

Carnegie Mellon University

CARNEGIE INSTITUTE OF TECHNOLOGY

THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF Doctor of Philosophy

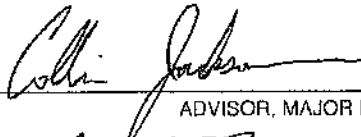
TITLE Protecting Browsers from

Network Intermediaries

PRESENTED BY Lin-Shung Huang

ACCEPTED BY THE DEPARTMENT OF

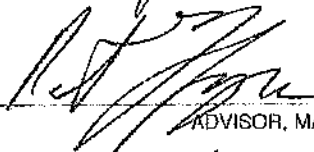
Electrical and Computer Engineering



ADVISOR, MAJOR PROFESSOR

8/15/14

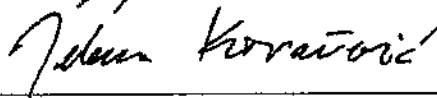
DATE



ADVISOR, MAJOR PROFESSOR

8/15/14

DATE



DEPARTMENT HEAD

9/30/14

DATE

APPROVED BY THE COLLEGE COUNCIL

DEAN

DATE

Protecting Browsers from Network Intermediaries

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Lin-Shung Huang

B.S., Computer Science, National Chiao Tung University
M.Eng., Electrical and Computer Engineering, Cornell University

Carnegie Mellon University
Pittsburgh, PA

September, 2014

Acknowledgements

Foremost, I am profoundly indebted to Collin Jackson, who introduced me to browser security research and provided incredible guidance and support during my time at Carnegie Mellon University. I owe a great debt of gratitude to my thesis committee, Collin Jackson, Patrick Tague, Nicolas Christin, and Dan Boneh. This dissertation greatly benefitted from the comments, suggestions and feedback. I am deeply grateful for my co-authors, Collin Jackson, Dan Boneh, Eric Chen, Adam Barth, Eric Rescorla, Alex Rice, Erling Ellingsen, Emily Stark, Dinesh Israni, Emin Topalovic, Brennan Saeta, and Shrikant Adhikarla, who collaborated with me on the research described in this thesis. I would like to thank my co-advisor Patrick Tague for providing guidance and help on my thesis prospectus and dissertation. I am grateful for my co-advisor Adrian Perrig for providing guidance on my qualifying examination in Pittsburgh. I have learned invaluable knowledge and lessons from interacting with my mentors, Helen Wang, Alex Moshchuk, Alex Rice and Mark Hammell, during my internships at Microsoft Research and Facebook. I appreciate Linda Bickham for coaching my presentation for my thesis prospectus. I would like to thank my family for their unconditional support.

My research has been funded by the U.S. National Science Foundation Team for Research in Ubiquitous Secure Technology (TRUST), and grants from Symantec, VeriSign and Google.

Abstract

Network intermediaries relay traffic between web servers and clients, and are often deployed on the Internet to provide improved performance or security. Unfortunately, network intermediaries can actually do more harm than good. In this thesis, we articulate the dangers of network intermediaries, which motivates the need for pervasive encryption. We further seek to understand the reasons why encryption isn't more widely deployed and fix them.

The existence of network intermediaries makes web security particularly challenging, considering that network intermediaries may operate (1) erroneously, or (2) maliciously. We verified that 7% of Internet users are behind proxies that allow either IP hijacking attacks or cache poisoning attacks, and that 0.2% of encrypted connections on a large global website were intercepted without authorization. While the need for encryption is clear, many websites have not deployed Transport Layer Security (TLS) due to performance concerns. We identified three opportunities to reduce the performance overhead of TLS without sacrificing security: (1) prefetching and prevalidating certificates, (2) using short-lived certificates, and (3) configuring elliptic curve cryptography for forward secrecy.

Contents

Acknowledgements	i
Abstract	ii
1 Introduction	1
1.1 Threat Models	3
1.1.1 Web Attacker	4
1.1.2 Network Attacker	4
1.1.3 Out-of-Scope Threats	5
1.2 Browser Network Access Policies	6
1.2.1 Same-Origin Policy	6
1.2.2 Verified-Origin Policy	8
1.3 Why are Network Intermediaries Dangerous?	10
1.3.1 Transparent Proxy Vulnerabilities	11
1.3.2 Unauthorized Interceptions	11
1.4 Why isn't TLS More Widely Deployed?	12
1.4.1 Background	12
1.4.2 Deployment Concerns	15
1.5 Opportunities for Improving TLS Performance	18

1.5.1	Prefetching and Prevalidating Certificates	19
1.5.2	Short-Lived Certificates	20
1.5.3	Forward Secrecy Performance	21
1.6	Organization	21
2	Transparent Proxy Vulnerabilities	22
2.1	Attacks on Java and Flash Sockets	22
2.1.1	Auger’s IP Hijacking Attack	23
2.1.2	Our Cache Poisoning Attack	25
2.1.3	Ad-based Experiment	27
2.2	Attacks on WebSocket Protocols	31
2.2.1	POST-based Strawman Handshake	31
2.2.2	Upgrade-based Strawman Handshake	33
2.2.3	CONNECT-based Strawman Handshake	34
2.2.4	Ad-based Experiment	35
2.3	Securing the WebSocket Protocol	38
2.3.1	Our Proposal: Payload Masking	38
2.3.2	Performance Evaluation	40
2.3.3	Adoption	42
2.4	Related Work	42
2.4.1	Cross-Protocol Attacks	42
2.4.2	HTTP Cache Poisoning	43
2.4.3	HTTP Response Splitting	43
3	Unauthorized Interceptions	44
3.1	Man-in-the-Middle Attacks	45

3.1.1	How it Works	45
3.1.2	Why is Detection Challenging?	47
3.2	Flash-Based Detection Method	48
3.2.1	Design	48
3.2.2	Implementation	52
3.2.3	Experimental Results	53
3.2.4	Limitations	56
3.3	Analysis of Forged SSL Certificates	58
3.3.1	Certificate Subjects	58
3.3.2	Certificate Issuers	60
3.4	Related Work	67
3.4.1	Webpage Tamper Detection	67
3.4.2	Certificate Observatories	69
3.4.3	Certificate Validation with Notaries	69
3.4.4	HTTP Strict Transport Security	70
3.4.5	Certificate Pinning	70
3.4.6	Certificate Audit Logs	72
3.4.7	DNS-based Authentication	72
4	Prefetching and Prevalidating Certificates	73
4.1	Performance Cost of OCSP	74
4.1.1	Experimental Setup	74
4.1.2	Results	75
4.1.3	Lessons	77
4.2	Server Certificate Prefetching and Prevalidation	79

4.2.1	Benefits of prefetching	79
4.2.2	Prefetching methods	80
4.2.3	Implementation	83
4.3	Performance Evaluation	83
4.3.1	Experimental setup	84
4.3.2	Results	86
4.3.3	Analysis	87
5	Short-Lived Certificates	90
5.1	Deficiencies of Existing Revocation Mechanisms	91
5.1.1	CRL	91
5.1.2	OCSP	93
5.2	An Old Proposal Revisited	94
5.2.1	Design	95
5.2.2	Implementation	100
5.2.3	Deployment Incentives	102
5.3	Analysis of Post-OCSP World	104
5.3.1	Chrome’s CRL	104
5.3.2	Hard-Fail OCSP	105
5.3.3	Short-Lived Certificates	106
5.3.4	Hybrid approach	108
5.4	Related Work	109
5.4.1	OCSP Stapling	109
6	Forward Secrecy Performance	111
6.1	Controlled Experiment - Server throughput	113

6.1.1	TLS server setup	113
6.1.2	Methodology	114
6.1.3	Results	115
6.2	Ad-based experiment - Client latencies	116
6.2.1	Methodology	116
6.2.2	Results	117
6.3	Discussion	120
6.3.1	Forward secrecy is free	120
6.3.2	RSA vs. ECDSA authentication	121
6.4	Related Work	122
7	Conclusion	124
	Bibliography	139

List of Tables

2.1	HTTP Host header spoofing via plug-in sockets	29
2.2	HTTP Host header spoofing via HTML5 WebSocket strawman protocols .	36
3.1	Number of clients that completed each step of the detection procedure . .	54
3.2	Categorization of reports	55
3.3	Subject organizations of forged certificates	59
3.4	Subject common names of forged certificates	60
3.5	Issuer organizations of forged certificates	61
3.6	Issuer common names of forged certificates	62
4.1	Validity lifetime of OCSP responses.	76
4.2	Response times of OCSP responders.	77
4.3	Latency measurements for a Snap Start handshake with prevalidated server certificate (no verification during the handshake), a Snap Start handshake with online certificate verification, and a normal (unabbreviated) TLS handshake.	87
5.1	Comparison of Certificate Revocation Approaches	104
6.1	TLS certificate chains for evaluation issued by Symantec CA	114
6.2	Comparing ECDHE-RSA and ECDHE-ECDSA	122

List of Figures

1.1	A standard TLS handshake, with RSA key exchange and no client certificate.	14
2.1	Auger’s Flash-based IP hijacking attack on a benign transparent proxy . .	23
2.2	Our Java-based cache poisoning attack on a benign transparent proxy . . .	25
2.3	Performance of WebSocket with 1,000-byte data frames	40
2.4	Performance of WebSocket with 100-byte data frames	41
2.5	Performance of WebSocket with 10-byte data frames	41
3.1	An SSL man-in-the-middle attack between the browser and the server, using a forged SSL certificate to impersonate as the server to the client. . .	46
3.2	Detecting TLS man-in-the-middle attacks with a Flash applet.	49
3.3	Geographic distribution of forged SSL certificates generated by the mali- cious issuer <code>lopFailZeroAccessCreate</code>	66
4.1	Cumulative distribution of OCSP lookup response times.	77
4.2	Median latency and throughput for HTTP HEAD requests with different types of prefetching traffic.	88
4.3	Data transfer overhead for certificate prefetching.	88

5.1	Expired certificate warning in Google Chrome <i>When encountering expired certificates, the browser blocks the page, removes the lock icon from the address bar, and presents a security warning dialog to the user.</i>	98
5.2	Short-Lived Certificates and Chrome's CRL <i>1. Certificate authorities pre-sign batches of short-lived certificates. 2. A plugin on the SSL website periodically fetches fresh short-lived certificates from the CA and loads it onto the server. 3. Google generates a list of revoked certificates, and automatically updates the global CRL to browsers. 4. When the user visits the SSL website, the certificate is validated against Chrome's CRL and also fails on expiration.</i>	100
6.1	Server throughput of different configurations under synthetic traffic	115
6.2	Comparison of TLS setup times in Chrome browsers on Windows, OS X and Android. The box plots show the 10th, 25th, 50th, 75th and 90th percentiles of measured TLS setup times for each cipher suite. The corresponding bar charts show the number of unique clients that successfully completed each test.	118
6.3	Comparison of TCP + TLS setup times in Chrome, Firefox and Internet Explorer browsers (on all platforms).	119

Chapter 1

Introduction

The World Wide Web has grown into a versatile platform that supports an increasing variety of computing tasks. Modern web applications not only deliver static content, but also offer rich interactive experiences such as managing personal communications and processing credit card purchases. As web applications nowadays often involve handling sensitive user information, security is absolutely critical. Arguably, the web platform has been successful due to providing the security guarantee that *users can safely visit arbitrary websites and execute scripts provided by those sites*. Essentially, browsers mediate how web applications can access the user's network according to a handful of important restrictions, known as the *same-origin policy* [1]. However, web security can be challenging since the platform is still rapidly evolving and new functionalities are constantly added. Moreover, the interactions with the underlying network transport path, in particular the unpredictable behaviors of certain *network intermediaries*, are still not fully understood.

Network intermediaries are servers (or applications) that relay network packets between clients and servers (e.g., acting as the network's default gateway). Network intermediaries have been quite commonly deployed across the Internet for various useful purposes, including improving performance (e.g., caching static web resources) and improving security (e.g., monitoring and filtering malicious software). However, network

intermediaries can actually cause more harm than good. Unsurprisingly, a dishonest network intermediary that acts maliciously could steal the user’s unencrypted data (e.g., Firesheep [2]). The less obvious argument is that, a seemingly benign network intermediary could also severely degrade security (e.g., due to mis-implementations). For instance, Auger [3] described how a web attacker can leverage transparent proxies to establish connections with any host accessible by the proxy, including hosts within the user’s private network. Even worse, we demonstrate new attacks that allow a web attacker can poison a transparent proxy’s cache for arbitrary URLs, causing all users of the proxy to receive the attacker’s malicious content [4].

Rather than solely wishing all misbehaving intermediaries (including dishonest intermediaries, as well as benign but mis-implemented intermediaries) to be removed from the Internet, a readily available solution to protect sensitive web communications is the Transport Layer Security (TLS) [5] encryption protocol. With TLS enabled, web applications and browsers can cryptographically ensure that private communications are not eavesdropped or tampered during transit, especially over untrusted networks. Despite that TLS has been supported across major platforms and software libraries, many websites still have not deployed TLS. One of the frequently-cited reasons is that using TLS imposes a significant performance penalty. Indeed, TLS induces some computational overhead and, even more so, adds round-trip latencies when establishing connections. Moreover, existing studies [6] have indicated that website slowdowns further results in loss of users, reputation and revenue.

In light of these existing problems, there are two major research questions that we aimed to answer in this thesis:

Q1: How prevalent are the harmful network intermediaries on the Internet?

Q2: How can we improve the performance of TLS (and encourage adoption)?

To answer research question **Q1**, we designed Internet experiments to detect misbehaving network intermediaries in the wild, including mis-implemented and malicious intermediaries. To detect mis-implemented intermediaries, we leveraged advertisement networks to demonstrate web-based attacks against victim servers in our laboratory. To detect malicious intermediaries, we developed a novel method to detect unauthorized interception of TLS connections, by observing the server’s certificate from the client’s perspective, and experimented our system on a large global website.

For research question **Q2**, we studied the current TLS protocol and proposed three ideas to reduce the performance overhead without sacrificing security, as follows:

- Browsers can prefetch and prevalidate TLS certificates of websites to reduce a significant portion of the initial connection time.
- Certificate authorities (CAs) can issue short-lived certificates to websites, reducing the client-side latencies due to online certificate revocation lookups.
- When enabling TLS forward secrecy, websites can use elliptic curve cryptography to improve performance over traditional RSA cipher suites.

1.1 Threat Models

To study the browser’s ability to protect users from misbehaving network intermediaries, we consider two threat models that are most commonly used for analyzing the security of web browsers, including (1) the web attacker and (2) the network attacker. We study how these two types of attackers may leverage network intermediaries to cause harm.

1.1.1 Web Attacker

The *web attacker*, proposed by Jackson [7], is a malicious principal who controls its own web servers and domain names. Furthermore, it is assumed that the victim user visits the web attacker’s website and renders malicious content in the user’s browser (for instance, displaying an advertisement banner from the web attacker). We note that the web attacker is still constrained by the browser’s security policy and that merely rendering the attacker’s content is sufficient (without user interaction). We do not assume that the web attacker employs a low-level exploit (e.g., buffer overflow in the browser implementation that can execute arbitrary code) to bypass the browser’s security policy, nor do we assume that the user mistakenly inputs any sensitive information on the web attacker’s website.

While the web attacker can control the network traffic sent from its own servers, it does not have direct access to the network connections between the browser and other websites. The web attacker cannot eavesdrop or modify the user’s network traffic with other sites, nor can it generate spoofed responses that purport to be from some other website. However, we consider that some network intermediaries may be implemented erroneously by a benign operator, such that the relayed traffic might be mis-handled under certain circumstances.

1.1.2 Network Attacker

A *network attacker* is a malicious principal who has direct control over the user’s network connections. For example, a network attacker is capable of setting up a malicious WiFi hotspot, such that all of the user’s traffic is sent through the attacker’s router. It is assumed that network attackers have all the abilities of a web attacker. This is evident, since the network attacker can intercept HTTP requests and inject malicious responses that purport to come from a web attacker’s website. Similarly, it is assumed that the

network attacker does not have access to secrets stored on other web servers, nor does it have special access to the client's machine.

Such an attacker could intercept network connections between the user and the legitimate website, and choose to eavesdrop the user's network traffic (known as a *passive network attacker*), or even modify the user's network packets (known as an *active network attacker*). Notably, the Transport Layer Security (TLS) [5] protocol was designed to secure communications against network attackers.

1.1.3 Out-of-Scope Threats

We do not consider a number of related threats. For example, in a *malware* attack, the attacker runs malicious software on the user's machine. Such attackers are capable of bypassing browser-based defenses completely, since the malware could possibly overwrite the user's browser with a custom-built executable, or read confidential data from the machine's storage or memory. Malware attackers can also tap into the user's network interfaces, thus are assumed have all abilities of a network attacker.

Another type of attack, called *phishing*, lures the user to enter their account passwords and other personal information on the attacker's website (that is disguised as another legitimate site, such as a popular bank). Phishing web sites can copy the appearance of the victim web site, and often use carefully selected domains (e.g., `goog1e.com`) or URLs to confuse the users to believe that they are actually visiting a legitimate site. In practice, attackers can send forged emails to the victim that appear to be from a trusted friend (generally known as *social engineering*), containing a link to the attacker's site. Although phishing attackers have identical abilities of a web attacker, the additional assumption is that the user does not accurately check the browser's URL address bar and other security-related indicators.

Cross-site scripting (XSS) is a type of web site vulnerability that allows the attacker’s malicious scripts to be injected and executed on a legitimate site. XSS attacks are known to bypass the browser’s security policies and exfiltrate sensitive information from the victim’s browser. In our work, we assume that sites correctly deploy XSS defenses (e.g., HTML escaping), and consider new web attacks that can succeed even without XSS vulnerabilities.

1.2 Browser Network Access Policies

In this section, we review the network access mechanisms browsers provide to web applications in the context of the web attacker threat model described in Section 1.1.1. In particular, we consider a network topology in which the user connects to the Internet via a transparent proxy, as is common in enterprise networks. The transparent proxy intercepts outbound HTTP requests, perhaps to monitor employee network access, to enforce a security policy, or to accelerate web traffic.

The browser is tasked to enforce a set of security policies that prevent malicious web sites from interacting arbitrarily with other hosts from the client’s IP address. Our assumption is that the victim user visits the malicious web site, that the browser properly enforces its security policy, and that the attacker has no direct control over the network intermediaries. The relevant question, then, is what security policy should the browser enforce on the malicious web site’s network access?

1.2.1 Same-Origin Policy

One natural response to the threat of web attackers is to simply forbid web applications running in the browser from communicating with any server other than the one hosting the application. This model, called the same-origin policy, was first introduced for Java

applets. Java was originally designed as a general purpose programming language and so, unsurprisingly, offers generic networking primitives, including an API that lets the programmer request the virtual machine to open a raw socket to an arbitrary network address and port. If the virtual machine fulfilled these requests unconditionally, these API would be extremely dangerous. For this reason, Java allows network connections only to the source of the Java bytecode.¹

Unfortunately, Java's notion of "source" has proved to be quite problematic. One natural definition of "source" is to simply compare host names, but there is no guarantee that the same host name will always be bound to servers controlled by the same entity. In particular, if the Java virtual machine does its own name resolution, then the system becomes vulnerable to DNS rebinding attacks [8, 9]. In these attacks, it is assumed that the victim visits the attacker's web site (e.g., **attacker.com**) that loads a malicious Java applet. The attacker's DNS server is programmed to respond to the browser's initial DNS query (for **attacker.com**) with the IP addresses of the attacker's server, which serves the attacker's malicious Java applet. Subsequently, the attacker's Java applet running on the client's machine is instructed to open a socket to **attacker.com**. Since the DNS response for **attacker.com** has expired (due to a short time-to-live), the Java virtual machine resolves the host name again, but this time the attacker's DNS server indicates that **attacker.com** points to the target server. As a result, the applet (which is under the attacker's control) opens a socket connection to the target server from the client's IP address. DNS rebinding attacks have been known for a long time and are addressed by basing access control decisions on the IP address rather than the host name, either directly by checking against the IP address (as in Java) or by *pinning*, forcing a constant mapping between DNS name and IP address regardless of the time-to-live of the DNS

¹These restrictions do not apply to signed applets which the user has accepted. Those applets have the user's full privileges.

response.

