or a digit. This change introduced valid DNS names which are identical to IP addresses.

Unfortunately, the fact that IP addresses are also valid DNS names may open a new avenue for an attack as we describe below. Notice that, for this attack to become practical, a

Table 5.6: Behaviors of TLS implementations for X.509 certificates with IPv4 addresses in CN/subjectAltName

TLS Library/Application	Certificate with IPv4 in	
	Subject CN	SubjectAltName DNS
OpenSSL	app	app
GnuTLS	accept	accept
MbedTLS	accept*	accept*
MatrixSSL	accept	accept
JSSE	reject	reject
CPython SSL	accept	reject
HttpClient	accept	reject
cURL	accept	reject

accept*: library/application does not support IP-based certification verification but allows IPv4-format string in hostname verification.

numeric Top Level Domain (TLD) in the range 0-255 must exist, something that is currently unavailable. Nevertheless, our description should be taken as a precautionary note for new TLDs.

The attack is based on the fact that certain implementations first check if the given hostname matches the certificate's CN/SAN as a domain name and afterward as IP address. Therefore, consider an attacker controlling an IP address, say `80.50.12.33` and holding an IP-based certificate with that IP address. Then, assuming that `"33"` is a valid TLD, the same entity is automatically in possession of a certificate for the *domain name* `"80.50.12.33"` and can perform man-in-the-middle attacks on that domain.

We evaluated whether this attack is feasible in current TLS implementations. Table 5.6 shows the results of our evaluation. All libraries/applications which are marked with an *accept* either in the subject CN or subjectAltName DNS columns are vulnerable to this attack. Even though this issue is not currently exploitable, it presents a security risk for these libraries in case numerical TLDs are introduced in future.

5.6.4 Embedded NULL Bytes in CN/SAN Identifiers

In 2008, Kaminsky et al. [111] demonstrated a vulnerability in the hostname verification implementations of popular TLS libraries where early NULL-byte (`\0`) terminations in an X.509 CN causes some libraries to recognize different CN values. In a nutshell, a client accepts certificate from an attacker's subdomain `"www.bank.com\0.attacker.com"` when attempting to connect to `"www.bank.com"` and therefore allow the attacker to hijack the connection.

In order to defend against this attack, two lines of defense were followed. The first option was to reject any certificate containing NULL bytes embedded within any CN/SAN identifiers. The second line was to simply patch the API functions which retrieve the CN/SAN identifiers from the certificate in order to recover the entire identifier even in the presence of embedded NULL bytes.

We thoroughly evaluated the defense implemented in each TLS library. Table 5.7 presents the results of our evaluation. The second column describes whether the TLS library allows embedded NULL bytes, the third column presents the corresponding API function which is used to retrieve the CN/SAN identifier, and the fourth column describes whether the API call also returns the length of the corresponding CN/SAN identifier. Note that this is a very important feature since, otherwise, the application using the TLS library cannot know where the identifier string is terminating. We notice that this important feature is implemented by all libraries except JSSE. Notice though that, even though JSSE is not returning the length of the corresponding identifier, since JSSE is written in Java, it is not vulnerable to the embedded NULL byte attacks because Java strings are not NULL terminated.

Despite the fact that TLS implementations take precautions against embedded NULL byte attacks, this doesn't imply that the applications using the libraries are also secure. Indeed, applications implementing the hostname verification functionality must ensure that they do not use vulnerable functions such standard string comparison function from libc (e.g., `strcmp, strcasecmp, fnmatch`), as they match strings in NULL-termination style.

In order to evaluate the security of applications using TLS libraries against embedded NULL byte attacks, we conducted a manual audit against several applications. Unfortu-

Table 5.7: Support for embedded null character in CN/subjectAltName in different TLS libraries

SSL Libraries	ID	Allows Embedded NULL?	Function / Structure Name	Returns Length
OpenSSL	CN	✓	`X509_NAME_get_text_by_NID()`	✓
	CN	✓	`X509_NAME_get_text_by_OBJ()`	✓
	CN	✓	`X509_NAME_get_index_by_NID()`[1]	✓
	CN	✓	`X509_NAME_get_index_by_OBJ()`[1]	✓
	SAN	✓	`X509_get_ext_d2i()`[2]	✓
GnuTLS	CN	✓	`gnutls_x509_crt_get_dn_by_oid()`	✓
	SAN	✓	`gnutls_x509_crt_get_subject_alt_name()`	✓
MbedTLS	CN	✓	`mbedtls_x509_name`	✓
	SAN	✗	`mbedtls_x509_sequence`	✓
MatrixSSL	CN	✗	`x509DNattributes_t`	✗
	SAN	✗	`x509GeneralName_t`	✓
JSSE	CN	✓	`getSubjectX500Principal()`	✗
	SAN	✓	`getSubjectAlternativeNames()`	✗
CPython SSL			–Functionality not exposed to apps –	

[1] followed by `X509_NAME_get_entry()`

[2] followed by `sk_GENERAL_NAME_value()`

nately, we found several popular applications being vulnerable to man-in-the-middle attacks using embedded NULL byte certificates. Some examples include FreeRadius server [81] which is one of the most widely deployed RADIUS (Remote authentication dial-in user service) servers, OpenSIPS [145] which is a popular open-source SIP server, Proxytunnel [155] which is a stealth tunneling proxy, and Telex Anticensorship system [176] which is an open-source censorship-circumventing software.

An important takeaway from this section is that embedded NULL byte attacks, even though addressed at the TLS library level, still present a very realistic and overlooked threat for applications using these libraries.

5.7 Disclosure and Developer Responses

We notified the developers of each affected library/application for all of our findings, including RFC violations and discrepancies. In this section, we present an overview of the developer responses for each different library/application.

GnuTLS. The GnuTLS team is currently working on a patch to fix the issue of seeking a match in the CN when an IP address identifier is in the subjectAltName [91]. The developers also plan to provide a way to specify the identifier type in order to avoid the confusion between hostnames and IP addresses [90]. Additionally, the team plans to remove a fallback option which matches an IP address with a subjectAltName DNS [92], thus resolving the potential attack presented in Section 5.6.3 [89]. Finally, GnuTLS has recently introduced IDNA2008 support in version 3.5.9 and performs extensive checks to verify the format of the DNS names stored in the certificate.

MbedTLS. As the time of writing, we have informed the issue of lacking of hostname validation and matching IP address on subjectAltName DNS. The developer are studying and investigating them.

MatrixSSL. MatrixSSL is prioritizing the fixes for the RFC violations, including the incorrect order of checking between subject CN and subjectAltName identifier (violation of RFC 6125) and matching the local-part of an email address in a case-insensitive manner (violation of RFC 5280). These fixes are deployed in their new version 3.9.0 [128]. This version also addresses other discrepancies we reported by providing an optional flag for hostname input validation, and providing parameters for users in order to specify the type of the identifier (e.g., DNS, IP ADDR) in order to address the attack discussed in Section 5.6.3.

JSSE. The JSSE team does not consider RFC 6125 compliance to be a feature of the current version of the library. However, the team informed us that they are currently working on plans to add compliance with RFC 6125 in the next versions of the library [148].

CPython SSL. CPython plans to deprecate their hostname verification implementation and directly use OpenSSL's implementation in the next release.

OpenSSL. The OpenSSL team decides not to address the issue of matching a partial hostname suffix of a subject CN/subjectAltName, as this discrepancy is not an RFC violation. For the other discrepancies e.g., matching a wildcard in a public suffix or matching an invalid hostname, the OpenSSL team believes that they should be handled at the application level or by certificate authorities and therefore, they should not be fixed in the library itself.

HttpClient. The HttpClient team has addressed the violations of matching a subject CN in case sensitive manner (violation of RFC 6125 and RFC 5280) and attempting to match subject CN when a subjectAltName is present (violation of RFC 6125). These issues are resolved in version 4.5.3, which is currently an alpha release [104]. The HttpClient team decided not to address the other reported issues as they are handled correctly if the application calls the `DefaultHostnameVerifier` with the `PublicSuffixMatcher` in the verifier constructor.

5.8 Contribution

Our testing framework, HVLearn is able to discover 8 unique previously unknown violations of RFC specifications in 7 TLS hostname verification implementations we tested. All of those violations are critical and can render the affected implementations vulnerable to active man-in-the-middle attacks. We report our findings to the developers of implementations we tested. Currently, GnuTLS, JSSE, MatrixSSL, and HttpClient have deployed fixes for the violations we have reported as mentioned in Section 5.

Additionally, our framework can be used to discover 141 previously known CVEs [54] regarding TLS hostname verification, present in popular applications such as internet browsers (Opera, Apple Safari, Google Chrome), wget, Chase mobile banking, Paypal mobile application, AOL Instant Messenger (AIM), cURL and a variety of TLS libraries e.g., libcurl, OpenSSL, GnuTLS.