1.2.2 Verified-Origin Policy

Unfortunately, the same-origin policy, strictly construed, is quite limiting: many web application developers wish to communicate with other web sites, for example to incorporate additional functionality or content (including advertisements). Allowing such communication is unsafe in the general case, but the browser can safely allow communication as long as it verifies that the target site consents to the communication traffic. There are a number of Web technologies that implement this *verified-origin policy* [10].

Flash Cross-Domain Policies

Prior to letting Adobe Flash multimedia files, known as Small Web Format (SWF) files, open a socket connections to a server, Flash Player first connects to the site and fetches a cross-domain policy file: an XML blob that specifies the origins that are allowed to connect to that site [11].² The location of the policy file is itself subject to a number of restrictions, which make it more difficult for an attacker who has limited access to the target machine to generate a valid file. For instance, policy files hosted on port 1024 or higher cannot authorize access to ports below 1024.

Flash Player uses the same general mechanism to control access both to raw sockets and to cross-domain HTTP requests. As with Java, Flash Player's consent mechanism was vulnerable to DNS rebinding attacks in the past [8]. Indeed, the mechanism described above where the cross-domain policy file is always checked is a response to some of these rebinding attacks which exploited a *time-of-check-time-of-use* (TOCTOU) issue between the browser's name resolution and that performed by Flash Player.

²This description is a simplification of Flash Player's security policy [12].

JavaScript Cross-Origin Communication

Until recently, network access for JavaScript applications was limited to making HTTP requests via `XMLHttpRequest`. Browsers heavily restrict these requests and forbid requesting cross-origin URLs [13]. Recently, browser vendors have added two mechanisms to allow web applications to escape (hopefully safely) from these restrictions.

- **Cross-Origin Resource Sharing.** Cross-Origin Resource Sharing (CORS) [14] allows web applications to issue HTTP requests to sites outside their origin. When a web application issues a cross-origin `XMLHttpRequest`, the browser includes the application's origin in the request in the `Origin` header. The server can authorize the application to read back the response by echoing the contents of the `Origin` request header in the `Access-Control-Allow-Origin` response header. This consent-base relaxation of the same-origin policy makes it easier for different web applications to communicate in the browser.
- **WebSocket.** Although CORS is targeted only at HTTP requests, the WebSocket protocol [15] lets web applications open a socket connection to any server (whether or not the server is in the application's own origin) and send arbitrary data. This feature is extremely useful, especially as an optimization for scenarios in which the server wishes to asynchronously send data to the client. Currently, such applications use a rather clumsy set of mechanisms generally known as Comet [16]. Like Flash Player and CORS, WebSocket uses a verified-origin mechanism to let the target server consent to the connection. Unlike Flash Player and CORS, the verification is performed over the same socket connection as will be used for the data (using a cryptographic handshake where the server replies to a client-provided nonce). This handshake is initiated by the browser and only after the handshake has completed

does the browser allow the application to send data over the raw socket.

1.3 Why are Network Intermediaries Dangerous?

In this section, we articulate how network intermediaries can cause harm to the security of web applications and browsers. Existing literature [17] describes how mis-implemented proxies, or proxy servers, may fail to interoperate with the rest of the Web and cause unexpected errors in web applications (due to improperly handling HTTP headers).

We further study the security impact of bad network intermediaries. We consider two separate scenarios: (1) a benign but confused network intermediary controlled by an honest operator, and (2) a malicious network intermediary controlled by a dishonest operator. Although the dangers of a malicious network intermediary are rather evident, the less obvious argument is that even a benign network intermediary could also severely degrade security. First, we describe how transparent proxies may be vulnerable to web attacks. Second, we determine whether TLS connections are being intercepted on a large global website.

1. **Transparent Proxy Vulnerabilities.** Java, Flash Player, and HTML5 provide socket APIs to web sites, but we discovered, and experimentally verified, web attacks that exploit the interaction between these APIs and transparent proxies. At a cost of less than \$1 per exploitation, a web attacker can remotely poison the proxy's cache, causing all clients of the proxy to receive malicious content.
2. **Unauthorized Interceptions.** In a TLS man-in-the-middle attack, the network attacker uses forged TLS certificates to intercept encrypted connections between clients and servers. We designed and implemented a method to detect the occurrence of TLS man-in-the-middle attack on a top global website, Facebook.

1.3.1 Transparent Proxy Vulnerabilities

Unlike traditional HTTP proxies, which are explicitly configured and known to the client, *transparent proxies* insert themselves into the transport path (e.g., by acting as the network’s default gateway or as a bridge) and then act as proxies without the client’s knowledge. Such proxies are common in traffic filtering applications but also can serve as network accelerators or proxy caches.

Unfortunately, these transparent proxies often forward the server’s consent without understanding its semantics. When a server provides a Flash policy file authorizing a SWF to connect to the server’s IP address on port 80, Flash Player will allow the SWF to open a raw socket connection to the server, not aware that the SWF is actually talking to a transparent proxy instead of the server itself. Once the attacker has opened a socket to the proxy server, the type of harm that the attacker can perform depends on details of how the proxy behaves. For example, Auger [3] describes how an attacker can leverage transparent proxies to establish connections with any host accessible by the proxy.

We present new attacks that can poison the proxy’s cache for an arbitrary URL, causing users of the proxy to receive the attacker’s malicious content instead of the honest server’s content. Such attacks are critical, since every successful cache poisoning attack would also affect *all* users of the vulnerable proxy (potentially the entire enterprise). In response to our discovery of transparent proxies vulnerabilities, the WebSocket protocol adopted our proposal to obfuscate application payloads.

1.3.2 Unauthorized Interceptions

TLS man-in-the-middle attack attempts have previously been reported in the wild (i.e. in Iran [18] and Syria [19]). However, it is unclear how prevalent these attacks actually are. We introduce a novel method for websites to detect man-in-the-middle attacks on

a large scale, without alterations on the client’s end (e.g., customized browser). We utilized the widely-supported Flash Player plugin to enable socket functionalities not natively present in current browsers, and implemented a partial TLS handshake on our own to capture forged certificates. We deployed this detection mechanism on an Alexa top 10 website, Facebook, which terminates connections through a diverse set of network operators across the world. Based on real-world data, we categorized the most common causes of TLS interceptions. We showed that most of the TLS interceptions are due to antivirus software and organization-scale content filters. We provided evidence of SSL interceptions by malware, which have infected users across at least 45 countries.

1.4 Why isn’t TLS More Widely Deployed?

In this section, we provide background on the Transport Layer Security (TLS) protocol, which is designed to protect against network attackers (described in Section 1.1.2). We then discuss the most common reasons of why TLS is not more widely deployed.

1.4.1 Background

Transport Layer Security (TLS) [5] provides authentication based on the X.509 public key infrastructure [20], protects data confidentiality using symmetric encryption, and ensures data integrity with cryptographic message digests.³ To establish a secure connection, the client and server perform a *TLS handshake* in which each party can authenticate itself by providing a certificate signed by a trusted certificate authority (CA). Using a *cipher suite* negotiated in the handshake, the client and server agree on a key to secure the application data that is sent after the handshake.

³TLS is the successor of Secure Sockets Layer (SSL) [21].

Cipher Suites

The TLS protocol supports an extensible set of cipher suites, where each cipher suite defines a combination of authentication, key exchange, bulk encryption, and message authentication code (MAC) algorithms to be used.

- **Authentication** algorithms allow communicating parties to verify the identities of each other based on public key cryptography, e.g., RSA, DSA, and ECDSA.
- **Key exchange** schemes allow peers to securely agree upon on a session key used for the bulk encryption of subsequent payloads, e.g., RSA, ephemeral Diffie-Hellman (DHE), and ephemeral Elliptic Curve Diffie-Hellman (ECDHE).
- **Bulk encryption** ciphers are used to encrypt application data, e.g., AES and RC4.
- **MAC** algorithms are used to generate message digests, e.g., SHA-1 and SHA-256.

TLS Handshake

Figure 1.1 shows a full TLS handshake using RSA key exchange and no client certificate, which is a common configuration on the web. The **ClientHello** and **ServerHello** establish an agreement between the client and the server on which version of TLS and which cipher suite to use. The **ClientHello** message specifies a list of client-supported cipher suites and a client-generated random number (the *pre-master secret*). The **ServerHello** message carries the server-chosen cipher suite and a server-generated random number. These initial messages also allow the client and server to exchange fresh random values used in deriving the session key, which prevents message replay. In addition, the **Certificate** message contains the server's public key certificate, digitally signed by a CA, in which the client is responsible of verifying. The client then encrypts the pre-master secret using the

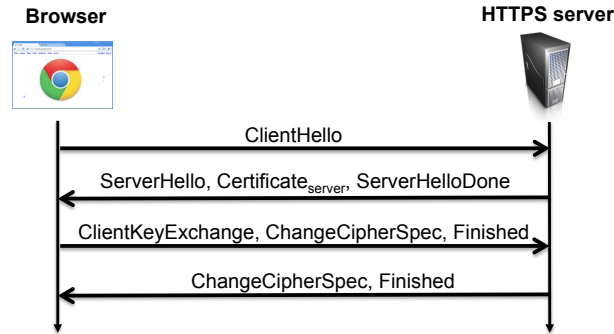


Figure 1.1: A standard TLS handshake, with RSA key exchange and no client certificate.

server's public key and sends the pre-master secret to the server over a **ClientKeyExchange** message. After the server has received the **ClientKeyExchange** message, both the client and the server can derive the master key with which the application data is encrypted. The **ChangeCipherSpec** messages indicate to the other party that subsequent messages will be encrypted with the negotiated cipher suite. **Finished** messages contain a hash of the entire handshake to ensure to both parties that handshake messages have not been altered by a network attacker. After two round trips between the client and server, the client can finally send application data over the encrypted connection.

Certificate Validation

In the X.509 [20] public key infrastructure, a certificate issued by a CA binds a public key with an individual, commonly a domain name. Fundamentally, a valid certificate must be signed by a trusted source. Web browsers and operating systems come with a pre-installed list of trusted signers in their root CA store. More often, the root CAs will not directly sign certificates due to security risks, but delegate authority to intermediate CAs that actually sign the certificates. Therefore, the browser should verify that the leaf certificate is bundled along with a certificate chain leading to a trusted signer.

To determine the validity period of a public key certificate, each certificate specifies

the date it becomes valid, and the date it expires. It is current standard practice for commercial CAs to issue certificates with a relatively long validity period such as a year or longer. In addition, X.509 defines mechanisms for the issuing CA to revoke certificates that haven't expired but should no longer be trusted, e.g., when the private key corresponding to the certificate has been compromised, or more often because the certificate was reissued. The common certificate revocation checking mechanisms are Certificate Revocation Lists (CRL) [20] and the Online Certificate Status Protocol (OCSP) [22].

- **Certificate Revocation Lists.** The basic idea of CRL is that, when a certificate goes bad, its identifying serial number is published to a list, signed and timestamped by a CA. In order to trust a certificate, the client must ensure that the certificate is not listed in CRLs. CRLs are published by CAs at a URL indicated by the CRL distribution point extension. We note that for CRLs to be effective, one assumes that (1) up-to-date lists are published frequently by the CA, (2) the most recent list is available to the verifier, and (3) verification failures are treated as fatal errors.
- **Online Certificate Status Protocol.** The Online Certificate Status Protocol (OCSP), an alternative to CRLs proposed in RFC 2560 [22], allows client software to obtain current information about a certificate's validity on a certificate-by-certificate basis. When verifying a certificate, a client sends an OCSP request to an OCSP responder, which responds whether the certificate is valid or not. OCSP responders themselves are updated by CAs as to the status of certificates they handle. Typically, clients are instructed to cache the OCSP response for a few days [23].

1.4.2 Deployment Concerns

TLS provides encryption for web applications and significantly raises the difficulty of network attacks (as well as erroneous handling by transparent network intermediaries).

While the security benefits of TLS are clear, TLS is still not pervasively deployed across websites. To understand why TLS is not more widely deployed, we point out some of the common arguments against deploying TLS.

- **Computational Costs.** Enabling TLS introduces additional computational costs (e.g., CPU, memory, etc.) on servers and clients, including asymmetric encryption for authentication and symmetric cryptography to encrypt all TLS records. Due to encryption, TLS may also induce some additional network bandwidth usage due to the encryption overhead. Fortunately, significant improvements in modern CPUs have reduced the impact of these costs on servers [24], and may eventually reduce the need for dedicated “SSL accelerator” hardware [25]. Furthermore, session resumption support (with TLS Session ID or TLS Session Tickets) on servers also reduces the computational costs.
- **Handshake Latency.** The standard TLS handshake requires two round trips before a client or server can send application data. A key bottleneck in a full TLS handshake is the need to fetch and validate the server certificate before establishing a secure connection. The web browser must validate the server’s certificate using certificate revocation protocols such as the Online Certificate Status Protocol (OCSP) [22], adding more latency and leading clients to cache certificate validation results. The network latency imposed by the handshake impacts user experience and discourages websites from enabling TLS. Moreover, existing studies [6] have indicated that slower websites result in loss of users, reputation and revenue.
- **Proxy Caching.** Since all encrypted TLS communications should appear oblivious to network intermediaries, web proxies cannot simply perform caching on encrypted traffic. As a result, more requests from clients must be handled at the servers,

which may also incur higher server load as well as larger client-perceived round trip latencies. In response, many modern websites have employed content delivery networks (CDNs) across the world to offload the server traffic and minimize client-perceived round trip times.

- **Browser Caching.** A little-known but significant contributor to the cost of TLS is the modified browser caching behavior under HTTPS. We give two examples.
 - First, Internet Explorer will not use locally cached HTTPS content without first establishing a valid TLS connection to the source web site [26]. While web servers can use a `Cache-Control` header to tell the browser that certain content is static and can be used directly from cache, Internet Explorer ignores this header for HTTPS content and insists on an HTTPS handshake with the server before using the cached content (in IE9 this session is used to send an unnecessary `If-Modified-Since` query). This behavior is especially damaging for sites who use a content distribution network (CDN) since IE will insist on an HTTPS handshake with the CDN before using the cached content. These redundant handshakes, which include a certificate validation check, discourage web sites from using HTTPS.
 - Second, some browsers, such as Firefox, are reluctant to cache HTTPS content unless explicitly told to do so using a `Cache-Control: public` header [27]. Websites that simply turn on TLS without also specifying this header see vastly more HTTPS requests for static content. This issue has been fixed in Firefox 4.
- **Operational Costs.** Although popular web servers all provide support for TLS, properly configuring and maintaining a TLS server can be a burden for the deployers [28]. First of all, acquiring valid certificates from commercial CAs may incur a

cost. Moreover, properly installing the certificates and rotating the keys securely can be a challenging task. Furthermore, correctly configuring a TLS server (e.g., cipher suites, Diffie-Hellman key sizes, etc.) requires a certain amount of knowledge, which can be (surprisingly) difficult.

- **Breakage.** There may be a small portion of legacy browsers or clients that do not support the latest TLS protocols, and may encounter errors when connecting to secure websites. For example, the Server Name Indication (SNI) extension for TLS enables multiple TLS servers to be hosted on the same IP address using distinct certificates, but is still not supported on certain mobile or legacy platforms. Similarly, certain transparent network intermediaries may fail to forward TLS traffic correctly, and might break the user experience of secure websites. These interoperability issues may discourage websites to deploy TLS at the risk of site breakage. Lastly, network intermediaries are often deployed in enterprise networks for auditing or security purposes. However, enabling TLS essentially breaks traffic monitoring tools, and would require additional work to install custom root certificates on the client's system.

1.5 Opportunities for Improving TLS Performance

As discussed in Section 1.4.2, one of the most common arguments against deploying TLS falls in the category of performance-related concerns. To encourage wider use of encryption, we identify three opportunities to improve TLS performance, without sacrificing security.

1. **Prefetching and Prevalidating Certificates.** A key bottleneck in a full TLS handshake is the need to fetch and validate the server certificate before establishing

a secure connection.

2. **Short-Lived Certificates.** OCSP imposes a massive performance penalty on TLS yet failed to mitigate multiple high-profile CA breaches [29, 30].
3. **Forward Secrecy Performance.** Contrary to traditional performance arguments, forward secrecy (using elliptic curve cryptography) is not much slower, and can even be faster, than RSA-based setups with no forward secrecy.

1.5.1 Prefetching and Prevalidating Certificates

We propose that the client-perceived TLS connection time can be reduced in two ways: (1) eliminate the certificate validation time by enabling prefetching and prevalidation of server certificates, and even (2) eliminate two round trips by enabling the abbreviated TLS handshake. This requires the client to obtain the server certificate and parameters beforehand, when it is likely that the user might navigate to the website. The browser can use the same triggers that it uses to pre-resolve hostnames [31] to determine when certificate prefetching is useful: for example, when the user is typing in the address bar or when the user’s mouse cursor hovers over a HTTPS link.

A naïve prefetching method is to open a *dummy* TLS connection to the server. These dummy connections pre-warm the client’s connection cache basically performing a standard TLS handshake with the server, and would eventually disconnect on timeout. We discuss four more options for certificate prefetching that induce less server load:

- **Prefetching with a truncated handshake.** The browser can prefetch the certificate by initiating a TLS connection, but truncate the handshake (via the TLS Alert protocol) before the computationally expensive steps are performed.

- **Prefetching via HTTP GET.** The browser can prefetch the certificate information as a special file from a standardized location on the HTTP server.
- **Prefetching from a CDN.** To avoid placing any extra load on the server, a client can attempt to prefetch certificate information (as a file) from a CDN.
- **Prefetching from DNS.** Alternatively, the server may place its certificate information in a DNS record to offload the prefetching traffic.

1.5.2 Short-Lived Certificates

We propose to abandon the existing revocation mechanisms in favor of an old idea — short-lived certificates. A short-lived certificate is identical to a regular certificate, except that the validity period is configured with a short span of time. For instance, CAs could configure the validity period of short-lived certificates to match the average validity lifetime of an OCSP response that we measured in real-world, which was 4 days [23]. Such certificates expire shortly, and most importantly, *fail-closed* (treating them as insecure) after expiration on clients without the need for a revocation mechanism. In our proposal, when a web site purchases a year-long certificate, the CA's response is a URL that can be used to download on-demand short-lived certificates. The URL remains active for the year, but issues certificates that are valid for only a few days.

We argue that short-lived certificates are far more efficient than CRLs and OCSP (since online revocation checks are eliminated), and require no client-side changes. Moreover, since clients typically fail closed when faced with an expired certificate, this approach is far more robust than the existing best-effort OCSP and CRL based approaches. We further note that browser-maintained global CRLs complement nicely with short-lived certificates. Browser vendors will be able to revoke fraudulent certificates and rogue CAs,

while CAs may control administrative revocations for benign certificates.

1.5.3 Forward Secrecy Performance

Forward secrecy guarantees that eavesdroppers simply cannot reveal secret data of past communications. Given current parameter and algorithm choices, we show that the traditional performance argument against forward secrecy is no longer true. We compared the server throughput of various TLS setups supporting forward secrecy, and measured real-world client-side latencies using an ad network. Our results indicate that forward secrecy is no harder, and can even be faster using elliptic curve cryptography (ECC), than no forward secrecy. We suggest that sites should migrate to ECC-based forward secrecy for both security and performance reasons.

1.6 Organization

The remainder of this thesis is organized as follows. In Chapter 2, we explain transparent proxy vulnerabilities, including existing and new attacks, and describe how the WebSocket protocol adopted our defense proposal to prevent these attacks. In Chapter 3, we describe how we detected unauthorized TLS man-in-the-middle attacks on a large global website, and investigate why the connections were being intercepted. Chapter 4 describes how prefetching and prevalidating TLS certificates can significantly reduce the handshake latencies. Chapter 5 explains how using short-lived certificates can eliminate the latencies of certificate revocation lookups. Chapter 6 surveys TLS forward secrecy deployments and evaluates the performance of using elliptical curve cryptography over traditional RSA-based setups. Finally, Chapter 7 concludes.

Chapter 2

Transparent Proxy Vulnerabilities

The work in this chapter was done in collaboration with Eric Chen, Adam Barth, Eric Rescorla, and Collin Jackson.

In this chapter, we show that the consent protocols (described in Section 1.2) for network access used by browsers today are vulnerable to attack in network configurations involving network intermediaries, specifically *transparent proxies*. Unlike traditional HTTP proxies, which are explicitly configured and known to the client, transparent proxies insert themselves into the transport path (e.g., by acting as the network’s default gateway or as a bridge) and then act as proxies without the client’s knowledge. Such proxies are common in traffic filtering applications but also can serve as network accelerators or proxy caches. Although colloquially referred to as “transparent” proxies, these proxies are more accurately termed “intercepting” proxies because, as we show, they are not quite as transparent as their deployers might wish.

2.1 Attacks on Java and Flash Sockets

Consider the situation in which the user is behind a transparent proxy and visits `attacker.com`. The attacker embeds a malicious SWF served from `attacker.com`, and the browser uses

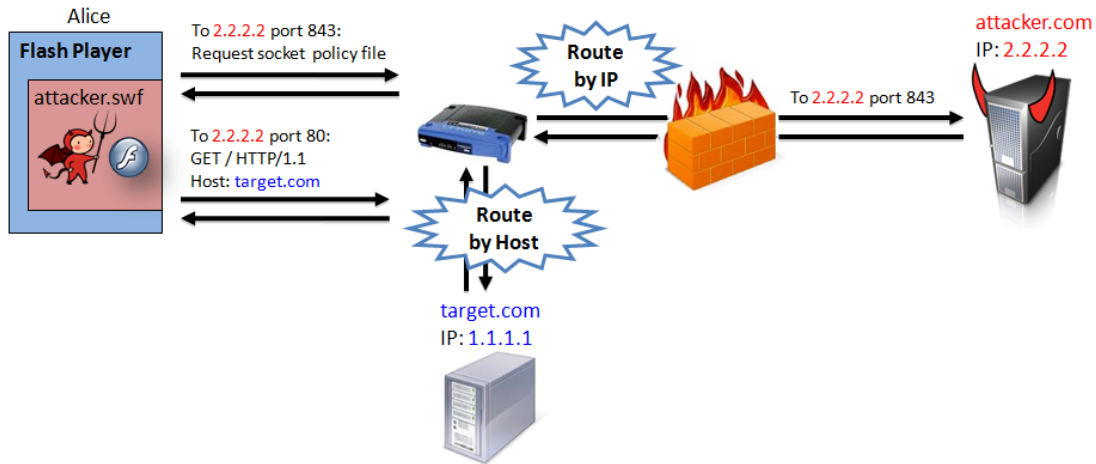


Figure 2.1: Auger's Flash-based IP hijacking attack on a benign transparent proxy

Flash Player to run the SWF. The attacker can now mount a number of different attacks, depending on how the proxy behaves.

When using a traditional proxy, the browser connects directly to the proxy and sends an HTTP request, which indicates to the proxy which resource the browser wishes to retrieve. When a transparent proxy intercepts an HTTP request made by a browser, the proxy has two options for how to route the request:

- The **HTTP Host header**.
- The **IP address** to which the browser originally sent the request.

2.1.1 Auger's IP Hijacking Attack

Unfortunately, as described by Auger [3], if the proxy routes the request based on the `Host` header, an attacker can trick the proxy into routing the request to any host accessible to the proxy, as depicted in Figure 2.1:

1. The attacker hosts a permissive Flash socket policy server on `attacker.com:843` that allows access to every port from every origin.

2. The attacker's SWF requests to open a raw socket connection to `attacker.com:80` (which has IP address `2.2.2.2`).
3. Flash Player connects to `attacker.com:843` and retrieves the attacker's socket policy file, which indicates that the server has opted into the socket connection.
4. Flash Player lets the attacker's SWF open a new socket connection to `attacker.com:80`.
5. The attacker's SWF sends a sequence of bytes over the socket crafted with a fake Host header as follows:


```
GET / HTTP/1.1
Host: target.com
```
6. The transparent proxy treats these bytes as an HTTP request and routes the request according to the `Host` header (and *not* on the original destination IP address). Notice that the request is routed to `target.com:80` (which has an IP address of `1.1.1.1`).
7. The target server responds with the document for the URL `http://target.com/`, requested from the client's IP address, and the transparent proxy forwards the response to the attacker's SWF.

Notice that Flash Player authorized the attacker's SWF to open a socket to the attacker's server based on a policy file it retrieved from the attacker's server. However, the transparent proxy routed the request to a different server because the socket API let the attacker break the browser's security invariant that the `Host` header matched the destination IP address, leading to the vulnerability. Alternatively, the attacker can try to trick the proxy into tunneling a raw socket connection to the target server by using the HTTP `CONNECT` method [32] in Step 5:

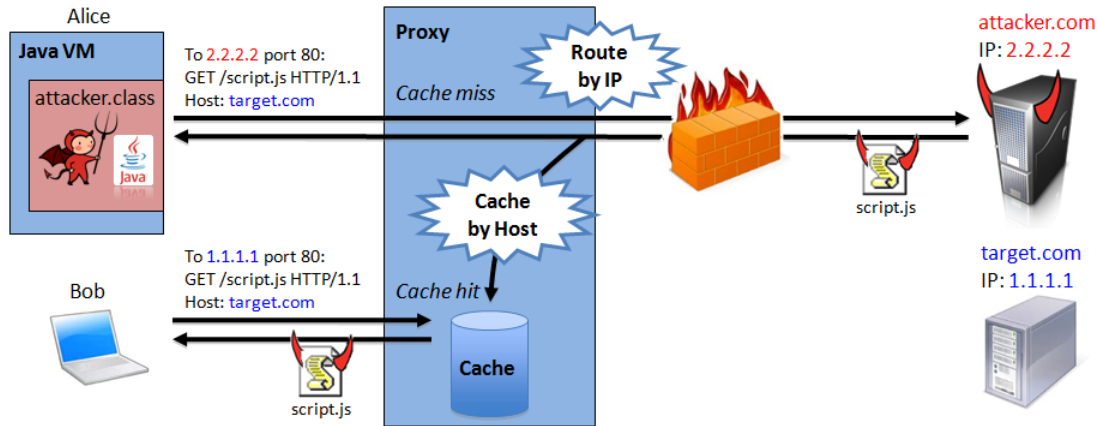


Figure 2.2: Our Java-based cache poisoning attack on a benign transparent proxy

```
CONNECT target.com:80 HTTP/1.1
Host: target.com:80
```

By leveraging the user's machine to connect to other hosts in the Internet over these proxies, the attacker may hijack a user's IP address to perform misdeeds and frame the user. For example, the attacker may generate fake clicks on pay-per-click web advertisements to increase their advertising revenue [33], using different client IP addresses. IP hijacking attacks may also allow web attackers to access protected web sites that authenticate by IP address, or send spam email from the victim user's IP address.

An attacker can also exploit Java sockets in the same way. The attack steps are identical, except that the attacker need not host a policy file because Java implicitly grants applets the authority to open socket connections back to its origin server without requiring the server to consent.

2.1.2 Our Cache Poisoning Attack

In the attacks described in the previous section, we considered transparent proxies that route HTTP requests according to the `Host` header. However, not all proxies are configured that way. Some proxies route the request to the original destination IP address,

regardless of the `Host` header. Although these proxies are immune to IP hijacking attacks, we find that the attacker can still leverage some of these proxies to mount other attacks.

In particular, some transparent proxies that route by IP are also caching proxies. As with routing, proxies can cache responses either according to the `Host` header or according to the destination IP address. If a proxy routes by IP but caches according to the `Host` header, we discover that the attacker can instruct the proxy to cache a malicious response for an arbitrary URL of the attacker's choice, as shown in Figure 2.2:

1. The attacker's Java applet opens a raw socket connection to `attacker.com:80` (as before, the attacker can also a SWF to mount a similar attack by hosting an appropriate policy file to authorize this request).
2. The attacker's Java applet sends a sequence of bytes over the socket crafted with a forged `Host` header as follows:


```
GET /script.js HTTP/1.1
Host: target.com
```
3. The transparent proxy treats the sequence of bytes as an HTTP request and routes the request based on the original destination IP, that is to the attacker's server.
4. The attacker's server replies with malicious script file with an HTTP `Expires` header far in the future (to instruct the proxy to cache the response for as long as possible).
5. Because the proxy caches based on the `Host` header, the proxy stores the malicious script file in its cache as from `http://target.com/script.js`, instead of the original URL `http://attacker.com/script.js`.
6. In the future, whenever any client requests `http://target.com/script.js` via the proxy, the proxy will serve the cached copy of the malicious script.

One particularly problematic variant of this attack is for the attacker to poison the cache entry for Google Analytics' JavaScript library. Every user of the proxy (possibly the entire enterprise) will now load the attacker's malicious JavaScript into every page that uses Google Analytics, which is approximately 57% of the top 10,000 web sites [34]. Because the Google Analytics JavaScript runs with the privileges of the embedding web site, the attacker is able to effectively mount a persistent cross-site scripting attack against the majority of the Internet, as viewed by users of the proxy.

2.1.3 Ad-based Experiment

The attacks described above have very specific network configuration requirements. To determine how commonplace these network configurations are on the Internet, we developed proof-of-concept exploits for both the IP hijacking and cache poisoning attacks using both Flash Player and Java. We then ran an advertisement on a public advertising network that mounted the attacks against servers in our laboratory.

Methodology

Our experiment consisted of two machines in our laboratory, with different host names and IP addresses. One machine played the role of the target server and the other played the role of the attacking server. The target was a standard Apache web server. The attacking server ran a standard Apache web server and a Flash socket policy server on port 843. We used a rich media banner advertisement campaign on an advertising network to serve our experimental code to users across the world. Our advertisement required no user interaction, and was designed to perform the following tasks in the user's web browser:

- **IP Hijacking.** Our advertisement opens a raw socket connection back to the attacking server using both Java and Flash Player. The attacking server runs a

custom Flash socket policy server on port 843 that allows Flash socket connections to port 80 from any origin. Upon a successful connection, the advertisement spoofs an HTTP request over the socket by sending the following request:

```
GET /script.php/<random> HTTP/1.1
Host: target.com
```

The attacking server and the target server each host a PHP file at `/script.php`, but because these files are different we can easily determine which server the request went to. The random value on the end of the URL serves to bypass caches used by plug-ins, browsers, or proxies. Alternatively, we could have included the random value in the query string (i.e., after a `?` character) but some caching proxies treat URLs containing query strings inconsistently. If the HTTP response was from the target server instead of from the attacking server, that is direct evidence that the request was routed by the `Host` header, which implies that the user is vulnerable to IP hijacking.

- **Cache Poisoning.** In the previous test, the script files were served with the HTTP response headers `Cache-Control: public`, `Last-Modified` and `Expires` that allowed them to be cached for one year. To check whether the socket connection has poisoned the proxy's cache, we added a script tag to our advertisement that attempts to load a script from the target server at `http://target.com/script.php/<random>`, reusing the random value from the previous request. Because the random value was only used previously via the socket API, this URL will not be present in the browser's HTTP cache (as the browser does not observe the bytes sent over the socket). By checking the contents of the response (specifically, a JavaScript variable), we can determine whether the script was from the attacker or the target server. If we re-

	Flash Player	Java
Spoof request routed to target[★]	3152	2109
Spoof request routed to attacker	47839	26759
Script file cached from target	51163	26612
Script file cached from attacker[†]	108	53

Table 2.1: HTTP Host header spoofing via plug-in sockets

★ Allows attacker to open a direct socket from the client to an arbitrary server

† Allows attacker to poison the HTTP cache of all clients of the proxy

ceive the version of the script hosted on the attack server, we can deduce that a transparent proxy has cached the response.

Results

We ran our advertisement on five successive days in March 2011, spending \$100 in total. We garnered a total of 174,250 unique impressions. We discarded repeat visits by the same users by setting a cookie in the user's browser. The advertisement ran our JavaScript, SWF, and Java bytecode without user intervention and sent results back to server in our laboratory after completing the experiment. If the user closed the browser window or navigated away before the experiment finished running, we did not receive the results from that part of the experiment. We collected 51,273 results from SWFs and 30,045 results from Java applets (19,117 of the impressions produced results from both tests). The most likely reason for the low response rate is that the loading time of our SWF and Java applet was noticeably slow, and users did not stay on the page long enough for the experiment to run. Our experimental results show that both IP hijacking attacks and cache poisoning exist in real world scenarios, as shown in Table 2.1.

- **IP Hijacking.** In the IP hijacking test using Flash sockets, we observed that the spoofed request was routed back to the attacking server on 47,839 of 51,273 impressions (93.3%), suggesting that the client made a direct connection or the

network intermediaries routed regardless of the `Host` header. We logged 233 of 51,273 impressions (0.4%) where the Flash socket failed to open, possibly due to firewalls that blocked port 843, preventing Flash Player from fetching the socket policy file. There were 49 cases where the client received an HTML error message, possibly generated by a transparent proxy that blocked the spoofed request. On 3,152 impressions (6.1%) the spoofed request was routed by the `Host` header to the target server, indicating vulnerability to IP hijacking.

Using Java sockets, we observed that 26,759 of 30,045 impressions (89.1%) received the response from the attacker's server, implying that they were routing on IP. Out of 30,045 impressions, there were 1,134 (3.8%) connection errors that threw Java exceptions and 43 that received an HTML error message. We found that 2,109 of 30,045 impressions (7%) routed on the `Host` header, allowing IP hijacking attacks.

- **Cache Poisoning.** In the cache poisoning test using Flash sockets, we observed that 51,163 of 51,273 impressions (99.8%) were able to fetch the script from the target. There were 2 cases where the client reported an error response. However, we discovered that the cache poisoning attack was successful on 108 of 51,273 impressions (0.21%). This suggests that some transparent proxies route HTTP requests by IP but cache according to the `Host` header.

In our cache poisoning test using Java sockets, we observed 26,612 of 30,045 impressions (88.6%) retrieved the response from the target server. We observed that 3,680 of 30,045 impressions (12.2%) caused exceptions when using Java to interrogate the results of the second query, which we were unable to determine whether the cache poisoning succeeded or not. Similarly to the results using Flash sockets, there were 53 of 30,045 impressions (0.18%) that reported a successful cache poisoning attack.

Our results show that the attacker may achieve a cost efficiency of 1.08 successful cache poisoning attacks per dollar spent, using Flash sockets on advertising networks. Note that each successful cache poisoning attack would in effect compromise other users of the vulnerable proxy, beyond our measurement.

2.2 Attacks on WebSocket Protocols

One diagnosis of the cause of the Java and Flash socket vulnerabilities is that both use an out-of-band mechanism to authorize socket connections. Because intermediaries are oblivious to these out-of-band signals, they misinterpret the information sent over the socket by the attacker.

In this section, we consider three *in-band* signaling mechanisms for authorizing socket connections, all based on HTTP. The first is a `POST`-based handshake of our own invention to illustrate some of the design issues. The second is the state-of-the-art `Upgrade`-based handshake used by HTML5. The third is an experimental `CONNECT`-based handshake that was designed in attempt to prevent attacks. We then evaluate the security of these three strawman handshakes for the WebSocket protocol.¹

2.2.1 POST-based Strawman Handshake

Design One natural approach to designing an in-band signaling mechanism is to model the handshake after HTTP. The idea here is that until we have established the server’s consent to receive WebSocket traffic, we will not send any data that the attacker could not already have generated with existing browser functionality—with the HTML form element being the most powerful piece of syntax in this respect—so what could possibly go wrong? This should protect servers which do not want to speak the WebSocket protocol

¹At the time of our initial study, the HTML5 WebSocket protocol was not standardized, therefore, we considered three strawmen proposals.

from being sent WebSocket data. With this goal in mind, consider the following strawman handshake based on an HTTP POST request:

Client → *Server*:

```
POST /path/of/attackers/choice HTTP/1.1
Host: host-of-attackers-choice.com
Sec-WebSocket-Key: <connection-key>
```

Server → *Client*:

```
HTTP/1.1 200 OK
Sec-WebSocket-Accept: <connection-key>
```

By echoing the connection key to the client, the server consents that it accepts the WebSocket protocol. If WebSockets are less generative than the form element, then we might believe that adding WebSocket support to browsers does not increase the attack surface.

Vulnerabilities Unfortunately, using this handshake, WebSockets are not less generative than the HTML form element. For example, WebSocket applications can generate data that appear as framing escapes and confuse network intermediaries into handling subsequent data as new HTTP connections, instead of a continuous single HTTP connection expressed by the form element. Although we have accomplished our initial goal of not sending any non-HTTP data to WebSocket servers, we can still confuse transparent proxies.

Consider an intermediary examining packets exchanged between the browser and the attacker's server. As above, the client requests a WebSocket connection and the server agrees. At this point, the client can send any traffic it wants on the channel. Unfortunately, the intermediary does not know about the WebSocket protocol, so the initial WebSocket handshake just looks like a standard HTTP request/response pair, with the request being terminated, as usual, by an empty line. Thus, the client program can inject

new data which looks like an HTTP request and the proxy may treat it as such. So, for instance, he might inject the following sequence of bytes:

```
GET /sensitive-document HTTP/1.1
Host: target.com
```

When the intermediary examines these bytes, it might conclude that these bytes represent a second HTTP request over the same socket. If the intermediary is a transparent proxy, the intermediary might route the request or cache the response according to the forged `Host` header, discussed in Section 2.1.

2.2.2 Upgrade-based Strawman Handshake

Design In an attempt to improve the security of its socket handshake, HTML5 uses HTTP's `Upgrade` mechanism to upgrade from the HTTP protocol to the WebSocket protocol. HTTP's `Upgrade` mechanism is a generic mechanism for negotiating protocols using HTTP which was originally designed for layering TLS over HTTP. HTTP's `Upgrade` mechanism has two pieces: a `Connection` header whose value is the string "Upgrade" and an `Upgrade` header whose value is the name of the protocol to which the client wishes to switch. Below is a simplified version of the HTML5 WebSocket handshake using HTTP's `Upgrade` mechanism.

```
Client → Server:
GET /path/of/attackers/choice HTTP/1.1
Host: host-of-attackers-choice.com
Connection: Upgrade
Sec-WebSocket-Key: <connection-key>
Upgrade: WebSocket
```

```
Server → Client:
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Upgrade: WebSocket
Sec-WebSocket-Accept: HMAC(<connection-key>, "...")
```

Vulnerabilities Unfortunately, HTTP’s **Upgrade** mechanism is virtually unused in practice. Instead of layering TLS over HTTP using **Upgrade**, nearly every deployment of HTTP over TLS uses a separate port, typically port 443 (the generic name for this mode is HTTPS [35]). Consequently, many organizations are likely to deploy network intermediaries that fail to implement the **Upgrade** mechanism because these intermediaries will largely function correctly on the Internet today. Implementers and users of these intermediaries have little incentive to implement **Upgrade**, and might, in fact, be unaware that they do not implement the mechanism.