84 out of those found 141 CVEs do not verify hostname during certification validation. The rest, 57 CVEs, regard cases where the hostname verification is implemented incorrectly e.g., mishandling certificate with embedded NULL in CN/SAN identifier, and RFC violations such as allowing wildcard in IP address, and matching wildcard on IDN.

Finally, we have made HVLearn open-source so that the community can continue to build on it. The framework and all certificate templates can be accessed at `https://github.com/HVLearn`.

5.9 Conclusion

We designed, implemented and extensively evaluated HVLearn, an automated black-box automata learning framework for analyzing different hostname verification implementations. HVLearn supports automated extraction of DFA models from multiple different implementations as well as efficient differential testing of the inferred DFA models. Our extensive evaluation on a broad spectrum of hostname verification implementations found 8 RFC violations with serious security implications. Several of these RFC violations could enable active man-in-the-middle attacks. We also discovered 121 unique differences on average between each pair of inferred DFA models. In addition, given that the RFC specifications are often ambiguous about corner cases, we expect that the models inferred by HVLearn will be very useful to the developers for checking their hostname verification implementations against the RFC specifications and therefore can help in reducing the chances of undetected security flaws.

Chapter 6

Conclusion

6.1 Closing Remarks

The increase of user personal and sensitive information on the web is massive. Regardless of how important of the information, due to the design of the Internet, this information is transferred through multiple untrusted entities, ranging from application-level (e.g., web applications, web browsers) to network-level (e.g., routers, ISPs). Unfortunately, these entities are often the target of attacks, such as WiFi eavesdroppers, ISP- and state-level adversaries. An internet user needs to trust these applications and services in protecting their information against the large spectrum of attacks and threat models of today. A challenging decision these entities have to make is to select the appropriate set of the available defenses that should be adopted, however, an equally challenging responsibility is to correctly implement and deploy the selected defenses.

As we explored in this thesis, however, oftentimes we observe numerous web-based security incidents on systems even with security mechanisms in-place, due to lack of understanding of correctly integrating them with other already-existing protocols. The incorrect or imprecise integration can lead to additional vulnerabilities that attackers can exploit to leak user information. Things become worse when it comes to real-world where, as we presented, services continue to sacrifice security for usability. For example, as demonstrated by our information exposure attacks on major websites that support HTTPS (Chapter 3) [162], and incomplete deployment of web encryption enforcement (Chapter 4) [163].

In addition to how user information is handled on the server-side, web clients also play a crucial role in the comprehensive protection of the user. Specifically, one cannot neglect incorrect implementations of web encryption in web applications including their underlining protocols (i.e., TLS protocols). For example, we discovered RFC violations and discrepancies on TLS in hostname verification, several of which could be used in man-in-the-middle attacks (Chapter 5) [160]. We also demonstrated why setting HTTP as the default behavior of web browsers is a possible attack surface that adversaries could take advantage to hijack sessions over the network (Chapter 3) [162].

In this dissertation, we first presented various case studies of incomplete HTTPS deployment that results in massive information leakage including top websites (e.g., Google, Amazon, Yahoo, Ebay, Bing) on real user traffic. We monitored part of our university's public wireless network over the course of one month and identified over 282K user accounts that exposed the HTTP cookies required for the hijacking attacks. To this end, based on what we observed on our measurement and audit on the top websites, we create a developer guideline (Section 3.8) for correctly deploying web encryption, HTTPS, and web encryption enforcement, while using HSTS is a key takeaway for web administrators.

On the client-side, in addition to HSTS, we evaluate the most effective browser extensions designed to force users to connect through HTTPS, including HTTPS Everywhere. 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.

Finally, we designed and implemented a testing framework, HVLearn, that allows TLS or web browser developers to test their hostname verification functions against other implementations in order to find RFC violations and discrepancies. HVLearn is specially designed to handle string testing against specification and output an inferred DFA using automated learning. The framework supports automated extraction of the inferred DFA, which can be easily interpreted and utilized to identify the differences from a given specification. We evaluated HVLearn on a broad spectrum of hostname verification implementations and found 8 RFC violations with serious security implications, some potentially could enabling

active man-in-the-middle attacks. In addition, given that the RFC specifications are often ambiguous about corner cases, we expect that the models inferred by HVLearn will be very useful to developers for checking their hostname verification implementations against the RFC specifications and therefore can help in reducing the chances of undetected security flaws.

6.2 Future Directions

People utilize the web for more and more aspects of their life. Due to this trend, we expect the amount of the user information to continuously expand. In an effort to improve the privacy of the web user, we presented current challenges in web encryption and web encryption enforcement as well as introduced novel testing frameworks to enhance the security of encryption. However, there is still many interesting research questions and challenges that remain open. In this final section, we outline our vision towards user privacy and web encryption in respect to the evolution of the online services.

6.2.1 Web Encryption

We showed that information leakage on the web is still prevalent in practice regardless of the deployment of HTTPS. This is because of websites employing encryption but not ubiquitously. Additionally, to make the problem worse, HTTP cookies do not have strong integrity properties thus using HTTP cookies is a "bearer" token that makes information leakage even more possible. While users cannot fully trust the service to always correctly deploy web encryption, we argue that users should have an option to better protect themselves. Although the extension like HTTPS Everywhere attempts to enforce HTTPS, the extension still relies on how the websites deploy HTTPS.

In recent years, we have seen an increase in the widespread use of security protections to authenticate, through two-factor, biometrics, and extra devices. For example, in 2017 Google launched their "Advanced Protection" mechanism which offers high-risk users the ability to U2F authentication (universal two-factor) in addition to password [93] authentica-

tion. Unfortunately, none of these security enhancements focus to strengthen the encryption of the connection after login.

HTTPS by Default. Most websites already support HTTPS, however, it is crucial to extend modern major browsers to use HTTPS by default. Specifically, all requests should be made to HTTPS by default and then fall back to HTTP if the initial attempt is not successful. In order to be comprehensive, the browser should prepend all addresses in the address bar and any resource full URL with `https://` (instead of `http://`).

Due to the lack of universal deployment of HTTPS in websites, connecting to HTTPS by default could slow down websites. Browsers could cache whether the initial attempt to connect to HTTPS was successful or not and then refer to this result for subsequent connections.

6.2.2 Hostname Checking in Certificate Authority

We developed HVLearn to perform differential testing against hostname verification of each TLS client implementation. Despite the numerous advantages on testing hostname verification on the client side, our current design is also applicable for testing the correctness of hostname certificate authority when issuing a certificate in order to prevent rogue certificates described in Section 2.4.3.

6.2.3 RFC Specification

The RFC specification covers many protocols, along with various procedures and concepts and acts as a standard guide for implementation. Here we describe some of the issues we found and point out room for improvement in future work.

Specification of Involved Protocols. As mentioned, the deployment of web encryption involves deploying various protocols (e.g., HTTP, HTTPS, HSTS), as well as hostname verification (e.g., TLS X.509, hostname, IDN). The vulnerabilities we found are the result of the improper integration of these protocols. Typically, each protocol is written separately and the integration process has no strict specification. For example, in our analysis, the

HTTPS specifications [5, 6] do not cover the implications of partial HTTPS deployment when HTTP cookies are used (Section 3.3), as well as the implications of hostname- and IP-based common name checking in X.509 certificate (Section 5.6).

TLS Specification. Our hostname verification testing framework detected numerous RFC violations and discrepancies we found in different implementations. Particularly the discrepancies we found, stem from unclear or unspecified corner cases in the specifications. This allows each developer to make her own decisions when implementing, something that could potentially lead to security flaws.

Another challenge in this work is that the certificate templates used by our testing set, were extracted from the specification manually. Although we tried to thoroughly generate certificates for every possible case, one might argue that it is possible to miss some corner cases. Therefore, in addition to descriptive RFC specifications, providing machine-readable specifications would allow developers to employ these specifications for testing. For example, using DFA to describe accepted strings for hostname verification. This would also help to remove unclear or unspecified corner cases.

Bibliography

[1] Expect-CT Extension for HTTP draft-ietf-httpbis-expect-ct-00. `https://tools.ietf.org/html/draft-ietf-httpbis-expect-ct-00`.

[2] RFC 1035 - Domain Names - Implementation and Specification.

[3] RFC 1123 - Requirements for Internet Hosts – Application and Support.

[4] RFC 1945 - Hypertext Transfer Protocol – HTTP/1.0.

[5] RFC 2660 - The Secure HyperText Transfer Protocol.

[6] RFC 2818 - HTTP Over TLS.

[7] RFC 3492 - Punycode: A Bootstring encoding of Unicode for Internationalized Domain Names in Applications (IDNA).

[8] RFC 4985 - Internet X.509 Public Key Infrastructure Subject Alternative Name for Expression of Service Name.

[9] RFC 5077 - Transport Layer Security (TLS) Session Resumption without Server-Side State.

[10] RFC 5246 - The Transport Layer Security (TLS) Protocol Version 1.2.

[11] RFC 5280 - Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile.

[12] RFC 5321 - Simple Mail Transfer Protocol.

[13] RFC 5890 - Internationalized Domain Names for Applications (IDNA): Definitions and Document Framework.

[14] RFC 6125 - Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS).

[15] RFC 6265 - HTTP State Management Mechanism.

[16] RFC 6797 - HTTP Strict Transport Security.

[17] RFC 6818 - Updates to the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile.

[18] RFC 7469 - Public Key Pinning Extension for HTTP.

[19] RFC 7568 - Deprecating Secure Sockets Layer Version 3.0.

[20] Alessandro Acquisti, Ralph Gross, and Fred Stutzman. Faces of Facebook: Privacy in the Age of Augmented Reality. *Black Hat Webcast*, 2011.

[21] Nadhem J Al Fardan and Kenneth G Paterson. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 526–540, 2013.

[22] American Fuzzy Lop (AFL) Fuzzer. `http://lcamtuf.coredump.cx/afl/`.

[23] Nader Ammari, Gustaf Björksten, Peter Micek, and Deji Olukotun. Am I Being Tracked? `https://www.accessnow.org/aibt/`.

[24] Chaitrali Amrutkar, Kapil Singh, Arunabh Verma, and Patrick Traynor. VulnerableMe: Measuring Systemic Weaknesses in Mobile Browser Security. In *Proceedings of the International Conference on Information Systems Security*, pages 16–34, 2012.

[25] Chaitrali Amrutkar, Patrick Traynor, and Paul C van Oorschot. An Empirical Evaluation of Security Indicators in Mobile Web Browsers. *IEEE Transactions on Mobile Computing*, 14(5):889–903, 2015.

[26] Dana Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[27] Apache Software Foundation. Class PublicSuffixMatcher. `https://hc.apache.org/httpcomponents-client-ga/httpclient/apidocs/org/apache/http/conn/util/PublicSuffixMatcher.html`.

[28] Apache Software Foundation. HttpComponents HttpClient Overview. `https://hc.apache.org/httpcomponents-client-ga/`.

[29] George Argyros, Ioannis Stais, Suman Jana, Angelos D. Keromytis, and Aggelos Kiayias. SFADiff: Automated Evasion Attacks and Fingerprinting Using Black-box Differential Automata Learning. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 1690–1701, 2016.

[30] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. DROWN: Breaking TLS Using SSLv2. In *Proceedings of the USENIX Conference on Security Symposium*, pages 689–706, 2016.

[31] Paul Barford, Igor Canadi, Darja Krushevskaja, Qiang Ma, and S. Muthukrishnan. Adscape: Harvesting and Analyzing Online Display Ads. In *Proceedings of the International Conference on World Wide Web*, pages 597–608, 2014.

[32] BBC News. NSA 'targets' Tor web servers and users – BBC News. `http://www.bbc.com/news/technology-28162273`, July 2014.

[33] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A Messy State of the Union: Taming the Composite State Machines of TLS. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 535–552, 2015.

[34] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, , Alfredo Pironti, and Pierre-Yves Strub. Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 98–113, 2014.

[35] Daniel Bleichenbacher. Chosen Ciphertext Attacks against Protocols based on The RSA Encryption Standard PKCS# 1. In *Proceedings of the International Conference on Advances in Cryptology*, pages 1–12, 1998.

[36] Andrew Bortz, Adam Barth, and Alexei Czeskis. Origin Cookies: Session Integrity for Web Applications. In *Proceedings of the Web 2.0 Security and Privacy*, 2011.

[37] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 114–129, 2014.

[38] Eric Butler. Firesheep. `http://codebutler.com/firesheep`.

[39] CAcert. Welcome to CAcert.org. `https://www.cacert.org/`.

[40] Joseph A. Calandrino, Ann Kilzer, Arvind Narayanan, Edward W. Felten, and Vitaly Shmatikov. "You Might Also Like:" Privacy Risks of Collaborative Filtering. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 231–246, 2011.

[41] Can I Use. Strict Transport Security. `http://caniuse.com/stricttransportsecurity`. (Accessed on 02/2018).

[42] CAPEC. CAPEC-102: Session Sidejacking. `https://capec.mitre.org/data/definitions/102.html`.

[43] CAPEC. CAPEC-62: Cross Site Request Forgery. `https://capec.mitre.org/data/definitions/62.html`.

[44] CAPEC. CAPEC-63: Simple Script Injection. `https://capec.mitre.org/data/definitions/63.html`.

[45] Claude Castelluccia, Emiliano De Cristofaro, and Daniele Perito. Private Information Disclosure from Web Searches. In *Proceedings of the International Conference on Privacy Enhancing Technologies*, pages 38–55, 2010.

[46] Abdelberi Chaabane, Gergely Acs, and Mohamed Ali Kaafar. You Are What You Like! Information Leakage Through Users' Interests. In *Proceedings of the Network and Distributed System Security Symposium*, 2012.

[47] Sze Yiu Chau, Omar Chowdhury, Md. Endadul Hoque, Huangyi Ge, Aniket Kate, Cristina Nita-Rotaru, and Ninghui Li. SymCerts: Practical Symbolic Execution for Exposing Noncompliance in X.509 Certificate Validation Implementations. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 503–520, 2017.

[48] Yuting Chen and Zhendong Su. Guided Differential Testing of Certificate Validation in SSL/TLS Implementations. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*, pages 793–804, 2015.

[49] Nicolas Christin, Sally S. Yanagihara, and Keisuke Kamataki. Dissecting One Click Frauds. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 15–26, 2010.

[50] Chromium Git Repositories. HSTS Preloaded List. `https://chromium.googlesource.com/chromium/src/net/+/master/http/transport_security_state_static.json`.

[51] Cisco. Configuring SPAN and RSPAN. `https://www.cisco.com/c/en/us/td/docs/switches/lan/catalyst4000/8-2glx/configuration/guide/span.html`.

[52] Jeremy Clark and Paul C. van Oorschot. SoK: SSL and HTTPS: Revisiting Past Challenges and Evaluating Certificate Trust Model Enhancements. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 511–525, 2013.

[53] Michael Cobb. HTTP vs. HTTPS: Is digital SSL certificate cost hurting Web security? – SearchSecurity. `http://searchsecurity.techtarget.com/answer/HTTP-vs-HTTPS-Is-digital-SSL-certificate-cost-hurting-Web-security`, 2010.

[54] Common Vulnerabilities and Exposures. `https://cve.mitre.org/`.

[55] cURL. Compare SSL Libraries. `https://curl.haxx.se/docs/ssl-compared.html`.

[56] CVE. CVE-2014-0092. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0092`, March 2014.

[57] CVE. CVE-2014-1492. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1492`, March 2014.

[58] CVE. CVE-2015-1855. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1855`, March 2015.

[59] CWE. CWE-1004: Sensitive Cookie Without 'HttpOnly' Flag. `https://cwe.mitre.org/data/definitions/1004.html`.

[60] Joeri De Ruiter and Erik Poll. Protocol State Fuzzing of TLS Implementations. In *Proceedings of the USENIX Conference on Security Symposium*, pages 193–206, 2015.

[61] Antoine Delignat-Lavaud, Martín Abadi, Andrew Birrell, Ilya Mironov, Ted Wobber, and Yinglian Xie. Web PKI: Closing the Gap between Guidelines and Practices. In *Proceedings of the Network and Distributed System Security Symposium*, 2014.

[62] Michael Dietz, Alexei Czeskis, Dirk Balfanz, and Dan S. Wallach. Origin-Bound Certificates: A Fresh Approach to Strong Client Authentication for the Web. In *Proceedings of the USENIX Conference on Security Symposium*, pages 317–331, 2012.

[63] DigiCert, Inc. Symantec SSL/TLS Certificates. `https://www.websecurity.symantec.com/ssl-certificate`.

[64] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The Second Generation Onion Router. In *Proceedings of the USENIX Conference on Security Symposium*, pages 21–21, 2004.

[65] Docjar. HostnameChecker. `http://www.docjar.com/docs/api/sun/security/util/HostnameChecker.html`.

[66] Thai Duong and Juliano Rizzo. Here Come The ⊕ Ninjas. 2011.

[67] Thai Duong and Juliano Rizzo. The CRIME Attack. 2012.

[68] Zakir Durumeric, James Kasten, Michael Bailey, and J. Alex Halderman. Analysis of the HTTPS Certificate Ecosystem. In *Proceedings of the ACM SIGCOMM Conference on Internet Measurement*, pages 291–304, 2013.

[69] Kit Eaton. How One Second Could Cost Amazon $1.6 Billion In Sales – Fast Company. `https://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales`, March 2012.