To an intermediary that does not understand HTTP’s **Upgrade** mechanism, the HTML5 WebSocket handshake appears quite similar to our strawman **POST**-based handshake. These intermediaries are likely to process the connection the same way for both the **POST**-based handshake and the **Upgrade**-based handshake. If such an intermediary is vulnerable to the attacks on the **POST**-based handshake, the intermediary is likely to be vulnerable to the same attacks when using the **Upgrade**-based handshake.

2.2.3 CONNECT-based Strawman Handshake

Design Rather than relying upon the rarely used HTTP **Upgrade** mechanism to inform network intermediaries that the remainder of the socket is not HTTP, we consider using HTTP’s **CONNECT** mechanism. Because **CONNECT** is commonly used to establish opaque tunnels to pass TLS traffic through HTTP proxies, transparent proxies are likely to interpret this request as an HTTPS **CONNECT** request, assume the remainder of the socket is unintelligible, and simply route all traffic transparently based on the IP. We create a strawman handshake based on the **CONNECT** mechanism.

Client → *Server*:

```
CONNECT websocket.invalid:443 HTTP/1.1
Host: websocket.invalid:443
```

```
Sec-WebSocket-Key: <connection-key>  
Sec-WebSocket-Metadata: <metadata>
```

Server → Client:

```
HTTP/1.1 200 OK  
Sec-WebSocket-Accept: <hmac>
```

where `<connection-key>` is a 128-bit random number encoded in base64 and `<metadata>` is various metadata about the connection (such as the URL to which the client wishes to open a WebSocket connection). In the server’s response, `<hmac>` is the HMAC of the globally unique identifier `258EAF5-E914-47DA-95CA-C5AB0DC85B11` under the key `<connection-key>` (encoded in base64). By sending the `<hmac>` value, the server demonstrates to the client that it understands and is willing to speak the WebSocket protocol because computing the `<hmac>` value require “knowledge” of an identifier that is globally unique to the WebSocket protocol.

Notice that instead of using the destination server’s host name, we use an invalid host name (per RFC 2606 [36]). Any intermediaries that do not recognize the WebSocket protocol but understand this message according to its HTTP semantics will route the request to a non-existent host and fail the request.

2.2.4 Ad-based Experiment

To evaluate the practicality of mounting IP hijacking and cache poisoning attacks with the WebSocket handshakes, we implemented prototypes for each WebSocket handshake using Flash sockets and a WebSocket server written in Python. We reused the system from the Java and Flash socket experiment with the following changes. We setup a custom multiplexing server at port 80 on the attacking server, which forwards requests to either a standard Apache server or the WebSocket server depending on the request headers. We

	POST-based	Upgrade-based	CONNECT-based
Handshake pass and spoof request ignored	47741	47162	47204
Spoof request routed to target[*]	1376	1	0
Spoof request routed to attacker	97	174	2
Script file cached from target	54519	54526	54534
Script file cached from attacker[†]	15	8	0

Table 2.2: HTTP Host header spoofing via HTML5 WebSocket strawman protocols

^{*} Allows attacker to open a direct socket from the client to an arbitrary server

[†] Allows attacker to poison the HTTP cache of all clients of the proxy

ran an advertisement campaign for four successive days in November 2010, spending \$20 in the Philippines and \$80 globally. Our advertisement contains a SWF which performs the WebSocket handshake, spoofs an HTTP request upon handshake success, and instructs the browser to request a script from the target server using a script tag. We experimented with how intermediaries process each WebSocket handshake. Table 2.2 shows our results.

- **POST-based handshake.** Out of a total of 54,534 impressions, 49,218 (90.2%) succeeded with the POST-based handshake and 5,316 (9.4%) failed. Out of the 49,218 impressions on which we were able to run our IP hijacking test, 47,741 (96.9%) reported that no intermediaries were confused when sending the spoofed HTTP request. However, we found that the IP hijacking attack succeeded on 1,376 of 49,218 impressions (2.8%), where the client was behind a Host-routing proxy. There were 97 of 49,218 impressions (0.2%) where the spoofed request was routed by IP and 4 that received an HTML error. We ran the cache poisoning test on the clients that succeeded with the POST-based handshake, and found 15 successful cache poisoning attacks. These results show that the POST-based handshake is vulnerable to both attacks.
- **Upgrade-based handshake.** We tested how intermediaries in the wild process the Upgrade-based handshake. Out of a total of 54,534 impressions, 47,338 (86.8%)

succeeded with the handshake and 7,196 (13.2%) failed. The handshake failed more often than the `POST`-based handshake, possibly when the `Upgrade` mechanism was unsupported and, perhaps, stripped. Out of the 47,338 impressions on which we were able to run our IP hijacking test, 47,162 (99.6%) did not receive a response after spoofing an HTTP request. We noticed that the IP hijacking attack succeeded on 1 impression, where the client was behind a `Host`-routing proxy. There were 174 of 47,338 impressions (0.37%) where the spoofed request was routed by IP. One impression received an HTML error message.

Out of the 47,338 impressions that succeeded the `Upgrade`-based handshake, we ran the cache poisoning test and found 8 successful cache poisoning attacks. The 8 impressions were also vulnerable to cache poisoning when using the `POST`-based handshake.

- **CONNECT-based handshake.** We tested whether the `CONNECT`-based handshake would resist transparent proxy attacks in the real world. Out of a total of 54,534 impressions, 47,206 (86.6%) succeeded with the handshake and 7,328 (13.4%) failed. Out of the 47,206 impressions on which we were able to run our IP hijacking test, only three did receive a response after spoofing an HTTP request. We observed that the IP hijacking attack did not succeed on any clients. We logged 1 impression that returned an HTML error message. We observed 2 impressions where the spoof request was routed by IP to the attacking server, however none indicated proxy routing based on the `Host` header. It appears that these proxies simply passed the `CONNECT` to our server untouched and then treated the next spoofed request as if it were a separate request routed by IP. We proceeded to the cache poisoning test and did not find successful cache poisoning attacks.

2.3 Securing the WebSocket Protocol

In our experiments, we found successful attacks against both the `POST`-based handshake and the `Upgrade`-based handshake. For the `CONNECT`-based handshake, we observed two proxies which appear not to understand `CONNECT` but simply to treat the request as an ordinary request and then separately route subsequent requests, with all routing based on IP address. Although these proxies did not cache, it is possible that proxies of this type which cache do exist—though our data suggest that they would be quite rare. In this case the attacker would be able to mount a cache poisoning attack.

2.3.1 Our Proposal: Payload Masking

A mitigation for these attacks is to mask all the attacker-controlled bytes in the raw socket data with a stream cipher. The stream cipher is not to provide confidentiality from eavesdroppers but to ensure that the bytes on the wire appear to be chosen uniformly at random to network entities that do not understand the WebSocket protocol, making it difficult for the attacker to confuse the receiver into performing some undesirable action.

We propose masking the metadata in the initial handshake and all subsequent data frames with a stream cipher, such as AES-128-CTR. To key the encryption, the client uses HMAC of the globally unique identifier `C1BA787A-0556-49F3-B6AE-32E5376F992B` with the key `<connection-key>`. However, encrypting the raw socket writes as one long stream is insufficient because the attacker learns the encryption key in the handshake thus can generate inputs to the socket write function that produce ciphertexts of his choice. Instead, we encrypt each protocol frame separately, using a per-frame random nonce as the top part of the CTR counter block, with the lower part being reserved for the block counter. From the perspective of the attacker, this effectively randomizes the data sent on the wire even if the attacker knows the key exchanged in the handshake. Note that

each protocol frame must be encrypted with a fresh nonce and that the browser must not send any bytes on the wire until the browser receives the entire data block from the application. Otherwise, the attacker could learn the nonce and adjust the rest of the input data based on that information.² This mitigation comes at a modest performance cost and some cost in packet expansion for the nonce, which needs to be large enough that the attacker's chance of guessing the nonce is sufficiently low.

In the case that the cost of encryption is a burden, Stachowiak [37] suggests using a simple XOR cipher as a lightweight alternative to using AES-128-CTR. In particular, the client generates a fresh 32 bit random nonce for every frame, and the plaintext is XORed with a pad consisting of the nonce repeated. Because the nonce is unknown to the attacker prior to receiving the corresponding data frame, the attacker is unable to select individual bytes on the wire. However, because the pad repeats, the attacker is able to select correlations between the bytes on the wire, but we are unaware of how to leverage that ability in an attack.

Other proposals with simpler transformations have been discussed in the WebSocket protocol working group, such as flipping the first bit in the frame, or escaping ASCII characters and carriage returns in the handshake. However, these proposals do not protect servers or intermediaries with poor implementation that skip non-ASCII characters. Moreover, using cryptographic masking also mitigates other attack vectors, such as non-HTTP servers that speak protocols with non-ASCII bytes. We believe masking is a more robust solution to these attacks that is more likely to withstand further security analysis.

²A similar condition applies to TLS [5] packet writes.

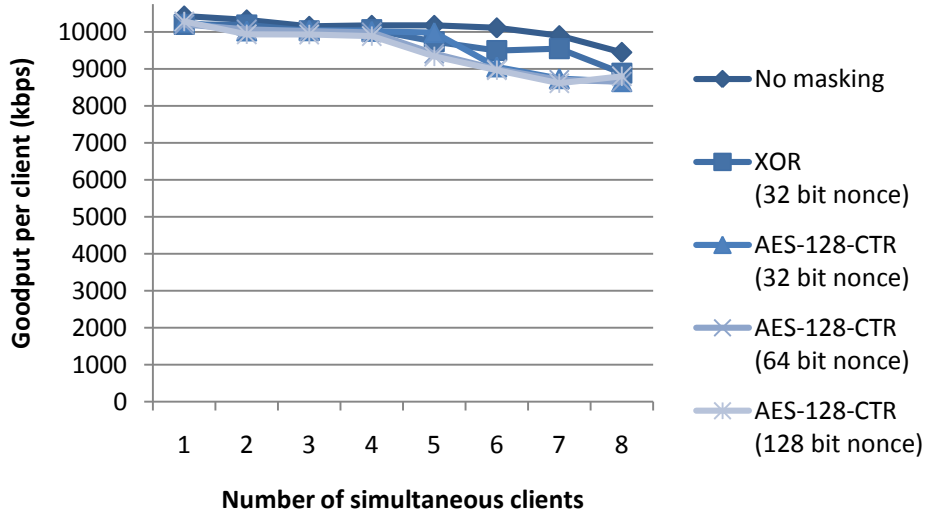


Figure 2.3: Performance of WebSocket with 1,000-byte data frames

2.3.2 Performance Evaluation

We evaluated the network performance of WebSockets using no masking, XOR masking (with 32 bit nonces) and AES-128-CTR masking (with 32, 64 and 128 bit nonces), modified on a Java implementation [38]. From `slicehost.com`, we acquired a 1,024 MB RAM machine as the server with uncapped incoming bandwidth and eight 256 MB RAM machines as the clients, each with 10 Mbps outgoing bandwidth. In our evaluation, we measured the elapsed time for each client to send 10 MB of application data to the server with various frame sizes, while the server handles up to 8 clients simultaneously. Results for sending 1,000 byte data frames, 100 byte data frames and 10 byte data frames are shown in Figure 2.3, Figure 2.4 and Figure 2.5, respectively. We observe that AES-128-CTR masking induces little overhead when the data frame size is as large as 1,000 bytes. However, the performance of AES-128-CTR masking drops off significantly for smaller data frames in comparison with no masking, whereas XOR masking still performs at acceptable speeds.

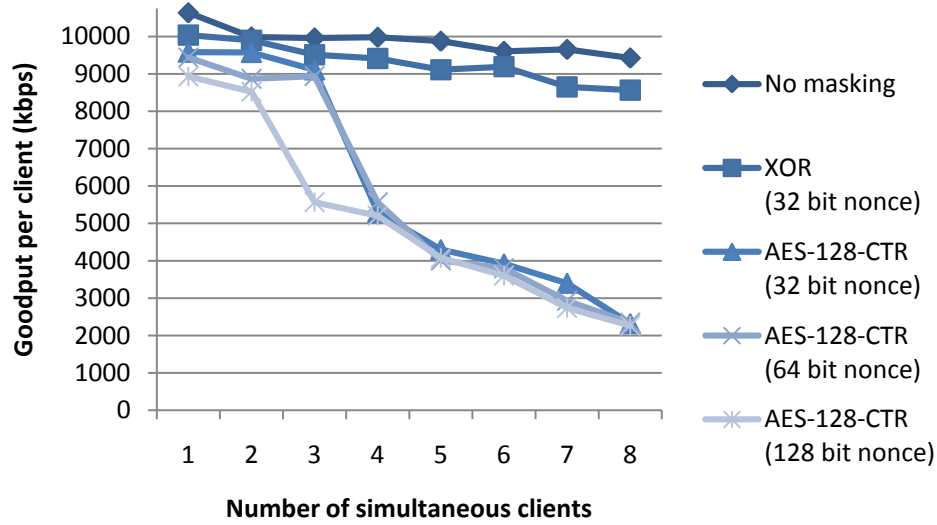


Figure 2.4: Performance of WebSocket with 100-byte data frames

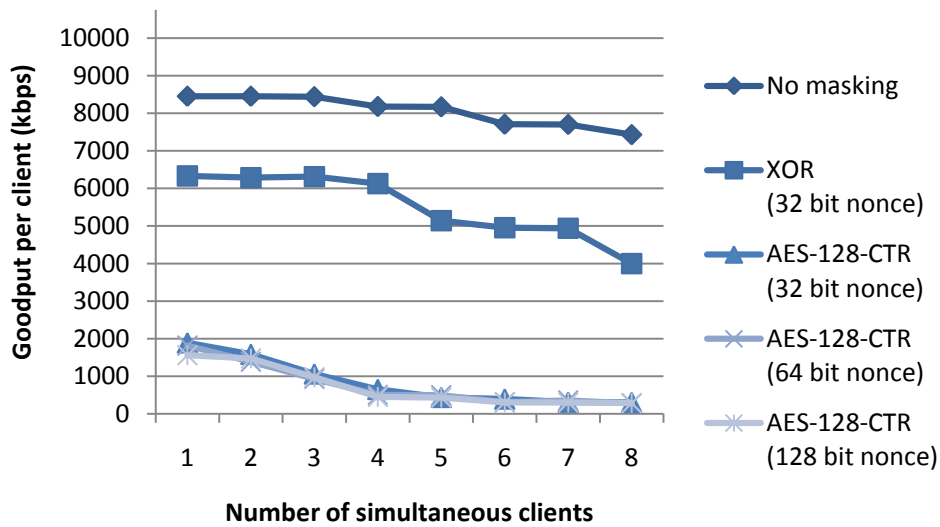


Figure 2.5: Performance of WebSocket with 10-byte data frames

2.3.3 Adoption

We reported the vulnerabilities to the IETF WebSocket protocol working group in November 2010. Due to concerns about these attacks, Firefox [39] and Opera [40] temporarily disabled the WebSocket protocol. In response to our suggestion, the working group reached consensus to prevent the attacker from controlling the bytes sent on the wire by requiring XOR-based masking. Internet Explorer adopted frame masking in their WebSocket prototype using Silverlight plug-in in HTML5 Labs [41]. In 2011, the IETF standardized the WebSocket protocol including the proposed defense in RFC 6455 [42]. We hope to assist the Flash Player and Java plug-ins in addressing these issues in the near future.

2.4 Related Work

2.4.1 Cross-Protocol Attacks

Cross-protocol attacks are used to confuse a server or an intermediary into associating a request with an incorrect protocol. We described an instance of a cross-protocol attack between HTTP and the WebSocket protocol. Topf [43] describes an attack that uses HTML forms to send commands to servers running ASCII based protocols like SMTP, NNTP, POP3, IMAP, and IRC. To prevent these attacks, browsers restrict access to well-known ports of vulnerable applications, such as port 25 for SMTP. This defense cannot be applied to WebSocket because WebSocket operates over port 80, the same port as HTTP, for compatibility. We suspect there are other forms of cross-protocol attacks and expect to address more of these problems in future work.

2.4.2 HTTP Cache Poisoning

Bueno [44] describes an HTTP cache poisoning attack on web pages that rely on the value of the HTTP `Host` header to generate HTML links. In particular, a malicious client sends an HTTP request with a crafted `Host` header, causing the server to rewrite links with an arbitrary string provided by the attacker. If there is any caching going on by proxies along the way, other clients will get the exploited page with injected text. A mitigation for these attacks is to not generate any page content using the `Host` header. In comparison, our cache poisoning attacks do not rely on the usage of `Host` header in the target page, and allow the attacker to poison the proxy’s cache for an arbitrary URL on any target host.

2.4.3 HTTP Response Splitting

In an HTTP response splitting attack [45], the attacker sends a single HTTP request that tricks the benign server into generating an HTTP response that is misinterpreted by the browser or an intermediary as two HTTP responses. Typically, the malicious request contains special line break character sequences, known as line feed (LF) and carriage return (CR), that are reflected by the server into the output stream and appear to terminate the first response, letting the attacker craft the byte sequence that the browser or intermediary interprets as the second response. The attacker can mount a cache poisoning attack by sending a second request to a benign server, which causes the browser or proxy associates with the second “response” and stores in its cache. Servers can prevent the attack by sanitizing data and not allowing CRLF in HTTP response headers. In our work, we introduce new cache poisoning attacks against transparent proxies, which are not addressed by previous mitigations.

Chapter 3

Unauthorized Interceptions

The work in this chapter was done in collaboration with Alex Rice, Erling Ellingsen, and Collin Jackson.

In the previous Chapter, we studied web-based attacks that affect users behind benign, but confused, network intermediaries. Next, we will investigate network attacks which can happen when users are behind malicious network intermediaries.

An SSL man-in-the-middle attack is an interception of an encrypted connection between a client and a server where an active network attacker (described in Section 1.1.2) impersonates the server through a *forged* SSL certificate. We define a forged SSL certificate as an SSL certificate not provided or authorized by the legitimate owner of the website. In TLS, if the network attacker cannot obtain a valid certificate of legitimate websites, browsers should fail to validate the server and trigger TLS certificate warnings. That said, these attacks can still succeed against a significant portion of real-world users who click-through the warnings [46, 47, 48, 49]. Furthermore, in the past, commercial CAs (DigiNotar [30], Comodo [29], and TURKTRUST [50]) have been found to mis-issue fraudulent certificates which would have been accepted by browsers.

Despite that SSL man-in-the-middle attack attempts have previously been spotted in

the wild (e.g., in Iran [18], Syria [19] and even on Tor networks [51]), it is unclear how prevalent these attacks actually are. Several existing SSL surveys [52, 53, 54, 55] have collected large amounts of SSL certificates via scanning public websites or monitoring SSL traffic on institutional networks, yet no significant data on forged SSL certificates have been publicly available. We hypothesize that real attackers are more likely to perform only highly targeted attacks at certain geographical locations, or on a small number of high-value sessions, therefore, previous methodologies would not be able to detect these attacks effectively.