[70] Peter Eckersley and Jesse Burns. An Observatory for the SSL Universe. 2010.

[71] EFF. Encrypting the Web. `https://www.eff.org/encrypt-the-web`.

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

[73] EFF. HTTPS Everywhere Rulesets, Mixed Content Blocking (MCB). `https://www.eff.org/https-everywhere/rulesets#mixed-content-blocking-mcb`.

[74] EFF. Ruleset Coverage Requirements. `https://github.com/EFForg/https-everywhere/blob/master/ruleset-testing.md`.

[75] EFF. NSA Turns Cookies (And More) Into Surveillance Beacons. `https://www.eff.org/deeplinks/2013/12/nsa-turns-cookies-and-more-surveillance-beacons`, December 2013.

[76] Let's Encrypt. `https://letsencrypt.org/`.

[77] Steven Englehardt, Dillon Reisman, Christian Eubank, Peter Zimmerman, Jonathan Mayer, Arvind Narayanan, and Edward W. Felten. Cookies That Give You Away: The Surveillance Implications of Web Tracking. In *Proceedings of the International Conference on World Wide Web*, pages 289–299, 2015.

[78] Facebook Engineering. Secure browsing by default. `https://www.facebook.com/notes/facebook-engineering/secure-browsing-by-default/10151590414803920/`.

[79] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 50–61, 2012.

[80] Adrienne Porter Felt, Richard Barnes, April King, Chris Palmer, Chris Bentzel, and Parisa Tabriz. Measuring HTTPS Adoption on the Web. In *Proceedings of the USENIX Conference on Security Symposium*, pages 1323–1338, 2017.

[81] FreeRADIUS. `http://freeradius.org/`.

[82] Susumu Fujiwara, G. v. Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. Test Selection Based on Finite State Models. *IEEE Transactions on Software Engineering*, 17(6):591–603, 1991.

[83] Brian Fung. What to expect now that Internet providers can collect and sell your Web browser history – The Washington Post. `https://www.washingtonpost.com/news/the-switch/wp/2017/03/29/what-to-expect-now-that-internet-providers-can-collect-and-sell-your-web-browser-history`, March 2017.

[84] Ryan Gallagher. From Radio to Porn, British Spies Track Web Users' Online Identities – The Intercept. `https://theintercept.com/2015/09/25/gchq-radio-porn-spies-track-web-users-online-identities/`, September 2015.

[85] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The Most Dangerous Code in the World: Validating SSL Certificates in Non-browser Software. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 38–49, 2012.

[86] Phillipa Gill, Vijay Erramilli, Augustin Chaintreau, Balachander Krishnamurthy, Konstantina Papagiannaki, and Pablo Rodriguez. Follow the Money: Understanding Economics of Online Aggregation and Advertising. In *Proceedings of the ACM SIGCOMM Conference on Internet Measurement*, pages 141–148, 2013.

[87] Jo Glanville. Readers' privacy is under threat in the digital age – The Guardian. https://www.theguardian.com/books/2012/aug/31/readers-privacy-under-threat, August 2012.

[88] GNU Compilers. Gcov - Using the GNU Compiler Collection (GCC). https://gcc.gnu.org/onlinedocs/gcc-4.8.1/gcc/Gcov.html.

[89] GnuTLS. Do not check against textual IP addresses in DNSname SAN field. https://gitlab.com/gnutls/gnutls/issues/187.

[90] GnuTLS. gnutls_x509_crt_check_hostname2 has no way to prevent IPs from being matched. https://gitlab.com/gnutls/gnutls/issues/185.

[91] GnuTLS. Improve IP name constraints support. https://gitlab.com/gnutls/gnutls/merge_requests/314.

[92] GnuTLS. X509 certificate API. https://www.gnutls.org/manual/html_node/X509-certificate-API.html#index-gnutls_005fx509_005fcrt_005fcheck_005fhostname2.

[93] Google. Advanced Protection Program. https://landing.google.com/advancedprotection/.

[94] Google. HTTPS Encryption on the Web. https://transparencyreport.google.com/https/overview, 2017. (Accessed on 02/2018).

[95] Google Bughunter University. Lack of HSTS (HTTP Strict Transport Security). https://sites.google.com/site/bughunteruniversity/nonvuln/lack-of-hsts.

[96] Google Code Archive. droidsheep. https://code.google.com/archive/p/droidsheep/.

[97] Google Webmaster Central Blog. HTTPS as a ranking signal. https://webmasters.googleblog.com/2014/08/https-as-ranking-signal.html, August 2014.

[98] Ilya Grigorik. *High Performance Browser Networking*. O'Reilly, 2013.

[99] Ralph Gross and Alessandro Acquisti. Information Revelation and Privacy in Online Social Networks. In *Proceedings of the ACM Workshop on Privacy in the Electronic Society*, pages 71–80, 2005.

[100] Aniko Hannak, Piotr Sapiezynski, Arash Molavi Kakhki, Balachander Krishnamurthy, David Lazer, Alan Mislove, and Christo Wilson. Measuring Personalization of Web Search. In *Proceedings of the International Conference on World Wide Web*, pages 527–538, 2013.

[101] Boyuan He, Vaibhav Rastogi, Yinzhi Cao, Yan Chen, VN Venkatakrishnan, Runqing Yang, and Zhenrui Zhang. Vetting SSL Usage in Applications with SSLINT. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 519–534, 2015.

[102] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices. In *Proceedings of the USENIX Conference on Security Symposium*, pages 205–220, 2012.

[103] HSTS Preload. `https://hstspreload.org/`.

[104] HttpClient. RFC violations in hostname checking. `https://issues.apache.org/jira/browse/HTTPCLIENT-1802`.

[105] HTTPS by Default. Firefox Add-ons. `https://addons.mozilla.org/firefox/addon/https-by-default/`.

[106] HTTPS Everywhere. Regexps matching host that is not in targets. `https://github.com/EFForg/https-everywhere/issues/12297`.

[107] Markus Huber, Martin Mulazzani, and Edgar Weippl. Tor HTTP Usage and Information Leakage. In *Proceedings of the IFIP International Conference on Communications and Multimedia Security*, pages 245–255, 2010.

[108] Ilya Grigorik. TLS has exactly one performance problem: it is not used widely enough. Everything else can be optimized. `https://istlsfastyet.com/`.

[109] Collin Jackson and Adam Barth. ForceHTTPS: Protecting High-security Web Sites from Network Attacks. In *Proceedings of the International Conference on World Wide Web*, pages 525–534, 2008.

[110] Jacob Hoffman-Andrews. Forward Secrecy at Twitter. `https://blog.twitter.com/engineering/en_us/a/2013/forward-secrecy-at-twitter.html`, November 2013.

[111] Dan Kaminsky, Meredith L. Patterson, and Len Sassaman. PKI Layer Cake: New Collision Attacks Against the Global x.509 Infrastructure. In *Proceedings of the International Conference on Financial Cryptography and Data Security*, pages 289–303, 2010.

[112] Alexandros Kapravelos, Chris Grier, Neha Chachra, Christopher Kruegel, Giovanni Vigna, and Vern Paxson. Hulk: Eliciting Malicious Behavior in Browser Extensions. In *Proceedings of the USENIX Conference on Security Symposium*, pages 641–654, 2014.

[113] KB SSL Enforcer. Chrome Web Store. `https://chrome.google.com/webstore/detail/kb-ssl-enforcer/flcpelgcagfhfoegekianiofphddckof`.

[114] Michael J. Kearns and Umesh Virkumar Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, 1994.

[115] Georgios Kontaxis and Angelos D. Keromytis. Protecting Insecure Communications with Topology-aware Network Tunnels. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 1280–1291, 2016.

[116] Dexter Kozen. Lower Bounds for Natural Proof Systems. In *Proceedings of the Annual Symposium on Foundations of Computer Science*, pages 254–266, 1977.

[117] Michael Kranch and Joseph Bonneau. Upgrading HTTPS in Mid-Air: An Empirical Study of Strict Transport Security and Key Pinning. In *Proceedings of the Network and Distributed System Security Symposium*, 2015.

[118] Adam Langley. Revocation checking and Chrome's CRL – ImperialViolet. `https://www.imperialviolet.org/2012/02/05/crlsets.html`, February 2012.

[119] Adam Langley. Apple's SSL/TLS Bug – ImperialViolet. `https://www.imperialviolet.org/2014/02/22/applebug.html`, February 2014.

[120] Mathias Lécuyer, Guillaume Ducoffe, Francis Lan, Andrei Papancea, Theofilos Petsios, Riley Spahn, Augustin Chaintreau, and Roxana Geambasu. XRay: Enhancing the Web's Transparency with Differential Correlation. In *Proceedings of the USENIX Conference on Security Symposium*, pages 49–64, 2014.

[121] Arjen Lenstra, James P Hughes, Maxime Augier, Joppe Willem Bos, Thorsten Kleinjung, and Christophe Wachter. Ron was wrong, Whit is right. Technical report, IACR, 2012.

[122] Let's Encrypt. Percentage of Web Pages Loaded by Firefox Using HTTPS. `https://letsencrypt.org/stats/`, 2017. (Accessed on 02/2018).

[123] Zi Lin. TLS Session Resumption: Full-speed and Secure – Cloudflare Blog. `https://blog.cloudflare.com/tls-session-resumption-full-speed-and-secure/`, February 2015.

[124] Yabing Liu, Han Hee Song, Ignacio Bermudez, Alan Mislove, Mario Baldi, and Alok Tongaonkar. Identifying Personal Information in Internet Traffic. In *Proceedings of the ACM Conference on Online Social Networks*, pages 59–70, 2015.

[125] LLVM Compiler Infrastructure. libFuzzer - A Library for Coverage-guided Fuzz Testing. `http://llvm.org/docs/LibFuzzer.html`.

[126] Moxie Marlinspike. New Trick For Defeating SSL in Practice (SSLStrip). *Black Hat USA*, 2009. `https://www.blackhat.com/presentations/bh-dc-09/Marlinspike/BlackHat-DC-09-Marlinspike-Defeating-SSL.pdf`.

[127] Vicent Martí. Yummy cookies across domains – The Github Blog. `https://blog.github.com/2013-04-09-yummy-cookies-across-domains/`, April 2013.

[128] MatrixSSL. MatrixSSL Release Notes Changes in 3.9.0. `https://github.com/matrixssl/matrixssl/blob/master/doc/CHANGES.md`.

[129] Jonathan R. Mayer and John C. Mitchell. Third-Party Web Tracking: Policy and Technology. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 413–427, 2012.

[130] Jonathan R Mayer and John C Mitchell. Third-party Web Tracking: Policy and Technology. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 413–427, 2012.

[131] Ben McIlwain. Broadening HSTS to secure more of the Web. `https://security.googleblog.com/2017/09/broadening-hsts-to-secure-more-of-web.html`, September 2017.

[132] MDN Web Docs. Mixed content. `https://developer.mozilla.org/en-US/docs/Web/Security/Mixed_content`.

[133] MDN Web Docs. Strict-Transport-Security. `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Strict-Transport-Security`.

[134] Joseph Medley. Remove support for commonName matching in certificates – Google Developers. `https://developers.google.com/web/updates/2017/03/chrome-58-deprecations#remove_support_for_commonname_matching_in_certificates`, March 2017.

[135] Mocana Corporation. The Dangers of Using OpenSSL for Secure IoT. `https://www.infineon.com/dgdl/Infineon-The+Dangers+of+Using+OpenSSL+for+Secure+IoT-ABR-v03_17-EN.pdf?fileId=5546d4625b10283a015b1e96f2b100e3`, 2017.

[136] Mozilla. Public Suffix List. `https://publicsuffix.org`.

[137] Mozilla. Web Security Guideline. `https://infosec.mozilla.org/guidelines/web_security`.

[138] Mozilla Security Blog. Firefox Preloading HSTS. `https://blog.mozilla.org/security/2012/11/01/preloading-hsts/`.

[139] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This POODLE Bites: Exploiting The SSL 3.0 Fallback. `https://www.openssl.org/~bodo/ssl-poodle.pdf`, September 2014.

[140] David Naylor, Alessandro Finamore, Ilias Leontiadis, Yan Grunenberger, Marco Mellia, Maurizio Munafò, Konstantina Papagiannaki, and Peter Steenkiste. The Cost of the "S" in HTTPS. In *Proceedings of the ACM International on Conference on Emerging Networking Experiments and Technologies*, pages 133–140, 2014.

[141] André Niemann and Jacqueline Brendel. A Survey on CA Compromises.

[142] Node.js. `https://nodejs.org/`.

[143] Office of Oversight and Investigations. A Review of the Data Broker Industry: Collection, Use, and Sale of Consumer Data for Marketing Purposes. `https://www.commerce.senate.gov/public/_cache/files/0d2b3642-6221-4888-a631-08f2f255b577/AE5D72CBE7F44F5BFC846BECE22C875B.12.18.13-senate-commerce-committee-report-on-data-broker-industry.pdf`, December 2013.

[144] Lucky Onwuzurike and Emiliano De Cristofaro. Danger Is My Middle Name: Experimenting with SSL Vulnerabilities in Android Apps. In *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks*, page 15, 2015.

[145] OpenSIPS. `https://github.com/OpenSIPS/opensips`.

[146] Oracle. Java Cryptography Architecture Oracle Providers Documentation. `https://docs.oracle.com/javase/7/docs/technotes/guides/security/SunProviders.html`.

[147] Oracle. Java Native Interface (JNI). `https://docs.oracle.com/javase/8/docs/technotes/guides/jni/`.

[148] Oracle. Oracle Critical Patch Update Advisory - July 2017. `https://www.oracle.com/technetwork/topics/security/cpujul2017-3236622.html`, July 2017.

[149] Chris Palmer. Intent to Deprecate and Remove: Public Key Pinning. `https://groups.google.com/a/chromium.org/forum/#!msg/blink-dev/he9tr7p3rZ8/eNMwKPmUBAAJ`, October 2017.

[150] Arnis Parsovs. Practical Issues with TLS Client Certificate Authentication. In *Proceedings of the Network and Distributed System Security Symposium*, 2014.

[151] Nicole Perlroth, Jeff Larson, and Scott Shane. NSA Able to Foil Basic Safeguards of Privacy on Web – The New York Times. `http://www.nytimes.com/2013/09/06/us/nsa-foils-much-internet-encryption.html`, September 2013.

[152] Mike Perry. HTTPS Everywhere Firefox addon helps you encrypt web traffic – Tor Blog. `https://blog.torproject.org/https-everywhere-firefox-addon-helps-you-encrypt-web-traffic`, June 2010.

[153] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D Keromytis, and Suman Jana. NEZHA: Efficient Domain-Independent Differential Testing. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 615–632, 2017.

[154] Iasonas Polakis, George Argyros, Theofilos Petsios, Suphannee Sivakorn, and Angelos D Keromytis. Where's Wally?: Precise User Discovery Attacks in Location Proximity Services. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 817–828, 2015.

[155] Proxytunnel. `http://proxytunnel.sf.net`.

[156] Harald Raffelt, Bernhard Steffen, and Therese Berg. LearnLib: A Library for Automata Learning and Experimentation. In *Proceedings of the International Workshop on Formal Methods for Industrial Critical Systems*, pages 62–71, 2005.

[157] Franziska Roesner, Tadayoshi Kohno, and David Wetherall. Detecting and Defending Against Third-Party Tracking on the Web. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation*, pages 12–12, 2012.

[158] Jose Selvi. Bypassing HTTP Strict Transport Security. *Black Hat Europe*, 2014.

[159] Michael Sipser. *Introduction to the Theory of Computation*. Thomson Course Technology Boston, 2006.

[160] Suphannee Sivakorn, George Argyros, Kexin Pei, Angelos D. Keromytis, and Suman Jana. HVLearn: Automated Black-box Analysis of Hostname Verification in SSL/TLS Implementations. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 521–538, 2017.

[161] Suphannee Sivakorn, Iasonas Polakis, and Angelos D. Keromytis. I Am Robot: (Deep) Learning to Break Semantic Image CAPTCHAs. In *Proceedings of the IEEE European Symposium on Security and Privacy*, pages 388–403, 2016.

[162] Suphannee Sivakorn, Iasonas Polakis, and Angelos D. Keromytis. The Cracked Cookie Jar: HTTP Cookie Hijacking and the Exposure of Private Information. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 724–742, 2016.

[163] Suphannee Sivakorn, Jason Polakis, and Angelos D. Keromytis. That's the Way the Cookie Crumbles: Evaluating HTTPS Enforcing Mechanisms. In *Proceedings of the ACM Workshop on Privacy in the Electronic Society*, pages 71–81, 2016.

[164] SLOCCount. `https://www.dwheeler.com/sloccount/`.

[165] Smart HTTPS. Firefox Add-ons. `https://addons.mozilla.org/firefox/addon/smart-https/`.

[166] Christopher Soghoian and Sid Stamm. Certified Lies: Detecting and Defeating Government Interception Attacks Against SSL. In *Proceedings of the International Conference on Financial Cryptography and Data Security*, pages 250–259, 2011.

[167] John Solomon. SSL Certificate Options with Features and Costs? – Chargebee's SaaS Dispatch. `https://www.chargebee.com/blog/options-ssl-certificate-cost/`, May 2017.