In this Chapter, we first introduce a practical method for websites to detect SSL man-in-the-middle attacks on a large scale, without alterations on the client's end (e.g., customized browsers). We utilized the widely-supported Flash Player plugin to enable socket functionalities not natively present in current browsers, and implemented a partial SSL handshake on our own to capture forged certificates. We deployed this detection mechanism on an Alexa top 10 website, Facebook, which terminates connections through a diverse set of network operators across the world.

3.1 Man-in-the-Middle Attacks

In this section, we describe how the man-in-the-middle attack works, and provide several reasons why detection is difficult.

3.1.1 How it Works

The SSL man-in-the-middle (MITM) attack is a form of active network interception where the attacker inserts itself into the communication channel between the victim client and the server (typically for the purpose of eavesdropping or manipulating private communications). The attacker establishes two separate SSL connections with the client and the

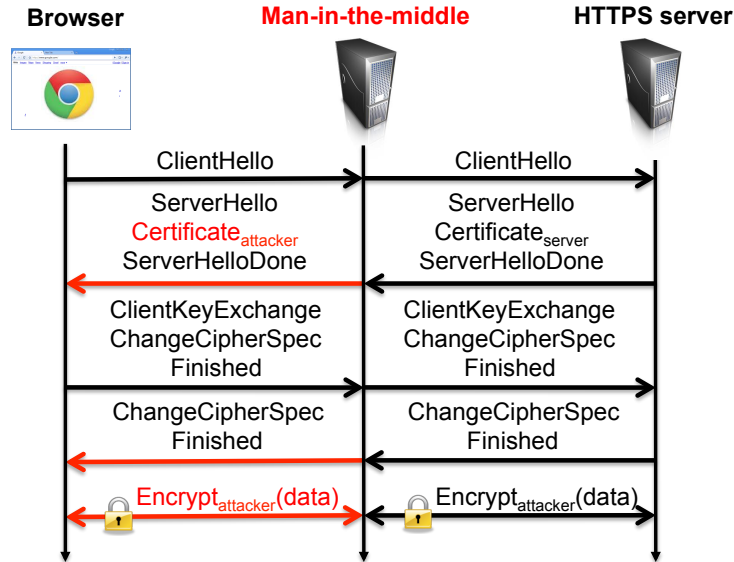


Figure 3.1: An SSL man-in-the-middle attack between the browser and the server, using a forged SSL certificate to impersonate as the server to the client.

server, and relays messages between them, in a way such that both the client and the server are unaware of the middleman. This setup enables the attacker to record all messages on the wire, and even selectively modify the transmitted data. Figure 3.1 depicts an SSL man-in-the-middle attack with a forged certificate mounted between a browser and a HTTPS server. We describe the basic steps of a generic SSL man-in-the-middle attack as follows:

1. The attacker first inserts itself into the transport path between the client and the server, for example, by setting up a malicious WiFi hotspot. Even on otherwise trusted networks, a local network attacker may often successfully re-route all of the client's traffic to itself using exploits like ARP poisoning, DNS spoofing, BGP hijacking, etc. The attacker could also possibly configure itself as the client's proxy server by exploiting auto-configuration protocols (PAC/WPAD) [56]. At this point, the attacker has gained control over the client's traffic, and acts as a relay server

between the client and the server.

2. When the attacker detects an SSL `ClientHello` message being sent from the client, the attacker accurately determines that the client is initiating an SSL connection. The attacker begins the impersonation of the victim server and establishes an SSL connection with the client. Note that the attacker uses a forged SSL certificate during its SSL handshake with the client.
3. In parallel to the previous step, the attacker creates a separate SSL connection to the legitimate server, impersonating the client. Once both SSL connections are established, the attacker relays all encrypted messages between them (decrypting messages from the client, and then re-encrypting them before sending to the server). Now, the attacker can read and even modify the encrypted messages between the client and the server.

As soon as the client accepts the forged SSL certificate, the client's secrets will be encrypted with the attacker's public key, which can be decrypted by the attacker. Numerous automated tools that can mount SSL man-in-the-middle attacks are publicly available on the Internet (e.g., `sslsniff` [57]), which greatly reduce the level of technical sophistication necessary to mount such attacks.

3.1.2 Why is Detection Challenging?

Unfortunately, detecting man-in-the-middle attacks from the website's perspective, on a large and diverse set of clients, is not a trivial task. First of all, most users do not use client certificates, thus servers cannot simply rely on SSL client authentication to distinguish legitimate clients from attackers. Furthermore, there are currently no APIs for a web application to check the certificate validation status of the underlying SSL connection,

not even when an SSL error has occurred on the client. Also, it is currently not possible for web applications to directly access the SSL handshake with native browser networking APIs, like XMLHttpRequest and WebSocket, to validate SSL certificates on their own. Although one could easily develop a program or a custom browser extension that probes SSL certificates as an SSL client, it would not be scalable to distribute additional software to a large number of normal users, especially for non-tech-savvy users. Since professional attackers are more likely to perform only highly targeted attacks at certain geographical locations, or on a small number of high-value sessions, these methodologies would not be able to detect localized attacks effectively.

3.2 Flash-Based Detection Method

In this section, we describe the design and implementation details of our detection method. We further discuss our experimental results and limitations.

3.2.1 Design

In order to determine whether an SSL connection is being intercepted, our fundamental approach is to observe the server's certificate from the client's perspective. Intuitively, if the client actually received a server certificate that does not exactly match the website's legitimate certificate, we would have direct evidence that the client's connection must have been tampered with. Ideally, we would like to develop a JavaScript code snippet to observe SSL certificates, which runs in existing browsers and can reach a large population of clients. However, there are currently no existing browser APIs that allows web applications to directly check the observed server certificate or validation status of their SSL connections. To work around this, we utilized browser plugins to implement a client-side applet that is capable of imitating the browser's SSL handshake, accompanied with the

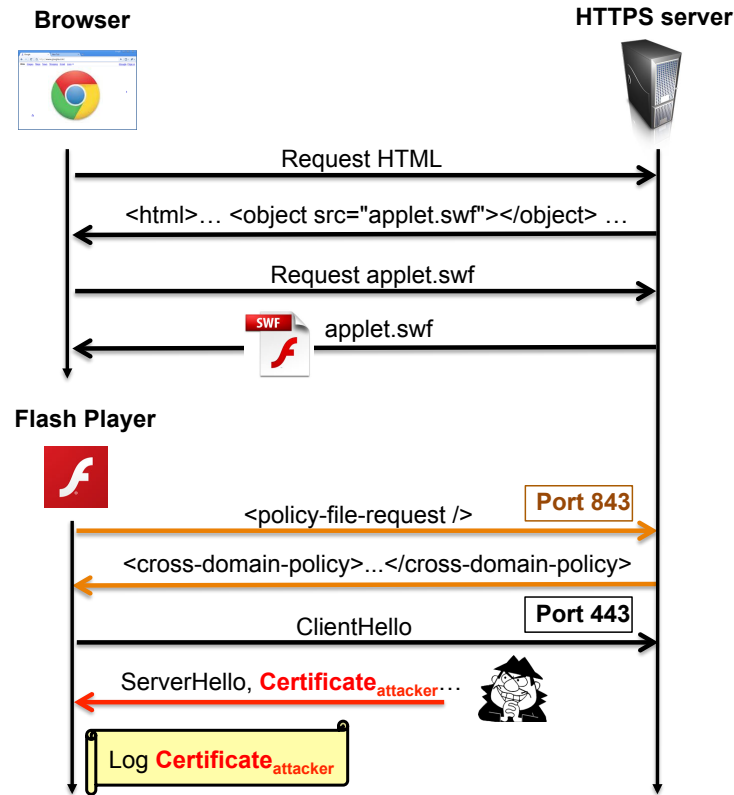


Figure 3.2: Detecting TLS man-in-the-middle attacks with a Flash applet.

ability to report the observed certificate chain. The applet can open a socket connection to the HTTPS server (skipping the browser's network stack), perform an SSL handshake over the socket, record the SSL handshake, and report the certificate chain back to our logging servers, shown in Figure 3.2. We describe our implementation details below.

Client-Side Applet

Our approach is to use a client-side applet that observes the server's SSL certificate from the client's perspective, directly during the SSL handshake. Since native browser networking APIs like XMLHttpRequest and WebSocket do not provide web applications access to raw bytes of socket connections, we must utilize browser plugins. We implemented a Shockwave Flash (SWF) applet that can open a raw socket connection to its own HTTPS

server (typically on port 443), and perform an SSL handshake over the connection in the Flash Player.

By default, the Flash Player plugin does not allow any applets to access socket connections, unless the remote host runs a Flash socket policy server [11]. The Flash socket policy server, normally running on port 843, serves a socket policy file that declares whether SWF applications may open socket connections to the server. Note that even if a SWF file is requesting a socket connection to the same host it was served from, a socket policy server is still required. As a result, in order for a SWF applet from `example.com` to open a socket connection to a HTTPS server `example.com` on port 443, a valid socket policy file must be served at `example.com` on port 843, which permits socket access from `example.com` applications to port 443, as follows (in XML format):

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy SYSTEM
    "http://www.adobe.com/xml/dtds/cross-domain-policy.dtd">
<cross-domain-policy>
    <allow-access-from domain="example.com" to-ports="443" />
</cross-domain-policy>
```

Note that the socket policy file should not be confused with the `crossdomain.xml` file served by web servers, which restricts access to HTTP, HTTPS, and FTP access, but not socket access. If the Flash Player cannot successfully retrieve a valid socket policy (e.g., blocked by a firewall), the socket connection will be aborted and an exception will be thrown.

Once the socket connection is permitted, our applet will initiate an SSL handshake by sending a `ClientHello` message over the socket, and wait for the server to respond with the `ServerHello` and `Certificate` messages, which will be recorded. To support clients behind explicit HTTP proxies, the applet may send a `CONNECT` request over the socket to create an SSL tunnel prior to sending the `ClientHello` message, as follows:

```
CONNECT example.com:443 HTTP/1.1
```

Our SSL handshake implementation was based on the SSL 3.0 protocol version. Since our goal to observe the server's certificate chain, our applet closes the socket connection after successfully receiving the certificate chain. Lastly, our applet converts the raw bytes of the recorded SSL handshake responses into an encoded string, and sends it back to our log server with a POST request.

We note that the Flash Player plugin is currently supported on 95% of web browsers [58], therefore, our applet should be able to run on most clients. In fact, one of the major browsers, Google Chrome, has the Flash Player plugin built in by default. Also, SWF applets are usually allowed to execute without any additional user confirmation, and do not trigger any visual indicators (e.g., system tray icons) while running, thus, deploying this method should not affect the visual appearance of the original web page.

Alternatively, the client-side applet may be implemented using other browser plugins, for example, the Java plugin. Java applets are allowed to create socket connections from the client to any port on the same host that the applet was served from. As an example, an applet served from port 80 on `example.com` can open a raw socket to port 443 on `example.com` without requesting any additional access. However, due to security concerns, the Java plugin is currently blocked by default on several client platforms, and may require additional user interaction to activate the Java plugin. Such user interaction would be too obtrusive for our experiment and client diversity suffers greatly once all potential interactive platforms are removed from the experiment. Another side effect of running a Java applet on some platforms is that a visible icon would be displayed in the system tray, which might annoy or confuse some of the website's users.

Lenient Certificate Parsing

Since we implemented the SSL handshake process on our own, we must extract the SSL certificates from a raw byte dump of the SSL handshake observed on the client, by parsing the `ServerHello` and `ServerCertificate` messages. Surprisingly, in our initial attempts, we found that this seemingly straightforward extraction process failed occasionally. By manual inspection, we noticed that some of the recorded SSL messages were slightly different from the SSL/TLS standards. As a result, we intentionally parsed the SSL handshake in as lenient a manner as possible in order to extract certificates even if the SSL message format did not conform exactly to the standards. We did not discard these malformed handshakes as we theorize that they are caused by either transmission errors or software errors in the intercepting proxy.

Websites may choose to perform certificate extraction on-the-fly in the client-side applet, or simply send the handshake raw bytes to their log servers for post-processing. We took the latter approach, since it enabled us to preserve the SSL handshake bytes for further investigation, even if early versions of our extraction code failed (or even crashed unexpectedly) while parsing certificates.

3.2.2 Implementation

We have implemented our client-side applets for both the Flash Player and Java plugins. With similar functionality, the SWF file (2.1 KB) was slightly smaller than the Java applet (2.5 KB). Since Flash Player was supported on a larger client population and is considered less obtrusive to users, we deployed the SWF file for our experiments.

To observe SSL connections on a large set of real-world clients, we deployed our client-side applet on Facebook's servers to run our experiments. We sampled a small portion (much less than 1%) of the total connections on Facebook's desktop website, particularly

on the `www.facebook.com` domain. To avoid affecting the loading time of the website’s original web pages, our applets are programmed to load several seconds after the original page has completed loading. This is done by using a JavaScript snippet that dynamically inserts a HTML object tag that loads the SWF file into the web page visited by the user. Basically, the script is triggered only after the original page finishes loading, and further waits a few seconds, before actually inserting the applet. Additionally, we built server-side mechanisms to allow granular control over the sampling rates in specific countries or networks. This granularity enables us to increase the sampling rate for certain populations in response to the detection of a specific attack.

To support Flash-based socket connections used by our SWF files, we have set up Flash socket policy servers that listens on port 843 of the website, which are configured with a socket policy file that allows only its own applets to open socket connections to port 443. We also setup a logging endpoint on the HTTPS servers, in PHP, that parses the reports, and aggregates data into our back-end databases. The extracted SSL certificates were processed and read using the OpenSSL library. In addition, we built an internal web interface for querying the log reports.

3.2.3 Experimental Results

Using the Flash-based detection method, we conducted the first large-scale experiment in an attempt to catch forged SSL certificates in the wild. We served our client-side applet to a set of randomly sampled clients on Facebook’s website. We collected and analyzed data from November 20, 2012 to March 31, 2013.¹

First of all, we noticed that only a portion of the sampled clients actually completed our detection procedure, explained below. As shown in Table 3.1, a total of 9,179,453

¹Personally identifiable information (IP addresses and HTTP cookies) were removed from our database after a 90-day retention period.

Procedure	Count
1. Inserted HTML object tag into web page	9,179,453
2. Downloaded SWF file from server	6,908,675
3. Sent report to logging server	5,415,689

Table 3.1: Number of clients that completed each step of the detection procedure

page views on Facebook’s desktop website had our HTML object tag dynamically inserted. Our web servers logged 6,908,675 actual downloads for the SWF file. The download count for the SWF file was noticeably lower than the number of object tags inserted. We reason that this is possibly due to: (1) the Flash Player plugin was not enabled on the client, (2) a few legacy browsers did not support our SWF object embedding method, or (3) the user navigated away from the web page before the object tag was loaded. Our log servers received a total of 5,415,689 reports from applets upon successful execution. Again, the number of received reports is lower than the number of SWF file downloads. This is likely due to the web page being closed or navigated away by the user, before the applet was able to finish execution.

Next, we noticed that only 64% out of the 5,415,689 received reports contained complete and well-formed certificate records, as shown in Table 3.2. We observed that 1,965,186 (36%) of the reported data indicated that the client caught `SecurityErrorEvent` or `IOErrorEvent` exceptions in the Flash Player and failed to open a raw socket. We believe that most of these errors were caused by firewalls blocking the socket policy request (for example, whitelisting TCP ports 80 and 443 to only allow web traffic), thus not allowing the Flash Player to retrieve a valid socket policy file from our socket policy servers (over port 843). For clients behind these firewalls, we were not able to open socket connections using Flash Player, although using Java might have worked in some legacy client platforms. We discuss in Section 3.2.4 that similar measurements can be conducted on native

Type	Count
Well-formed certificates	3,447,719 (64%)
Flash socket errors	1,965,186 (36%)
Empty reports	2,398 (0%)
Bogus reports	290 (0%)
HTTP responses	96 (0%)

Table 3.2: Categorization of reports

mobile platforms to avoid the drawbacks of Flash sockets.

In addition to the Flash socket errors, we also observed a few other types of erroneous reports. There were 2,398 reports that were empty, indicating that the SWF file failed to receive any certificates during the SSL handshake. This might have been caused by firewalls that blocked SSL traffic (port 443). There were 96 reports that received HTTP responses during the SSL handshake, mostly consisting of error pages (HTTP 400 code) or redirection pages (HTTP 302 code). These responses suggest that some intercepting proxies contained logic that were modifying the client's web traffic to block access to certain websites (or force redirection to certain web pages, known as captive portals). We found that some clients received a HTML page in plaintext over port 443, for instance, linking to the payment center of Smart Bro, a Philippine wireless service provider. These type of proxies do not appear to eavesdrop SSL traffic, but they inject unencrypted HTTP responses into the client's web traffic.

In addition, there were 290 reports that contained garbled bytes that could not be correctly parsed by our scripts. Although we could not successfully parse these reports, manual inspection determined that 16 of the reports contained seemingly legitimate VeriSign certificates that had been truncated in transit, presumably due to lost network connectivity. Another 37 of these reports appear to be issued by Kurupira.NET, a web filter, which closed our SSL connections prematurely. We also found that 17 of the unrecognized POST

requests on our log servers were sent from a Chrome extension called Tapatalk Notifier (determined by the HTTP `origin` header), however we have no evidence that these false POST requests were intentional.

Finally, we successfully extracted 3,447,719 (64%) well-formed certificates from the logged reports. We used custom scripts (mentioned in Section 3.2.1) to parse the recorded SSL handshake bytes. A total of 3,440,874 (99.8%) out of 3,447,719 observed certificates were confirmed to be the website’s legitimate SSL certificates, by checking the RSA public keys (or more strictly, by comparing the observed certificate bit-by-bit with its legitimate certificates). We note that there were multiple SSL certificates (thus, multiple RSA public keys) legitimately used by Facebook’s SSL servers during the period of our study, issued by publicly-trusted commercial CAs including VeriSign, DigiCert, and Equifax. Most interestingly, we discovered that 6,845 (0.2%) of the observed certificates were not legitimate, nor were they in any way approved by Facebook. We further examine these captured forged certificates in Section 3.3.

3.2.4 Limitations

Before we move on, we offer insights on the limitations of our detection method. It is important to point out that the goal of our implementation was not to evade the SSL man-in-the-middle attacks with our detection mechanism. Admittedly, it would be difficult to prevent professional attackers that are fully aware of our detection method. We list below some ways that an attacker might adaptively evade our detection:

- Attackers may corrupt all SWF files in transmission, to prevent our client-side applet from loading. However, this approach would cause many legitimate applications using SWF files to break. Of course, the attacker could narrow the scope of SWF blacklisting to include only the specific SWF files used in this detection. In response,

websites may consider randomizing the locations of their SWF files.

- Attackers may restrict Flash-based sockets by blocking Flash socket policy traffic on port 843. To counter this, websites could possibly serve socket policy files over firewall-friendly ports (80 or 443), by multiplexing web traffic and socket policy requests on their servers. In addition, websites could try falling back to Java applets on supporting clients if Flash-based sockets are blocked.
- Attackers may try to avoid intercepting SSL connections made by the Flash Player. However, the website may tailor its client-side applet to act similarly to a standard browser.
- In theory, attackers could possibly tamper the reports (assuming that the measured client was under an SSL man-in-the-middle attack, and probably clicked through SSL warnings, if any), and trick our log servers to believe that the website's legitimate certificate was observed. Under this scenario, the website may need additional mechanisms to verify the integrity of their reports.

At the time of this study, there is no reason to think that any attacker is tampering our reports, or even aware of our detection method. We do not consider attackers that have obtained access to Facebook's internal servers. As shown in Section 3.2.3, our current methodology has successfully captured direct evidences of unauthorized SSL interceptions in the wild. However, if more websites become more aggressive about this sort of monitoring, we might get into an arms race, unfortunately.

Fortunately, many popular websites nowadays have the option to leverage their native mobile applications for detecting attacks. While our initial implementation targeted desktop browsers, we suggest that similar mechanisms can be implemented, more robustly,

on mobile platforms such as iOS and Android.² Native mobile applications have the advantage of opening socket connections without Flash-based socket policy checks, and are more difficult for network attackers to bypass (since the Flash applet is no longer necessary, and native applications can be programmed to act exactly like a standard browser). Furthermore, mobile clients can also implement additional defenses (e.g., certificate pinning [59]) to harden itself against SSL man-in-the-middle attacks (e.g., preventing the tampering of reports), while performing similar measurement experiments.

3.3 Analysis of Forged SSL Certificates

From the experiments in Section 3.2.3, we collected 6,845 forged certificates from real-world clients connecting to Facebook’s SSL servers. In this section, we analyze the root cause of these injected forged SSL certificates.

3.3.1 Certificate Subjects

First, Table 3.3 shows the subject organizations of forged certificates. As expected, the majority of them spoofed the organization as Facebook. There were over a hundred forged certificates that excluded the organization attribute entirely. Again, we confirmed 93 certificates that were attributed to Fortinet Ltd.

Next, we inspect the observed subject common names of the forged SSL certificates, summarized in Table 3.4. Normally, the subject common name of the SSL certificate should match the hostname of the website to avoid triggering SSL certificate warnings in the browser. While most of the forged certificates used the legitimate website’s domains as the subject common name, there were a few certificates that used unrelated domains as well.

²After our initial study, Facebook has implemented our methodology across their native mobile applications.

Subject Organization	Count
Facebook, Inc.	6,552
<i>Empty</i>	131
Fortinet Ltd. / Fortinet	93
Louisville Free Public Library	10
<i>Other</i>	59

Table 3.3: Subject organizations of forged certificates

Unsurprisingly, most of the forged SSL certificates used the wildcard domain `*.facebook.com` as the subject common name in order to avoid certificate name validation errors. This suggests that most of the attacking entities were either specifically targeting Facebook’s website by pre-generating certificates that match the website’s name, or using automated tools to generate the certificates on-the-fly. None of the forged certificates were straight clones of Facebook’s legitimate certificates (that replicated all the X.509 extension fields and values). There were some certificates that used IP addresses as common name, for example, `69.171.255.255` (which appears to be one of Facebook’s server IP addresses). We noticed that a number of forged certificates used a subject name that starts with two characters `FG` concatenated with a long numeric string (e.g., `FG600B3909600500`). These certificates were issued by Fortinet Ltd., a company that manufactures SSL proxy devices which offer man-in-the-middle SSL inspection. Similarly, we found 8 certificates that had a subject common name “labris.security.gateway SSL Filtering Proxy,” which is also an SSL proxy device. There were a few other common names observed that were likely amateur attempts of SSL interception, such as `localhost.localdomain`, which is the default common name when generating a self-signed certificate using the OpenSSL library.

For the forged SSL certificates that did not use a subject common name with `facebook.com` as suffix, we also checked if any subject alternative names were present in the certificate. Subject alternative names are treated as additional subject names, and allow certificates

Subject Common Name	Count
*.facebook.com	6,491
www.facebook.com	117
pixel.facebook.com	1
m.facebook.com	1
facebook.com	1
*	1
<i>IP addresses</i>	118
FG... / Fortinet / FortiClient	93
<i>Other</i>	22

Table 3.4: Subject common names of forged certificates

to be shared across multiple distinct hostnames. This may allow attackers to generate a single forged certificate for attacking multiple different websites. For the 233 forged certificates that did not provide a matching common name, none of them provided a matching subject alternative name. Even though these 233 (3.4%) forged certificates would definitely trigger name mismatch errors, there is still a significant possibility that users may ignore the browser’s security warnings anyway.

3.3.2 Certificate Issuers

We examine the issuer organizations and issuer common names of each forged SSL certificate. Table 3.5 lists the top issuer organizations of the forged certificates. At first glance, we noticed several forged certificates that fraudulently specified legitimate organizations as the issuer, including 5 using Facebook, 4 using Thawte, and one using VeriSign. These invalid certificates were not actually issued by the legitimate companies or CAs, and were clearly malicious attempts of SSL interception. Since 166 of the forged certificates did not specify its issuer organization (or *empty*), we also checked the issuer common names, listed in Table 3.6.

We manually categorized the certificate issuers of forged certificates into antivirus,

Issuer Organization	Count
Bitdefender	2,682
ESET, spol. s r. o.	1,722
BullGuard Ltd.	819
Kaspersky Lab ZAO / Kaspersky Lab	415
Sendori, Inc	330
<i>Empty</i>	166
Fortinet Ltd. / Fortinet	98
EasyTech	78
NetSpark	55
Elitecore	50
ContentWatch, Inc	48
Kurupira.NET	36
Netbox Blue / Netbox Blue Pty Ltd	25
Qustodio	21
Nordnet	20
Target Corporation	18
DefenderPro	16
ParentsOnPatrol	14
Central Montcalm Public Schools	13
TVA	11
Louisville Free Public Library	10
Facebook, Inc.	5
thawte, Inc.	4
Oneida Nation / Oneida Tribe of WI	2
VeriSign Trust Network	1
<i>Other</i> (104)	186

Table 3.5: Issuer organizations of forged certificates

Issuer Common Name	Count
Bitdefender Personal CA.Net-Defender	2,670
ESET SSL Filter CA	1,715
BullGuard SSL Proxy CA	819
Kaspersky Anti-Virus Personal Root Certificate	392
Sendori, Inc	330
lopFailZeroAccessCreate	112
...	
*.facebook.com	6
VeriSign Class 4 Public Primary CA	5
Production Security Services	3
Facebook	1
thawte Extended Validation SSL CA	1
<i>Other (252)</i>	794

Table 3.6: Issuer common names of forged certificates

firewalls, parental control software, adware, and malware. Notably, we observed an intriguing issuer named **lopFailZeroAccessCreate** that turned out to be produced by malware, which we discuss in detail below.

- **Antivirus.** By far the top occurring issuer was Bitdefender with 2,682 certificates, an antivirus software product which featured a “Scan SSL” option for decrypting SSL traffic. According to their product description, Bitdefender scans SSL traffic for the presence of malware, phishing, and spam. The second most common issuer was ESET with 1,722 certificates, another antivirus software product that provides SSL decryption capabilities for similar purposes. Several other top issuers were also vendors of antivirus software, such as BullGuard, Kaspersky Lab, Nordnet, DefenderPro, etc. These software could possibly avoid triggering the browser’s security errors by installing their self-signed root certificates into the client’s system. Note that the observed antivirus-related certificate counts are not representative of the general antivirus usage share of the website’s users, since SSL interception is

often an optional feature in these products. However, if any antivirus software enabled SSL interception by default, we would expect a higher number of their forged certificates observed.

Supposing that these users intentionally installed the antivirus software on their hosts, and deliberately turned on SSL scanning, then these antivirus-generated certificates would be less alarming. However, one should be wary of professional attackers that might be capable of stealing the private key of the signing certificate from antivirus vendors, which may essentially allow them to spy on the antivirus' users (since the antivirus' root certificate would be trusted by the client). Hypothetically, governments could also compel antivirus vendors to hand over their signing keys.

- **Firewalls.** The second most popular category of forged certificates belongs to commercial network security appliances that perform web content filtering or virus scanning on SSL traffic. As observed in the certificate subject fields, Fortinet was one of the issuers that manufactures devices for web content filtering with support for HTTPS deep inspection. NetSpark was another web content filtering device manufacturer offering similar capabilities. According to their product description, the user's content is unencrypted for inspection on NetSpark's servers, and then re-encrypted under NetSpark's SSL certificate for the end user. We observed a number of device vendors that provided similar devices, such as EliteCore, ContentWatch, and Netbox Blue. There were also software solutions that provided selective website blocking, such as Kurupira.NET. Some appliance vendors aggressively marketed SSL content inspection as a feature which cannot be bypassed by users. For example, ContentWatch's website provided the following product description for their firewall

devices.³

“This technology also ensures the users cannot bypass the filtering using encrypted web traffic, remote proxy servers or many of the other common methods used circumvent content filters.”

Interestingly, EliteCore’s Cyberoam appliances have previously been discovered [60] to be using the same CA private key across all Cyberoam devices. This is particularly dangerous, since the universal CA private key can be extracted from any single device by an attacker. This vulnerability allows an attacker to seamlessly perform SSL man-in-the-middle attacks against users of benign Cyberoam devices, because the attacker can issue forged server certificates that will be accepted by other clients that have installed Cyberoam’s CA certificate. Reportedly, Cyberoam issued an over-the-air patch to generate unique CA certificates on each device. Nevertheless, we should be aware that other device manufacturers are likely to introduce similar security vulnerabilities.

- **Adware.** We observed 330 instances of forged certificates issued by a company named Sendori. This company offers a browser add-on that claims to automatically correct misspelled web pages. However, using Google Search to query the string “Sendori” revealed alarming discussions about the add-on actually hijacking DNS entries for the purposes of inserting advertisements into unrelated websites.⁴ This form of adware actively injects content into webpages, and could possibly be detected using Web Tripwires or CSP (as described in Section 3.4.1).

³<http://www.contentwatch.com/solutions/industry/government>

⁴<http://helpdesk.nwciowa.edu/index.php?/News/NewsItem/View/10>