[168] Ashkan Soltani, Andrea Peterson, and Barton Gellman. NSA Uses Google Cookies to Pinpoint Targets for Hacking – The Washington Post.

https://www.washingtonpost.com/news/the-switch/wp/2013/12/10/nsa-uses-google, December 2013.

[169] Juraj Somorovsky. Systematic Fuzzing and Testing of TLS Libraries. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 1492–1504, 2016.

[170] Andreas Sotirakopoulos, Kirstie Hawkey, and Konstantin Beznosov. On the Challenges in Usable Security Lab Studies: Lessons Learned from Replicating a Study on SSL Warnings. In *Proceedings of Symposium on Usable Privacy and Security*, page 3, 2011.

[171] David Sounthiraraj, Justin Sahs, Garret Greenwood, Zhiqiang Lin, and Latifur Khan. Smv-hunter: Large Scale, Automated Detection of SSL/TLS Man-in-the-middle Vulnerabilities in Android Apps. In *Proceedings of the Network and Distributed System Security Symposium*, 2014.

[172] STATOPERATOR. HTTPS usage statistics on top 1M websites. https://statoperator.com/research/https-usage-statistics-on-top-websites/. (Accessed on 02/2018).

[173] Marc Stevens, Alexander Sotirov, Jacob Appelbaum, Arjen Lenstra, David Molnar, Dag Arne Osvik, and Benne De Weger. Short Chosen-prefix Collisions for MD5 and the Creation of A Rogue CA Certificate. In *Advances in Cryptology-CRYPTO*, pages 55–69, 2009.

[174] Joshua Sunshine, Serge Egelman, Hazim Almuhimedi, Neha Atri, and Lorrie Faith Cranor. Crying Wolf: An Empirical Study of SSL Warning Effectiveness. In *Proceedings of the USENIX Conference on Security Symposium*, pages 399–416, 2009.

[175] TechCrunch. Firesheep In Wolves' Clothing: Extension Lets You Hack Into Twitter, Facebook Accounts Easily – TechCrunch. https://techcrunch.com/2010/10/24/firesheep-in-wolves-clothing-app-lets-you-hack-into-twitter-facebook-accounts-easily/, October 2010.

[176] Telex Anticensorship. https://github.com/ewust/telex.

[177] The Heartbleed Bug. http://heartbleed.com/.

[178] Tim Taubert. BOTCHING FORWARD SECRECY: The sad state of server-side TLS Session Resumption implementations. https://timtaubert.de/blog/2014/11/the-sad-state-of-server-side-tls-session-resumption-implementations/, November 2014.

[179] Tor. Disable mixed content rulesets on FF 23+. https://trac.torproject.org/projects/tor/ticket/8774, April 2013.

[180] Janice Y. Tsai, Serge Egelman, Lorrie Cranor, and Alessandro Acquisti. The Effect of Online Privacy Information on Purchasing Behavior: An Experimental Study. *Information Systems Research*, 22(2):254–268, 2011.

[181] Jo-el van Bergan. What Is Mixed Content? – Google Developers. https://developers.google.com/web/fundamentals/security/prevent-mixed-content/.

[182] Nevena Vratonjic, Julien Freudiger, Vincent Bindschaedler, and Jean-Pierre Hubaux. The Inconvenient Truth about Web Certificates. In *Economics of Information Security and Privacy III*, pages 79–117, 2013.

[183] W3C. Content Security Policy. https://www.w3.org/TR/CSP/.

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

[185] W3C. Mixed Content. https://www.w3.org/TR/mixed-content/.

[186] W3C. Upgrade Insecure Requests. https://www.w3.org/TR/upgrade-insecure-requests/.

[187] W3C. Web Application Security Working Group. https://github.com/w3c/webappsec.

[188] David Wagner and Bruce Schneier. Analysis of the SSL 3.0 Protocol. In *Proceedings of the USENIX Workshop on Electronic Commerce*, pages 4–4, 1996.

[189] Colin Walls. *Embedded Software: the Works*. Elsevier, 2012.

[190] John Walsh. Free Can Make You Bleed. `http://blog.ssh.com/free-can-make-you-bleed`, April 2014.

[191] Randy Westergren. Widespread XSS Vulnerabilities in Ad Network Code Affecting Top Tier Publishers. `http://randywestergren.com/widespread-xss`, March 2016.

[192] David A. Wheeler. Preventing Heartbleed. *Computer*, 47(8):80–83, 2014.

[193] Philipp Winter, Richard Köwer, Martin Mulazzani, Markus Huber, Sebastian Schrittwieser, Stefan Lindskog, and Edgar Weippl. Spoiled Onions: Exposing Malicious Tor Exit Relays. In *Proceedings of the International Conference on Privacy Enhancing Technologies*, pages 304–331, 2014.

[194] WordPress Support. HTTPS and SSL. `https://en.support.wordpress.com/https-ssl/`.

[195] Xingyu Xing, Wei Meng, Dan Doozan, Alex C. Snoeren, Nick Feamster, and Wenke Lee. Take This Personally: Pollution Attacks on Personalized Services. In *Proceedings of the USENIX Conference on Security Symposium*, pages 671–686, 2013.

[196] Bryant Zadegan and Ryan Lester. Abusing Bleeding Edge Web Standards for AppSec Glory. *Black Hat USA*, 2016.

[197] Michal Zalewski. *The Tangled Web: A Guide to Securing Modern Web Applications*. No Starch Press, 2011.

[198] Xiaofeng Zheng, Jian Jiang, Jinjin Liang, Haixin Duan, Shuo Chen, Tao Wan, and Nicholas Weaver. Cookies Lack Integrity: Real-World Implications. In *Proceedings of the USENIX Conference on Security Symposium*, pages 707–721, 2015.

Appendix A

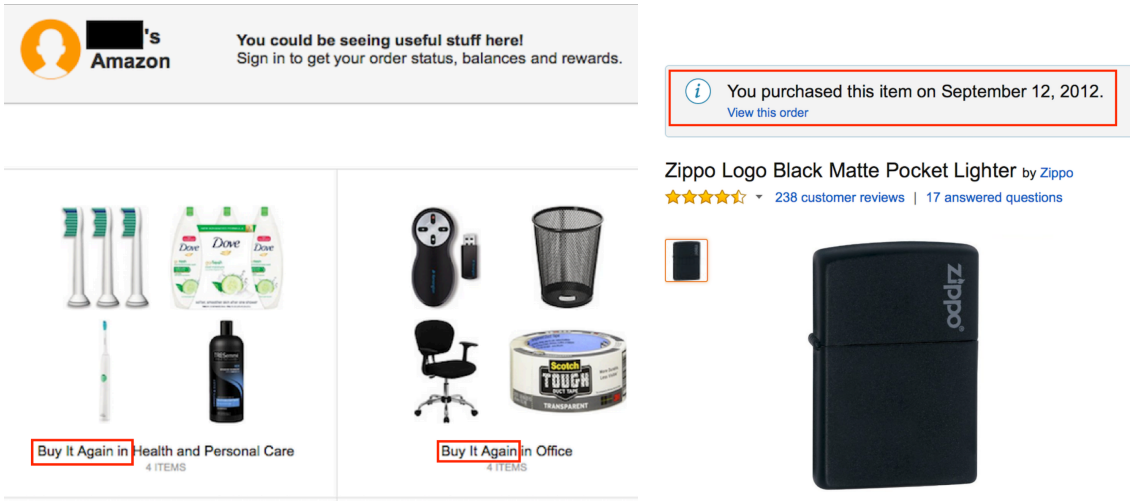
Exposure of Privacy Information on Real-world Services

A.1 Additional Real-world Privacy Leakages

A.1.1 E-commerce Websites

A.1.1.1 Amazon

The adversary can obtain information regarding previously purchased items either through the recommendation page (Figure A.1(a)) or through product pages (Figure A.1(b)). The iOS versions of the Amazon app also expose information about the user's mobile device in the HTTP request header, as shown in Listing A.1.



(a) Recommendations Page (b) Product Page

Figure A.1: Obtaining information about previously purchased items from user's Amazon account.

```
"User–Agent": "Mozilla/5.0 (iPhone; CPU iPhone OS 9_0_2 like Mac OS X)
    AppleWebKit/601.1.46 (KHTML, like Gecko) Mobile/13A452",
"cookie_name": "amzn–app–ctxt",
    "cookie_value": "1.4%20 {
    "os":"iOS"
    "ov":"9.0.2"
    "an":"Amazon"
    "av":"5.3.0"
    "dm":{"w":"640" "h":"960" "ld":"2.000000"}
    "uiv":5
    "nal":1
    "cp":8XXXXX
    "xv":"1.11"
    "di":{
        "ca":"AT&T"
            "ct":"Wifi"
            "mf":"Apple"
            "pr":"iPhone"
            "md":"iPhone"
            "v":"4S"
            "dti":"A287XXXXXXXXXX"
    }}"
```

Listing A.1: User device and network information disclosed in the value attribute of Amazon's mobile app HTTP request header.

A.1.1.2 Ebay

Apart from the login and checkout pages, the remaining Ebay website runs over HTTP. As a result, the stolen HTTP cookie gives the adversary access to both personal information and account functionality.

Personal Information. The site always reveals the user's first name and delivery address. Also, depending on what the victim uses for logging in (username or email address) is also exposed. By forging a cookie with the same value but a different scope (domain and path), we are also able to obtain the user's delivery address. The HTTP cookies can also access the user's messages, which are normally served over HTTPS.

History. The cookie provides access to the functionality that exposes the victim's purchase history, and also allows us to view and edit the items in the victim's watch and wish-lists. We can also see which items have been bought or bid upon in the past, and all the items being sold by the. victim.

Cart. Similarly to the other e-commerce websites we tested, the HTTP cookie enables access to the cart for viewing items already in it, or adding/removing items.

A.1.1.3 Walmart

If the adversary appends the stolen cookies when connecting, the website will reveal the user's first name, postcode, and also allow editing of the cart. However, upon inspection, we found that the `customer` HTTP cookie actually contains 34 fields of information about the user within its `value` attribute. Apart from the subset that can be seen in Listing A.2, which includes the user's first and last name and email address, the cookie also contains ID information that points to the user's reviews and comments, and a tracking ID for third parties.

Target. As with most e-commerce sites, the stolen cookie reveals the user's first name, email address, and the ability to view and edit the cart, and the user's wish-list. Furthermore, it also reveals items recently viewed by the user.

Vendor-assisted attacks. The cookie exposes functionality that can be leveraged for deploying spam, similarly to Amazon. The attacker can either add items to the cart and send an email about those items (sent by `orders@service.target.com`), or create and send a wish-list (sent by `noreply@service.target.com`). In both cases, the emails explicitly contain the user's full name (thus, making the last name obtainable to the attacker). While the attacker cannot include any text, which would facilitate deploying spam or phishing campaigns, one could promote arbitrary items. The attacker can also deploy the two aforementioned extortion scams.


```
{   "domain": ".walmart.com",
    "name": "customer",
    "path": "/",
    "secure": false,
    "httpOnly": false,
    "value": "{"firstName":"JANE", "lastName":"DOE", "emailAddress":"
        janedoe@example.com", "isMigrated":true, "omsCustomerId":"XXXXXXXX", "
        ReviewUser":{"isValid":true,  "AdditionalFieldsOrder":"XXXXXXXX", "
        Avatar":{}, "UserNickname":"XXXXXXXXX", "Photos":[], "
        ContextDataValues":"XXXXXXXX","Videos":[], "ContextDataValuesOrder":"
        XXXXXXXX", "SubmissionId":"XXXXX", "ContributorRank":"XXXX","StoryIds
        ":"XXXXXXXX","AnswerIds":"XXXXXX","QuestionIds":"XXXXXXX","BadgesOrder
        ":"XXXXXX", "Badges":"XXXXXXXX", "Location":"XXXXX", "
        SecondaryRatingsOrder":"XXXXXX", "ProductRecommendationIds":"XXXXXXXX
        ", "AdditionalFields":{}, "SubmissionTime":"20XX-XX-XXXXXXXXXXXX", "
        ModerationStatus":"APPROVED", "ReviewIds":"XXXXXXX", "ThirdPartyIds":"
        XXXXXXXX", "Id":"ff79XXXXXXXXXXXXX509dc5", "CommentIds":[], "
        SecondaryRatings":"XXXXX", "LastModeratedTime":"20XXXXXXXXXXXXXXX", "
        reviewStatus":{"hasReview":"XXXXX"}}}"
}
```


Listing A.2: User information disclosed in the value attribute of Walmart's HTTP customer cookie (values have been changed for privacy).

A.1.2 News Media

A.1.2.1 CNN

Almost the entire website runs over HTTP, including the login page, which can be exploited by active adversaries to modify or inject content. The credentials, however, are sent over HTTPS, preventing eavesdroppers from hijacking the user's session. Nonetheless, the HTTP cookie allows the attacker to view and edit the user's profile, which includes first and last name, postal address, email and phone number, profile picture and link to the user's

Facebook account. Furthermore, the attacker can write or delete article comments, and also obtain the recently viewed or created reports on iReport, CNN's citizen journalism portal[1].

A.1.2.2 New York Times

The HTTP cookie allows the adversary to obtain or change the user's profile photo, name and last name, a link pointing to a personal homepage, and a short personal description (bio). The adversary can also obtain and edit the list of articles that the user has saved.

A.1.2.3 The Guardian

Stolen HTTP cookies provide access to the user's public profile sections, which includes a profile picture and username, a short bio, the user's interests, and previous comments on articles. The adversary can also post comments as the user.

A.1.2.4 Huffington Post

Similar to CNN, almost the entire website runs overs unencrypted connections, and the HTTP cookie allows read and edit access to the user's profile, article subscriptions, comments, fans, and followings. The profile includes the user's login name, profile photo, email address, biography, postal code, city, and state. The attacker can also change the user's password, or delete the account.

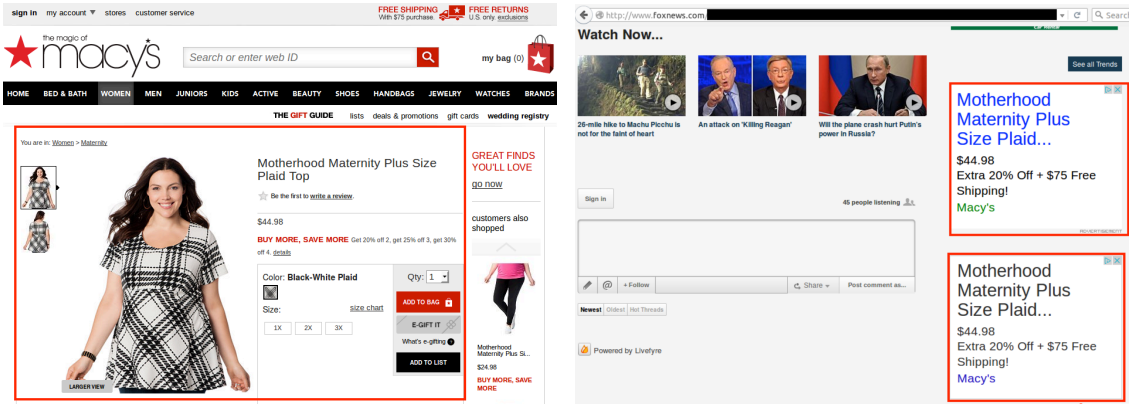
A.1.3 Ad Networks

Figure A.2 contains screenshots of our experiment that demonstrates how ad networks can reveal parts of a user's browsing history as described in Ads Network Section of Chapter 3 (Section 3.5.1).

A.2 Alternative Browser Extensions

KB SSL Enforcer. KB SSL Enforcer [113] is a Chrome browser extension that automatically detects the availability of HTTPS for a domain prior to upgrading to a secure

[1] `http://ireport.cnn.com/`



(a) Browsed page (b) Attacker receives ads exposing browsed page.

Figure A.2: Side-channel leak of user's browsing history by the Doubleclick ad network.

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.

Smart HTTPS. Smart HTTPS [165] 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.

HTTPS by Default. HTTPS by Default [105] 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.

Appendix B

TLS Hostname Verification

B.1 Details of Tested Hostname Verification Implementations

OpenSSL. has separate checking functions for each type identifiers as shown in Table 5.1. In our testing, we use the default setup that supports matching wildcards. OpenSSL also provides support for applications to turn some of these hostname verification functions on or off by calling different setup functions (e.g., `X509_VERIFY_PARAM_set1_host` and `X509_VERIFY_PARAM_set1_email`).

GnuTLS. The GnuTLS check hostname function is designed for certificate verification for HTTPS supporting domain names, IPv4, and IPv6. Like OpenSSL, GnuTLS also provides the application to select whether to verify hostname with wildcard or not. By default, GnuTLS wildcard matching is enabled. We use the default setting for our experiments.

MbedTLS. The hostname verification functions in MbedTLS only supports checking for domain name verification.

MatrixSSL. A single function `matrixValidateCerts` is responsible for checking all different types of identifiers (e.g., DNS, IPv4, and email). The library does not include support for IPv6 yet. MatrixSSL also provides a separate function, `psX509ValidateGeneralName`

that should be used before calling matrixValidateCerts for name checking for filtering out invalid input.

JSSE (Java Secure Socket Extension). SunJSSE [146], as part of the JSSE release, has internal built-in hostname checking support (sun.security.util.HostnameChecker [65]). It supports domain name, IPv4, and IPv6 verification through the HostnameChecker.match interface.