- **Malware.** As previously mentioned, we noticed that an unknown issuer named `lopFailZeroAccessCreate` appeared relatively frequently in our dataset. We manually searched the name on the Internet and noticed that multiple users were seeing SSL certificate errors of the same issuer, and some were suggesting that the user could be under SSL man-in-the-middle attacks by malware.⁵ Upon deeper investigation, we discovered 5 forged certificates that shared the same subject public key as `lopFailZeroAccessCreate`, yet were generated with their issuer attribute set as “VeriSign Class 4 Public Primary CA.” We confirmed with Symantec/VeriSign that these suspicious certificates were not issued through their signing keys. This was obviously a malicious attempt to create a certificate with an issuer name of a trusted CA. These variants provide clear evidence that attackers in the wild are generating certificates with forged issuer attributes, and even increased their sophistication during the time frame of our study.

In Figure 3.3, we illustrate the geographic distribution of the certificates issued by `lopFailZeroAccessCreate` (and the forged “VeriSign Class 4 Public Primary CA”) on a world map. As shown, the infected clients were widespread across 45 different countries. The countries with the highest number of occurrences were Mexico, Argentina and the United States, with 18, 12, and 11 occurrences, respectively. This shows that the particular SSL man-in-the-middle attack is occurring globally in the wild. While it is possible that all of these attacks were amateur attackers individually mounting attacks (e.g., at their local coffee shop), it is certainly odd that they happened to use forged certificates with the same subject public key. However, this is not so unreasonable if these attacks were mounted by malware. Malware researchers at Facebook, in collaboration with the Microsoft Security Essentials

⁵<http://superuser.com/q/421224>

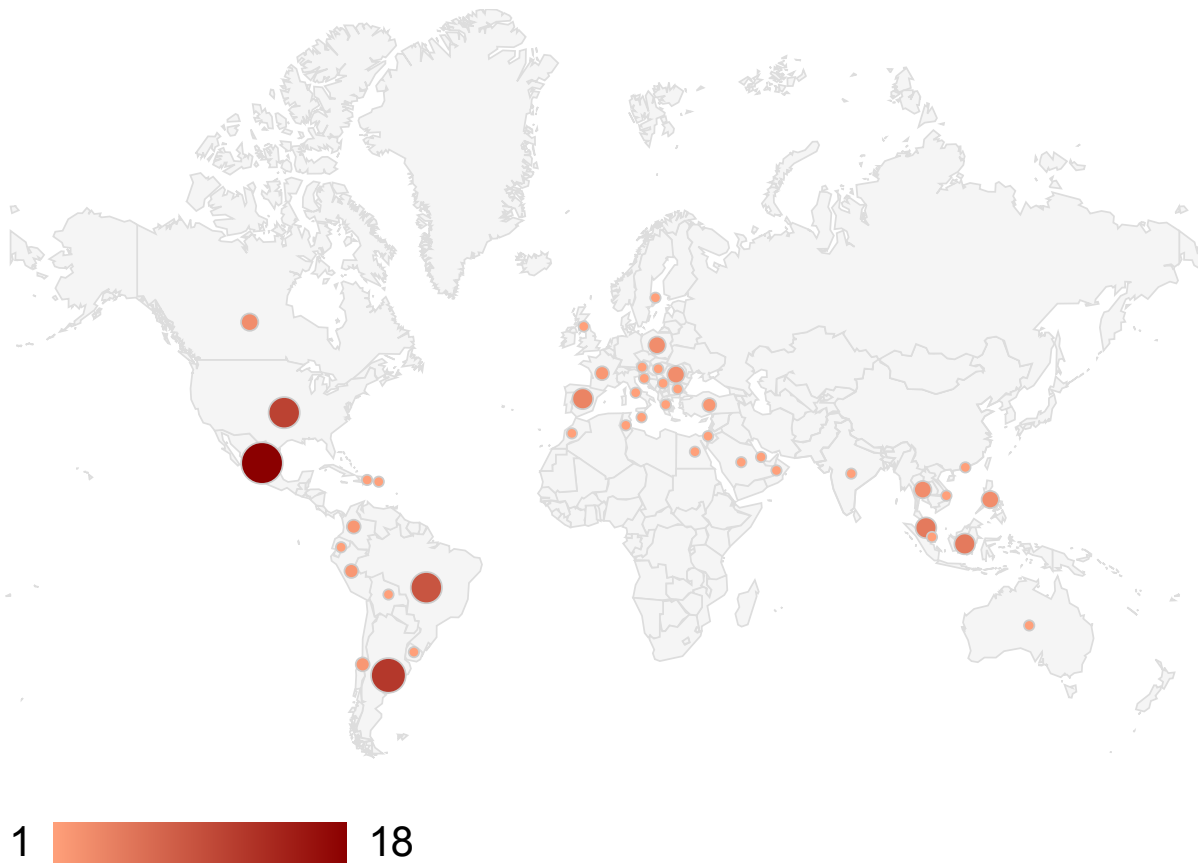


Figure 3.3: Geographic distribution of forged SSL certificates generated by the malicious issuer `lopFailZeroAccessCreate`

team, were able to confirm these suspicions and identify the specific malware family responsible for this attack. Since our experiments only tested SSL connections to Facebook’s servers (only for the `www.facebook.com` domain), we cannot confirm whether this attack also targeted other websites. In response to our discovery, the website notified the infected users, and provided them with malware scan and repair instructions.

In addition, there were 4 other suspicious certificates issued under the organization name of `thawte, Inc` with three of them using “Production Security Services” as the issuer common name, and one using “thawte Extended Validation SSL CA.” These

instances could be the same malware attack previously spotted by some Opera users [61], in which forged certificates pretending to be issued by Thwate were observed. These 4 forged certificates were observed in Italy, Spain, and the United States.

We note that a sophisticated attacker utilizing malware could install their self-signed CA certificates on clients in order to suppress browser security errors. Such an attacker is likely capable of stealing confidential information, by reading from protected storage or logging the user's keystrokes. Nevertheless, mounting an SSL man-in-the-middle attack further enables a general way of capturing and recording the victim's web browsing activities in real-time.

- **Parental Control Software.** Some forged SSL certificates were issued by parental control software, including 21 from Qustodio and 14 from ParentsOnPatrol. These type of software are designed to enable parents to monitor and filter the online activities of their children. Whether such level of control is appropriate is beyond the scope of our work.

While the remaining 104 other distinct issuer organizations in Table 3.5 and 252 other distinct common names in Table 3.6 do not appear to be widespread malicious attempts (based on manual inspection), the possibility remains that some may still be actual attacks.

3.4 Related Work

3.4.1 Webpage Tamper Detection

There are existing proposals aiming to assist websites in detecting whether their unencrypted webpages have been tampered with in transit. We focus on detection methods

that do not require user interaction, and do not require the installation of additional software or browser extensions on the clients' machines.

- **Web Tripwires.** Web Tripwires [62] is a technique proposed to ensure data integrity of web pages, as an alternative to HTTPS. Websites can deploy JavaScript to the client's browser that detects modifications on web pages during transmission. In their study of real-world clients, over 1% of 50,000 unique IP addresses observed altered web pages. Roughly 70% of the page modifications were caused by user-installed software that injected unwanted JavaScript into web pages. They found that some ISPs and enterprise firewalls were also injecting ads into web pages, or benignly adding compression to the traffic. Interestingly, they spotted three instances of client-side malware that modified their web pages. Web Tripwires was mainly designed to detect modifications to unencrypted web traffic. By design, Web Tripwires does not detect passive eavesdropping (that does not modify any page content), nor does it detect SSL man-in-the-middle attacks. In comparison, our goal is to be able to detect eavesdropping on encrypted SSL connections.
- **Content Security Policy.** Content Security Policy (CSP) [63] enables websites to restrict browsers to load page content, like scripts and stylesheets, only from a server-specified list of trusted sources. In addition, websites can instruct browsers to report CSP violations back to the server with the `report-uri` directive. Interestingly, CSP may detect untrusted scripts that are injected into the protected page, and report them to websites. Like Web Tripwires, CSP does not detect eavesdropping on SSL connections.

3.4.2 Certificate Observatories

A number of SSL server surveys [52, 53, 54, 55] have analyzed SSL certificates and certificate authorities on the Internet. The EFF SSL Observatory [52] analyzed over 1.3 million unique SSL certificates by scanning the entire IPv4 space, and indicated that 1,482 trusted certificate signers are being used. Similarly, Durumeric et al. [55] collected over 42 million unique certificates by scanning 109 million hosts, and identified 1,832 trusted certificate signers. Holz et al. [53] analyzed SSL certificates by passively monitoring live SSL traffic on a research network in addition to actively scanning popular websites, and found that over 40% certificates observed were invalid due to expiration, incorrect host names, or other reasons. Akhawe et al. [54] analyzed SSL certificates by monitoring live user traffic at several institutional networks, and provided a categorization of common certificate warnings, including server mis-configurations and browser design decisions. However, existing studies do not provide insights on forged certificates, probably because network attackers are relatively rare on those research institutional networks. In our work, we set out to measure real-world SSL connections from a large and diverse set of clients, in an attempt to find forged SSL certificates.

3.4.3 Certificate Validation with Notaries

Perspectives [64] is a Firefox add-on that compares server certificates against multiple notaries (with different network vantage points) to reveal inconsistencies. Since public notaries observe certificates from diverse network perspectives, a local impersonation attack could be easily detected. Convergence [65] extends Perspectives by anonymizing the certificate queries for improved privacy, while allowing users to configure alternative verification methods (such as DNSSEC). The DetecTor [66] project (which extends Doublecheck [67]) makes use of the distributed Tor network to serve as external notaries.

Crossbear [68] further attempts to localize the attacker’s position in the network using notaries. However, notary approaches might produce false positives when servers switch between alternative certificates, and clients may experience slower SSL connection times due to querying multiple notaries during certificate validation. Further, these pure client-side defenses have not been adopted by mainstream browsers, thus cannot protect the majority of (less tech-savvy) users.

3.4.4 HTTP Strict Transport Security

HTTP Strict Transport Security (HSTS) [69], the successor of ForceHTTPS [70], is a HTTP response header that allows websites to instruct browsers to make SSL connections mandatory on their site. By setting the HSTS header, websites may prevent network attackers from performing SSL stripping attacks [71] (relatedly, Nikiforakis et al. [72] proposed HProxy to detect SSL stripping attacks on clients by checking the security characteristics of a SSL site against previous visits in browser history). A less obvious security benefit of HSTS is that browsers simply hard-fail when seeing invalid certificates, and do not give users the option to ignore SSL errors. This feature prevents users from accepting untrusted certificates when under man-in-the-middle attacks by amateur script kiddies.

3.4.5 Certificate Pinning

The Public Key Pinning Extension for HTTP (HPKP) [73] proposal allows websites to declare a set of legitimate certificate public keys with a HTTP header, and instruct browsers to reject future connections with unknown certificate public keys. HPKP provides protection against SSL man-in-the-middle attacks that use unauthorized, but possibly trusted, certificates. HPKP automatically rejects fraudulent certificates even if they would be

otherwise trusted by the client. Both HSTS and HPKP defenses require that clients must first visit the legitimate website securely before connecting from untrusted networks. This requirement is lifted if public key pins are pre-loaded in the browser, such as in Google Chrome [74] and Internet Explorer (with EMET) [75], although this approach may not scale for the entire web. Notably, Chrome’s pre-loaded public key pinning mechanism has successfully revealed several high-profile CA incidents, in which mis-issued certificates were used to attack Google’s domains in the wild. However, in current implementations, Chrome’s public key pinning does not reject certificates that are issued by a locally trusted signer, such as antivirus, corporate surveillance, and malware.

A related proposal, Trust Assertions for Certificate Keys (TACK) [76], allows SSL servers to pin a server-chosen signing key with a TLS extension. In contrast with HPKP, TACK pins a signing key that chosen by the server, separate from the private key corresponding to the server’s certificate, and can be short-lived. TACK allows websites with multiple SSL servers and multiple public keys to pin the same signing key. Once the browser receives a TACK extension from an SSL site, it will require future connections to the same site to be signed with the same TACK signing key, or otherwise, reject the connection. Another proposal called DVCert [77] delivers a list of certificate pins over a modified PAKE protocol in an attempt to detect SSL man-in-the-middle attacks, but also requires modifications to existing clients and servers.

The concept of public key pinning (or certificate pinning) has previously been implemented as a pure client-side defense as well. Firefox add-ons such as Certificate Patrol [78] and Certlock [79] were designed to alarm users when a previously visited website starts using a different certificate. However, without explicit signals from the server, it may be difficult to accurately distinguish real attacks from legitimate certificate changes, or alternative certificates.

3.4.6 Certificate Audit Logs

Several proposals have suggested the idea of maintaining cryptographically irreversible records of all the legitimately-issued certificates, such that mis-issued certificates can be easily discovered, while off-the-record certificates are simply rejected. Sovereign Keys [80] requires clients to query public timeline servers to validate certificates. Certificate Transparency (CT) [81] removes the certificate queries from clients by bundling each certificate with an audit proof of its existence in the public log. Accountable Key Infrastructure (AKI) [82] further supports revocation of server and CA keys. These defenses are designed to protect against network attackers (not including malware). However, browsers need to be modified to support the mechanism, and changes (or cooperation) are needed on the CAs or servers to deliver the audit proof. Encouragingly, Google has announced their plan to use Certificate Transparency for all EV certificates in the Chrome browser [83].

3.4.7 DNS-based Authentication

DNS-based Authentication of Named Entities (DANE) [84] allows the domain operator to sign SSL certificates for websites on its domain. Similar to public key pinning defenses, DANE could allow websites to instruct browsers to only accept a pre-defined set of certificates. This approach prevents any CAs (gone rogue) from issuing trusted certificates for any domain on the Internet. Another related proposal, the Certification Authority Authorization (CAA) [85] DNS records, can specify that a website's certificates must be issued under a specific CA. However, these approaches fundamentally rely on DNSSEC to prevent forgery and modification of the DNS records. Until DNSSEC is more widely deployed on the Internet, websites must consider alternative defenses.

Chapter 4

Prefetching and Prevalidating Certificates

The work in this chapter was done in collaboration with Emily Stark, Dinesh Israni, Collin Jackson, and Dan Boneh.

The standard TLS handshake requires two round trips before a client or server can send application data. The network latency imposed by the handshake impacts user experience and discourages websites from enabling TLS. A less-known but significant contributor to the cost of TLS is the validation process of the server's certificate. The web browser validates the server's certificate using certificate revocation protocols such as the Online Certificate Status Protocol (OCSP), which adds additional latency and leads clients to cache certificate validation results (trading off the freshness of certificate validation).

A number of existing proposals have mitigated some of the cost of TLS by decreasing the number of round trips for a full TLS handshake. A proposal called Fast-track removes one round trip from the handshake when the client has cached long-lived parameters from a previous handshake [86]. Another proposal, TLS False Start, reduces the handshake to one round trip when whitelisted secure cipher suites are used [87], which works only when the client sends data first, as in the case of HTTP. A third proposal, TLS Snap Start,

reduces the handshake to zero round trips when the client has performed a full handshake with the server in the past and has cached static parameters [88]. Even with Snap Start, the client cannot cache the certificate’s validation status beyond its validity period, and so Snap Start cannot always eliminate the certificate validation step.

In this chapter, we conducted a study of OCSP responders in the wild, including measurements of the validity durations and response times. We observe a noticeable penalty for TLS connection time due to OCSP validation. We present server certificate prefetching and prevalidation, a method by which web browsers can perform zero round trip Snap Start handshakes with a server even if the browser has never seen the server before. In addition to enabling Snap Start handshakes, certificate prefetching allows the client to prevalidate the certificate, so that certificate validation does not lead to perceived latency for the user. By allowing browsers to use Snap Start more often and by removing certificate validation from the time-critical step of a page load, prefetching can encourage servers to enable TLS more widely and allow browsers to verify certificate status more often and strictly.

4.1 Performance Cost of OCSP

To better understand the performance cost of certificate validation, we conducted measurements of OCSP lookup response times in the wild.

4.1.1 Experimental Setup

To collect statistics of OCSP responses in the wild, we ran experiments on the Perspectives system [64]. Perspectives has a collection of network notary servers that periodically probe HTTPS servers and collect public key certificates, which allows clients (using our browser extensions) to compare public keys from multiple network vantage points. In this work,

we extended the Perspectives system to probe OCSP responders for certificate revocation statuses if the queried certificate was configured with an OCSP responder URL. The data collected on the notary servers include the revocation status of the certificate, the validity lifetime of the OCSP response, and the latency of the OCSP lookup. In addition to probing OCSP responders from the notary servers, we performed latency measurements for OCSP lookups on clients that have installed our Perspectives extension for Google Chrome. For each certificate that was fetched from an HTTPS website, we performed an OCSP request and measured the elapsed time to complete the lookup. As of May 2011, there were 242 active clients contributing data for this measurement. The notary servers receive data from clients with our Google Chrome extension as well as the previously deployed Firefox extension.

4.1.2 Results

OCSP Response Validity Lifetime

Table 4.1 gives the OCSP response validity lifetime for certificates from OCSP responders for which the notary servers have performed more than 1000 OCSP lookups. We observe that 87.14% of the OCSP responses are valid for a period of equal to or less than 7 days. The minimum observed lifetime was 13 minutes. Analyzing the lifetime of OCSP responses helps us determine how often a prefetched OCSP response would expire before the certificate is actually used. Shorter OCSP response validity lifetimes reduce the effectiveness of OCSP response caching.

OCSP Lookup Response Time

Figure 4.1 shows the distribution of the OCSP lookup response times that we recorded. The data shows that although 8.27% of the probes took less than 100 ms to complete, a majority of the OCSP probes (74.8%) took between 100 ms and 600 ms. In our measure-

OCSP responder	Number of OCSP lookups	Number of distinct certificates	Validity lifetime		
			Avg	Min	Max
http://EVSSL-ocsp.geotrust.com	2035	198	6 days 23 hours	12 hours	7 days 11 hours
http://ocsp.cs.auscert.org.au	1060	97	4 days	4 days	4 days
http://ocsp.cacert.org/	2381	76	3 hours	15 minutes	23 hours
http://ocsp.usertrust.com	3846	315	4 days	4 days	4 days
http://ocsp.godaddy.com	90925	4139	7 hours	6 hours	11 hours
http://ocsp.comodoca.com	56928	4581	4 days	4 days	4 days
http://ocsp-ext.pki.wellsfargo.com/	2612	53	20 hours	13 minutes	1 day
http://ocsp.entrust.net	18691	1474	7 days 14 hours	7 days	8 days 4 hours
http://ocsp.netsolssl.com	4117	570	4 days	4 days	4 days
http://EVIntl-ocsp.verisign.com	64403	1566	7 days	7 days	86 days 7 hours
http://ocsp.digicert.com	92093	1672	7 days	7 days	7 days
http://ocsp.starfieldtech.com/	9016	480	11 hours	6 hours	1 day 5 hours
http://ocsp.webspace-forum.de	2228	29	4 days	4 days	4 days
http://ocsp.startssl.com/sub/class1/server/ca	4963	348	5 hours	1 hour	1 day 4 hours
http://ocsp.startssl.com/sub/class2/server/ca	4597	160	6 hours	1 hour	1 day 4 hours
http://ocsp.serverpass.telesec.de/ocspr	2212	248	1 hour	1 hour	1 hour
http://ocsp.gandi.net	1060	78	4 days	4 days	4 days
http://EVSecure-ocsp.verisign.com	108993	465	7 days	7 days	7 days
http://ocsp.globalsign.com/ExtendedSSL	2441	115	7 days	7 days	7 days
http://ocsp.verisign.com	247251	12433	7 days	7 days	20 days 21 hours
http://ocsp.thawte.com	134321	3811	7 days	7 days	7 days
http://ocsp.tcs.terena.org	7823	675	4 days	4 days	4 days

Table 4.1: Validity lifetime of OCSP responses.

ments, the median OCSP lookup time is 291 ms and the mean is 497.55 ms. Table 4.2 gives the response time statistics breakdown of OCSP responders for which at least 500 OCSP probes were performed. Our data for OCSP responder response times only include measurements performed at the client side (using the Perspectives extension for Google Chrome) and not on the notary servers. We believe the measurements from real web clients more accurately reflect the latency experienced by a user. We observe that 95.3% of the OCSP responses are cached by the OCSP responders and are not generated at the time of request. These OCSP responders therefore do not support the optional OCSP nonce specified in RFC 2560. If OCSP responders are required to support nonces and generate responses at the time of request, we expect an increase in response time for the OCSP responder to generate a response.

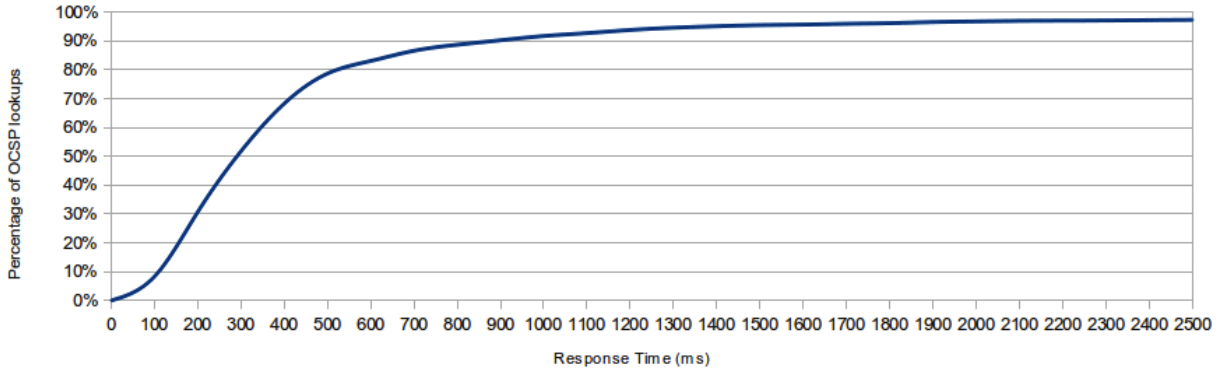


Figure 4.1: Cumulative distribution of OCSLP lookup response times.

OCSLP responder	Number of lookups	Response time			
		Median (ms)	Min (ms)	Max (ms)	Standard deviation
http://EVSecure-ocsp.verisign.com	938	167	25	7235	610.76
http://ocsp.digicert.com	1372	252	12	12303	759.64
http://ocsp.godaddy.com/	741	101	20	4832	515.53
http://ocsp.thawte.com	4209	564	10	12376	976.09
http://ocsp.verisign.com	1389	279	21	10209	743.53

Table 4.2: Response times of OCSLP responders.

4.1.3 Lessons

The actual response time of a user navigating to a previously unvisited HTTPS website typically consists of several round trip times: the DNS lookup, the TCP three-way handshake, the TLS handshake, the OCSLP lookup (usually blocking the completion of the TLS handshake), and finally the HTTP request-response protocol. Our measurements show that OCSLP validation is a significant source of user-perceived latency. Browsers have implemented a technique called *DNS prefetching* to reduce the DNS lookup time, which we further extend to prefetch and prevalidate TLS server certificates.

DNS Prefetching

When establishing connections with web servers, the web browser relies on the Domain Name System (DNS) [89] to translate meaningful host names into numeric IP addresses. The IP addresses of recently resolved domain names are typically cached by the local

DNS resolver, e.g., the web browser or operating system. If the resolution of a domain name is not locally cached, the DNS resolver sends requests over the network to DNS servers which answer the query by itself, or by querying other name servers recursively. Previous studies reveal that DNS resolution times cause significant user perceived latency in web surfing, more so than transmission time [90]. To increase responsiveness, modern browsers such as Google Chrome implement DNS prefetching (or pre-resolving), which resolves domain names before the user clicks on a link [91]. Once the domain names have been resolved, when the user navigates to that domain, there will be no effective user delay due to DNS resolutions.

Web browsers deploy various heuristics to determine when DNS prefetching should be performed. A basic approach is to scan the content of each rendered page, and resolve the domain name for each link. In Google Chrome, the browser pre-resolves domain names of auto-completed URLs while the user is typing in the omnibox. In addition, DNS prefetching may be triggered when the user's mouse hovers over a link, and during browser startup for the top 10 domains. Google's measurements show that the average DNS resolution time when a user first visits a domain is around 250 ms, which can be saved by DNS prefetching [31].

Browser vendors also allow web page authors to control which links on their pages trigger DNS preresolutions. When a web page includes a tag of the form