CPython SSL. CPython is the oldest and one of the most popular Python VM implementation. CPython's inbuilt SSL support depends on the OpenSSL library, but does not use OpenSSL's hostname verification function. Instead, it includes its own hostname verification implementation, `match_hostname` function. Currently, it only supports domain name and IP address verification but does not support email verification.

HttpClient. (Apache HttpClient) is used extensively in Web-services middleware such as Apache Axis 2 It supports IPv4, IPv6, and domain name verification [28]. By default the library provides a verify function in `DefaultHostnameVerifier` to perform the identity verification. The verifier can also be used with `PublicSuffixMatcher` object to perform additional checks.

cURL. By default, it uses OpenSSL [55] but implements its own hostname verification function `verifyhost` that supports domain name, IPv4, and IPv6 verification.

B.2 Detailed List of Discrepancies

In Table B.1, we present a detailed list of the discrepancies discovered between various TLS libraries and applications.

Table B.1: Sample strings accepted by the automata inferred from different hostname verification implementations

Test Certificate Identifier Template	OpenSSL	GnuTLS	MbedTLS	MatrixSSL	JSSE	CPython SSL	HttpClient	cURL
Wildcard in Certificate								
*.aaa.aaa	a.aaa.aaa .aaa.aaa *.aaa.aaa .aaa a.aaa.aaa\0 .aaa.aaa\0 .aaa\0 *.aaa.aaa\0	.aaa.aaa .aaa.aaa\0	a.aaa.aaa a.aaa.aaa\0	a.aaa.aaa a.aaa.aaa\0	a.aaa.aaa	a.aaa.aaa	.aaa.aaa	a.aaa.aaa a.aaa.aaa.\0 a.aaa.aaa\0 a.aaa.aaa.
aaa.*.aaa	aaa.*.aaa .aaa .*.aaa aaa.*.aaa\0 .aaa\0 .*.aaa\0	aaa.*.aaa aaa.*.aaa\0	aaa.*.aaa aaa.*.aaa\0	None	aaa.a.aaa	aaa.*.aaa	aaa..aaa	aaa.*.aaa aaa.*.aaa.\0 aaa.*.aaa\0 aaa.*.aaa.
a*.aaa.aaa	aa.aaa.aaa a.aaa.aaa a*.aaa.aaa .aaa.aaa .aaa aa.aaa.aaa\0 a.aaa.aaa\0 a*.aaa.aaa\0 .aaa.aaa\0 .aaa\0	a*.aaa.aaa a*.aaa.aaa\0	a*.aaa.aaa a*.aaa.aaa\0	None	a.aaa.aaa	a.aaa.aaa	a.aaa.aaa	aa.aaa.aaa aa.aaa.aaa.\0 aa.aaa.aaa\0 aa.aaa.aaa.
aaa.a*.aaa	aaa.a*.aaa .aaa .a*.aaa aaa.a*.aaa\0 .aaa\0 .a*.aaa\0	aaa.a*.aaa aaa.a*.aaa\0	aaa.a*.aaa aaa.a*.aaa\0	None	aaa.a.aaa	aaa.a*.aaa	aaa.a.aaa	aaa.a*.aaa aaa.a*.aaa.\0 aaa.a*.aaa\0 aaa.a*.aaa.

Test Certificate Identifier Template	OpenSSL	GnuTLS	MbedTLS	MatrixSSL	JSSE	CPython SSL	HttpClient	cURL
Wildcard in Certificate (Continued)								
xn--aaa*.aaa	.aaa .aaa\0	xn--aaa*.aaa xn--aaa*.aaa\0	xn--aaa*.aaa xn--aaa*.aaa\0	None	xn--aaa.aaa	xn--aaa*.aaa	xn--aaa.aaa	xn--aaa*.aaa xn--aaa*.aaa.\0 xn--aaa*.aaa\0 xn--aaa*.aaa.
*.xn--aaa.aaa	a.xn--aaa.aaa .aaa .xn--aaa.aaa *.xn--aaa.aaa a.xn--aaa.aaa\0 .aaa\0 .xn--aaa.aaa\0 *.xn--aaa.aaa\0	.xn--aaa.aaa .xn--aaa.aaa\0	.xn--aaa.aaa .xn--aaa.aaa\0	None	a.xn--aaa.aaa	a.xn--aaa.aaa	.xn--aaa.aaa	a.xn--aaa.aaa a.xn--aaa.aaa.\0 a.xn--aaa.aaa\0 a.xn--aaa.aaa.
xn--aaa.*.aaa	.aaa .*.aaa xn--aaa.*.aaa .aaa\0 .*.aaa\0 xn--aaa.*.aaa\0	xn--aaa.*.aaa xn--aaa.*.aaa\0	xn--aaa.*.aaa xn--aaa.*.aaa\0	None	xn--aaa.a.aaa	xn--aaa.*.aaa	xn--aaa.aaa	xn--aaa.*.aaa xn--aaa.*.aaa.\0 xn--aaa.*.aaa\0 xn--aaa.*.aaa.
Wildcard Unclear Practices								
*.aaa	.aaa *.aaa .aaa\0 *.aaa\0	None	a.aaa a.aaa\0	a.aaa a.aaa\0	a.aaa	a.aaa	.aaa	*.aaa *.aaa.\0 *.aaa\0 *.aaa.
a*b*c*.aaa.aaa	a*b*c*.aaa.aaa .aaa.aaa .aaa a*b*c*.aaa.aaa\0 .aaa.aaa\0 .aaa\0	a*b*c*.aaa.aaa a*b*c*.aaa.aaa\0	a*b*c*.aaa.aaa a*b*c*.aaa.aaa\0	None	abc.aaa.aaa	None	ab*c*.aaa.aaa	aab*c*.aaa.aaa aab*c*.aaa.aaa.\0 aab*c*.aaa.aaa\0 aab*c*.aaa.aaa.

Test Certificate Identifier Template	OpenSSL	GnuTLS	MbedTLS	MatrixSSL	JSSE	CPython SSL	HttpClient	cURL
Wildcard Unclear Practices (Continued.)								
..aaa.aaa	.aaa.aaa .*.aaa.aaa *.*.aaa.aaa .aaa .aaa.aaa\0 .aaa\0 .*.aaa.aaa\0 *.*.aaa.aaa\0	*.aaa.aaa *.aaa.aaa\0	a.*.aaa.aaa a.*.aaa.aaa\0	None	a.a.aaa.aaa	a.*.aaa.aaa	*.aaa.aaa	a.*.aaa.aaa a.*.aaa.aaa.\0 a.*.aaa.aaa\0 a.*.aaa.aaa.
*b.aaa.aaa	ab.aaa.aaa b.aaa.aaa .aaa.aaa *b.aaa.aaa .aaa ab.aaa.aaa\0 b.aaa.aaa\0 .aaa.aaa\0 .aaa\0 *b.aaa.aaa\0	b.aaa.aaa b.aaa.aaa\0	*b.aaa.aaa *b.aaa.aaa\0	None	ab.aaa.aaa b.aaa.aaa	b.aaa.aaa	b.aaa.aaa	ab.aaa.aaa ab.aaa.aaa.\0 ab.aaa.aaa\0 ab.aaa.aaa.
.aaa.aaa	.aaa.aaa .aaa .aaa.aaa\0 .aaa\0	None	.aaa.aaa .aaa.aaa\0	None	aaa.aaa	.aaa.aaa	.aaa.aaa	.aaa.aaa .aaa.aaa.\0 .aaa.aaa\0 .aaa.aaa.
Email Address								
SAN email: *@aaa.aaa	*@aaa.aaa *@aaa.aaa\0	*@aaa.aaa *@aaa.aaa\0	–	None	–	–	–	–
SAN email: aaa@*	aaa@* aaa@*\0	aaa@* aaa@*\0	–	None	–	–	–	–
SAN email: aaa@*.aaa	aaa@*.aaa aaa@*.aaa\0	aaa@*.aaa aaa@*.aaa\0	–	None	–	–	–	–
SAN email: aaa@aaa.*	aaa@aaa.* aaa@aaa.*\0	aaa@aaa.* aaa@aaa.*\0	–	None	–	–	–	–
SAN email: AAA@aaa.aaa	AAA@aaa.aaa AAA@aaa.aaa\0	AAA@aaa.aaa AAA@aaa.aaa\0	–	aaa@aaa.aaa aaa@aaa.aaa\0	–	–	–	–
SAN email: aaa@AAA.aaa	aaa@aaa.aaa aaa@aaa.aaa\0	aaa@aaa.aaa aaa@aaa.aaa\0	–	aaa@aaa.aaa aaa@aaa.aaa\0	–	–	–	–
IP Address								
SAN IP Addr: *.111.111.111	None	None	–	None	None	None	None	None