```
<link rel="dns-prefetching" href="//domain">
```

then `domain` will be preresolved. Further, a web page can use a

```
<meta http-equiv="x-dns-prefetch-control">
```

tag to specify that certain links should or should not be preresolved.

4.2 Server Certificate Prefetching and Prevalidation

To reduce OCSP latency, a server may allow clients to prefetch its handshake information by publishing its certificate, cipher suite choice, and orbit. (For simplicity, we refer to the prefetching of this information and the prevalidation of the certificate as “certificate prefetching.”) The client obtains this information when it is likely that the user might navigate to the website. The browser can use the same triggers that it uses to pre-resolve hostnames to determine when certificate prefetching is useful: for example, when the user is typing in the omnibox or when a user is viewing a page with links to HTTPS websites.

In this section, we discuss two major benefits of certificate prefetching, and describe various methods for clients to download server information.

4.2.1 Benefits of prefetching

Enable abbreviated handshakes.

After prefetching a server’s certificate, a web browser can use Snap Start without having performed a full handshake with the server in the past. Studies of user browsing behavior suggest that at least 20% of websites that a user visits in a browsing session are sites that the user has never visited before [92, 93, 94, 95]. These studies may underestimate how often certificate prefetching will be useful, since Snap Start without prefetching cannot be used when the browser cache has been cleared since the browser’s last full handshake with a server.

Enable prevalidation of server certificates.

Prefetching the server certificate allows the browser to validate the certificate in the background before the user navigates to the website. Our measurements in Section 4.1 show that certificate validation performed during the TLS handshake introduces significant

latency. Provided that the certificate status is not in the client's cache, a Snap Start handshake with a prefetched and prevalidated certificate is significantly faster than a Snap Start handshake without prefetching.

4.2.2 Prefetching methods

A naïve prefetching method is to open a TLS connection to the server and cache the necessary information needed to perform a Snap Start handshake. These dummy connections basically perform a standard TLS handshake with the server, and would eventually disconnect on timeout. However, many clients performing TLS dummy handshakes may negatively impact server performance and also flood the server's session cache. We discuss four options for certificate prefetching that add little or no server load.

1. **Prefetching with a truncated handshake.** To perform a Snap Start handshake, a web browser requires the server's certificate, cipher suite choice, and orbit. In a standard TLS handshake, the browser has obtained all this information by the time it receives the `ServerHelloDone` message, so the browser can prefetch the certificate and then truncate the handshake before either party performs any of the TLS handshake's expensive steps.

The browser can truncate the handshake by using the alert protocol that TLS specifies. An alert may be sent at any point during a TLS connection, and alerts specify a description (for example, `unexpected_message` or `bad_record_mac`) and an alert level of warning or fatal. If either party sends a fatal alert at any point during the connection, then the server must invalidate the session identifier.

Thus the browser can prefetch a server's certificate information by sending a `ClientHello` message with an empty Snap Start extension and sending a fatal alert after receiving the `ServerHelloDone` message. The alert ensures that the server closes the

session, so that prefetching does not flood the server's session cache or keep the socket open longer than necessary. After caching the appropriate information and validating the certificate, the browser can perform a Snap Start handshake if the user actually navigates to the website.

Like full dummy handshakes, truncated handshakes allow a browser to prefetch certificate information even if the server has not taken any actions to enable prefetching. A truncated handshake requires both the client and the server to do much less work than a full dummy handshake, and as a result the impact on the server is less dramatic. Truncated handshakes will also dirty server logs; without adding a new TLS alert number, a browser performing a truncated handshake for prefetching will have to use an inaccurate alert such as `user_canceled` or `internal_error` to close the connection.

2. **Prefetching via HTTP GET.** For a web browser to prefetch a certificate via a HTTP GET request to the server, the server must place the concatenation of its certificate, supported cipher suites, and orbit in a file at a standardized location. (In our implementation, we prefetched from `http://www.domain.com/cert.txt`.) The web browser retrieves the file, parses and validates the certificate, and caches all the information for use in a Snap Start handshake later. Transmitting certificates in plaintext over HTTP does not compromise security, as certificates are sent in plaintext during the normal TLS handshake.
3. **Prefetching from a CDN.** To avoid placing any extra load on the server, a client can attempt to prefetch certificate information from a CDN, for example by sending a request to `http://www.cdn.com/domain.com.crt`. The browser cannot know in advance which CDN a particular website uses to host its certificate information,

so it can send requests to multiple CDNs to have a high probability of successfully prefetching a server's certificate. Previous research suggests that sending requests to a small number of CDNs will cover a large percentage of the CDN market share [96]. Alternately, a DNS TXT record can hold the location where a browser should prefetch a server's certificate, so that the browser does not need to query multiple CDNs. Once the web browser has successfully obtained certificate information from a CDN, it proceeds to parse the certificate and cache the information.

4. **Prefetching from DNS.** The server may place its certificate information in a DNS record to offload the prefetching traffic. There has been previous work to store certificates or CRLs in DNS using CERT resource records [97], although not widely supported in practice. For the convenience of our prototype implementation, we stored the server's certificate information in a standard DNS TXT resource record, which allow servers to associate arbitrary text with the host name. Web browsers can prefetch certificates by querying for the domain's TXT record, in parallel with the domain's A record, during the DNS prefetching phase. Although TXT records were originally provisioned to hold descriptive text, in practice they have been freely used for various other purposes. For example, the Sender Policy Framework (SPF) [98] uses TXT records to specify which IP addresses are authorized to send mail from that domain. We also consider recent proposals in the IETF DNS-based Authentication of Named Entities (DANE) working group that suggest using DNSSEC to associate public keys with domain names. They introduce a new TLSA resource record type that allows storing a cryptographic hash of a certificate or the certificate itself in DNS [84].

As with HTTP GET prefetching, transmitting certificates from DNS or a CDN

does not decrease security. If the CDN or DNS servers are compromised and serve a forged certificate, the user will be prompted with a certificate warning, just as if an attacker had replaced a legitimate certificate in a normal TLS handshake.

4.2.3 Implementation

We developed prototype implementations of DNS and HTTP GET prefetching in Chromium, revision 61348, as well as an OpenSSL prototype of Snap Start for running our experiments. We modified Chromium’s DNS prefetching architecture; when the browser pre-resolves a domain name for a HTTPS URL, we added code to send an asynchronous request to fetch a DNS TXT record or a text file at a known location on the web server. If the request is successful, the certificate is parsed out of the data and the browser sends another asynchronous OCSP validation request. The certificate and validation status are stored in a cache, which is checked before each TLS handshake to determine if a Snap Start handshake is possible.

In our prototype implementation, certificate prefetches are triggered by the same heuristics that trigger DNS preresolutions. If browsers adopt certificate prefetching, we propose that they deploy certificate prefetching controls analogous to the DNS prefetching controls discussed in Section 4.1.3. These controls can allow web page authors to opt-in and opt-out of prefetches for specific domains, thereby helping the browser ensure that certificate prefetching requests are useful and not wasteful.

4.3 Performance Evaluation

Our experiments sought to answer the following questions about certificate prefetching:

- **By how much does prefetching reduce user-perceived latency?** To answer this question, we compared the latency of a Snap Start handshake with a prevali-

dated certificate to a Snap Start handshake using online certificate validation.

- **How does prefetching impact server performance?** For each certificate prefetching method, we measured user-perceived latency and server throughput as the server was flooded with certificate prefetching requests. This data let us compare the effect of traffic from different prefetching strategies on server performance. We used a cloud-based service to generate load on our test server.

4.3.1 Experimental setup

We used the hosting company Slicehost to acquire machines for running our experiments. Our server machine ran Apache 2.2.17 and OpenSSL 0.9.8p with our Snap Start prototype (on Ubuntu10.04 with 256MB of RAM and uncapped outgoing bandwidth). On separate client machines, we used Chromium, revision 61348 with our modifications to support certificate prefetching and Snap Start with a prevalidated certificate. We generated TLS 1.0 handshakes with RSA key exchange, AES-256-CBC encryption, and SHA-1 message authentication.

Comparing handshake latencies.

Our first experiments measure the latencies of three types of handshakes: 1.) a Snap Start handshake with a prefetched and prevalidated certificate, 2.) a Snap Start handshake with a cached but not validated certificate, and 3.) a normal full TLS handshake. We measured handshake latency by modifying Chromium on a client machine (which had 1GB of RAM and ran Ubuntu 10.04) to generate 500 requests one after the other and record the latency for each request.

Measuring the effects of certificate prefetching on server performance.

Our next experiments compare how different prefetching methods impact server performance. We first measured the server's latency and throughput when the server is not handling any other requests. We performed these measurements for HTTP HEAD requests, as well as for each of the three types of handshakes above (Snap Start with prevalidated certificate, Snap Start with online certificate validation, and normal full TLS handshake). We used the command-line tool `httping` [99] to generate HTTP HEAD requests, a Chromium client to generate TLS Snap Start handshakes, and OpenSSL to generate normal TLS handshakes. To measure throughput, we set up ten separate client machines (each with 256MB of RAM and capped at 10Mbps outgoing bandwidth) making continuous requests, and we logged each request on the server.

Some of our prefetching methods generate additional requests to the server stemming from client certificate prefetch requests. We therefore measured the server's latency and throughput as the server was flooded with prefetching requests from clients. For each prefetching method that affects the web server (i.e. HTTP, truncated handshakes, and full dummy handshakes), we set up client machines to simulate prefetching traffic using that method, with each prefetching client hitting the server with approximately twenty requests per second. While these clients were prefetching certificates from the web server, we again measured latency and throughput of HTTP HEAD requests and the three handshakes. For example, to measure the impact of truncated handshake prefetching on a web server handling HTTP HEAD requests, we set up ten clients to flood the server with truncated TLS handshakes, and then measured the latency and throughput of HTTP HEAD requests. We repeated the experiment with the number of prefetching clients varying from one to ten.

Since prefetching from DNS or a CDN does not affect the web server, the control

measurements (i.e. latency and throughput for requests while there is no prefetching traffic) cover those prefetching methods. The three types of prefetching traffic for which we measured server performance were HTTP GET requests, truncated handshakes, and the naïve method of full dummy TLS handshakes.

We also measured the data transfer overhead that a server can expect to incur by enabling certificate prefetching and Snap Start. The overhead is a function of the fetchthrough rate, the proportion of prefetches that lead to an actual page load. We measured data transfer for a HTTP GET prefetch, a truncated handshake prefetch, a page load using Snap Start, and a page load using a normal TLS handshake. We assume that with no prefetching, every page load requires a normal TLS handshake, and with prefetching, every page load uses a Snap Start handshake. Overhead is then calculated as $\frac{np+as}{at}$, where n is the number of prefetches, p is the bytes transferred for a prefetch, a is the number of actual page loads, s is the bytes transferred for a page load using Snap Start, and t is the bytes transferred for a page load using normal TLS.

4.3.2 Results

Table 4.3 shows the median and mean latency for each type of request. Snap Start with a prevalidated certificate corresponds to the situation when the client has prefetched and prevalidated the certificate and then performs a Snap Start handshake without needing to validate the certificate. The row labeled Snap Start corresponds to the situation when the client has cached the information necessary to perform a Snap Start handshake but must validate the certificate. **The data shows that the median latency for a Snap Start handshake with a prevalidated certificate is four times faster than a normal TLS handshake. Moreover, prevalidation speeds up basic Snap Start by close to a factor of three.**

	Median latency (ms)	Mean latency (ms)
Snap Start, prevalidated certificate	30.45	35.58
Snap Start, no prevalidation	83.40	99.86
Normal TLS	121.82	124.11

Table 4.3: Latency measurements for a Snap Start handshake with prevalidated server certificate (no verification during the handshake), a Snap Start handshake with online certificate verification, and a normal (unabbreviated) TLS handshake.

Figure 4.2 shows how different prefetching methods affect the server’s latency and throughput for HTTP HEAD requests, as we scale up the number of prefetching clients. For example, with ten prefetching clients, median latency for HTTP HEAD requests increased by 8.5% with HTTP GET prefetching, by 3.0% with truncated handshake prefetching, and by 26.7% with full dummy handshake prefetching.

Figure 4.3 shows the data transfer overhead incurred by HTTP GET and truncated handshake prefetching. (Prefetching from DNS or a CDN incurs no server overhead). Truncated handshake prefetching is about 10% less data transfer per prefetch than HTTP GET prefetching. The overhead varies widely depending on the fetchthrough rate, which is determined by the browser’s prefetching strategy and how accurately the browser can predict the user’s actions.

4.3.3 Analysis

Our experiments show that prefetching certificates allows for much faster handshakes than Snap Start without prefetching. We measured median latency for a Snap Start handshake with a prevalidated certificate to be 64% faster than a Snap Start handshake with an unvalidated certificate. However, this figure is probably a conservative estimate of the benefits of prevalidating, due to the unusually high speed of Slicehost’s network connection. Our measurements of OCSP response times in the wild, shown in Figure 4.1, show that prevalidating certificates will reduce latency even more in a real-world setting.

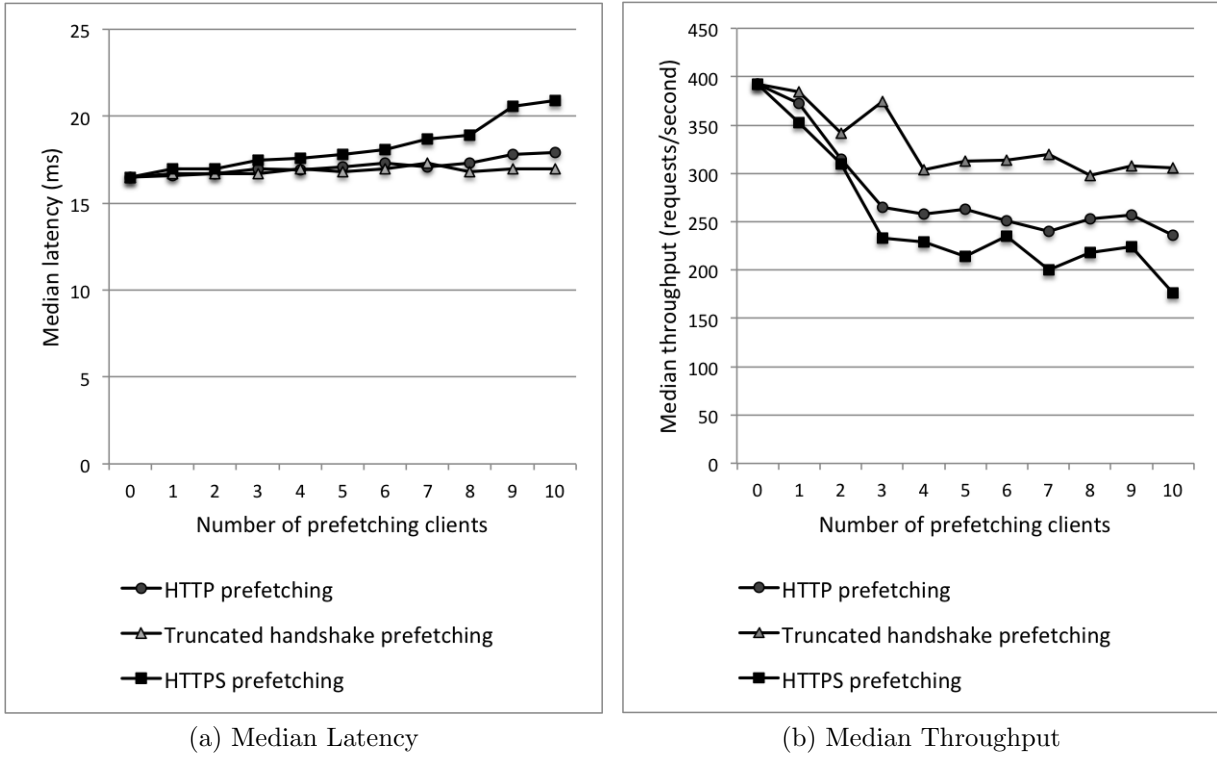


Figure 4.2: Median latency and throughput for HTTP HEAD requests with different types of prefetching traffic.

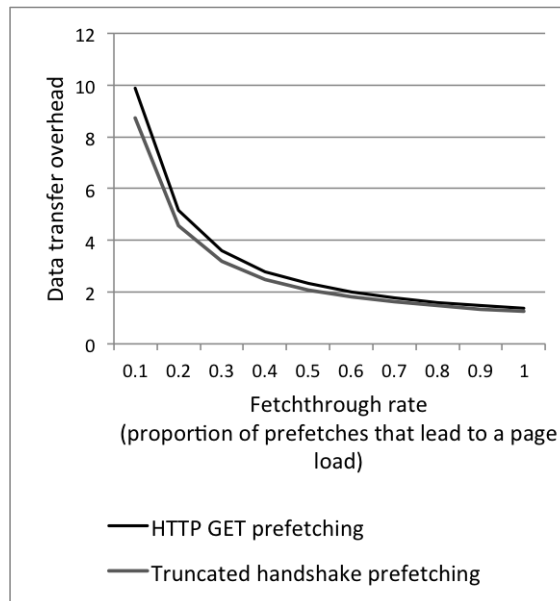


Figure 4.3: Data transfer overhead for certificate prefetching.

In addition to enabling Snap Start handshakes when the browser has never seen a website before, certificate prevalidation is useful when the browser has certificate information from a previous handshake but does not have its OCSP status cached.

Our experiments also show that prefetching via any of our proposed prefetching methods has a less dramatic impact on server performance than doing full dummy handshakes. Truncated handshake prefetching appears to have the smallest effect on server performance. However, in deciding between truncated handshake prefetching and HTTP GET prefetching, clients and servers may want to consider factors such as client-side code complexity, which we discuss below. Since prefetching via full dummy handshakes places a heavier load on the server and also requires more computation for the client, we conclude that full dummy handshakes are a poor choice for prefetching.

While Snap Start and prevalidating certificates reduce latency, throughput with no cover traffic is about the same for all three types of handshakes. This is because the server does about the same amount of computational work in each handshake, with the main difference being how long the socket stays open. Certificate prefetching and Snap Start are thus mechanisms for reducing client-side latency, not for improving server throughput.

As shown in Figure 4.3, for HTTP GET and truncated handshake prefetching, data transfer overhead can be high when the fetchthrough rate is low. If browsers prefetch aggressively, then DNS or CDN prefetching will avoid incurring this overhead for servers with data transfer limits. If browsers prefetch conservatively, then data transfer overhead is modest at less than 2x for fetchthrough rates higher than 0.5.

Chapter 5

Short-Lived Certificates

The work in this chapter was done in collaboration with Emin Topalovic, Brennan Saeta, Collin Jackson, and Dan Boneh.

As mentioned in Section 1.4.1, the purpose of OCSP is for revoking a certificate prior to its expiration date. This is because private keys corresponding to the certificate can be stolen, the certificate could have been issued fraudulently (e.g., by a compromised CA), or the certificate could have simply been reissued (e.g., due to a change of name or association). Unfortunately, OCSP has been ineffective in the event of high-profile security breaches of certificate authorities (i.e. Comodo [29] and DigiNotar [30]). With fraudulently issued certificates exposed in the wild, browser vendors were forced to issue software updates to blacklist bad certificates instead of relying on revocation checks.

In response, Google has announced plans to disable altogether OCSP in Chrome and instead reuse its existing software update mechanism to maintain a list of revoked certificates on its clients. For space considerations, their global CRL is not exhaustive, and can exclude the revocations that happen for purely administrative reasons. Network filtering attacks that block updates are still possible, but would require constant blocking from the time of revocation. Google's decision is mainly due to the ineffectiveness of soft-fail

revocation checks (treating OCSP query timeouts as valid), and also the massive costs in performance (as we observed in Section 4.1) and user privacy [100]. Similarly, Mozilla recently announced its plan [101] to implement a browser-maintained global CRL as part of their certificate revocation checking process.

However, revocation by software update takes control of revocation out of the hands of the CAs and puts it in the hands of software and hardware vendors who may have less of an incentive to issue a software update every time a certificate is revoked. For instance, multiple smartphone vendors have failed to issue software updates to block forged certificates even several months after the CA security breaches. We argue that certificate authorities should reassert control over the certificate revocation process by issuing certificates with a very short lifetime, for security *and* performance reasons. These certificates complement browser-based revocation by allowing certificate authorities to revoke certificates without the cooperation of browser vendors, while eliminating the performance penalty of OCSP.

5.1 Deficiencies of Existing Revocation Mechanisms

We begin by surveying the existing standards for removing trust from a valid certificate before its expiration date, and discuss the deficiencies that have caused Google Chrome to abandon them.

5.1.1 CRL

One solution to dealing with certificates that go bad is the certificate revocation list (CRL) [20]. When a certificate goes bad, its identifying serial number is published to a list, signed and timestamped by a CA. In order to trust a certificate, a user must not only verify the signature and expiration date, but also ensure that the certificate is not listed

in CRLs.

For CRLs to be effective, one assumes that (1) up-to-date lists are published frequently by the CA, (2) the most recent list is available to the verifier, and (3) verification failures are treated as fatal errors. These constraints on CRLs degrade their effectiveness as a revocation mechanism:

- An earlier study [102] on real-world CRLs indicated that more than 30% of revocations occur within the first two days after certificates are issued. For CAs, there is a tradeoff between their CRL publishing frequency and operational costs. For CAs that update CRL with longer intervals, there is a risk of not blocking recently revoked certificates in time.
- Since CRLs themselves can grow to be megabytes in size, clients often employ caching strategies, otherwise large transfers will be incurred every time a CRL is downloaded. This introduces cache consistency issues where a client uses an out-of-date CRL to determine revocation status.
- Browsers have historically been forgiving to revocation failures (a.k.a. “fail open”) so as not to prevent access to popular web sites in case their CAs are unreachable [103]. In practice, they either ignore CRL by default, or do not show clear indications when revocation fails [104]. Unfortunately, this lets a network attacker defeat revocation by simply corrupting revocation requests between the user and the CA.
- It should also be noted that the location of the CRL (indicated by the CRL distribution point extension) is a non-critical component of a certificate description, according to RFC5280. This means that for certificates without this extension, it is up to the verifier to determine the CRL distribution point itself. If it cannot, CRLs may be ignored [105].

5.1.2 OCSP

The Online Certificate Status Protocol (OCSP), an alternative to CRLs proposed in RFC 2560 [22], allows client software to obtain current information about a certificate's validity on a certificate-by-certificate basis. When verifying a certificate, a client sends an OCSP request to an OCSP responder, which responds whether the certificate is valid or not. Typically, clients are instructed to cache the OCSP response for a few days [23]. OCSP responders themselves are updated by CAs as to the status of certificates they handle.

In theory, it is possible for CAs to issue OCSP responses with short validity periods (since the response size is smaller than a CRL), however there are many real-world constraints that make this approach infeasible:

- OCSP validation increases client side latency because verifying a certificate is a blocking operation, requiring a round trip to the OCSP responder to retrieve the revocation status (if no valid response found in cache). In Section 4.1, our results indicate that 91.7% of OCSP lookups are costly, taking more than 100ms to complete, thereby delaying HTTPS session setup.
- OCSP may provide real-time responses to revocation queries, however it is unclear whether the responses actually contain updated revocation information. Some OCSP responders may rely on cached CRLs on their backend. It was observed that DigiNotar's OCSP responder was returning good responses well after they were attacked [106].
- Similar to CRLs, there are also multiple ways that an OCSP validation can be defeated, including traffic filtering or forging a bogus response by a network attacker [107]. Most importantly, revocation checks in browsers fail open. When they cannot verify a certificate through OCSP, most browser do not alert the user or

change their UI, some do not even check the revocation status at all [104]. We note that failing open is necessary since there are legitimate situations in which the browser cannot reach the OCSP responder. For example, at an airport, a traveler might be asked to pay for Internet service before connecting to the Internet. In this case of these *captive portals*, the browser cannot validate the gateway’s certificate using OCSP and must implicitly trust the provided certificate so that the user can enter her payment information and connect to the Internet.

- OCSP also introduces a privacy risk: OCSP responders know which certificates are being verified by end users and thereby responders can, in principle, track which sites the user is visiting.

5.2 An Old Proposal Revisited

We propose to abandon the existing revocation mechanisms in favor of an old idea [108, 109] — *short-lived certificates* — that puts revocation control back in the hands of the CAs. A short-lived certificate is identical to a regular certificate, except that the validity period is a short span of time.¹ Such certificates expire shortly, and most importantly, “fail closed” after expiration on clients without the need for a revocation mechanism.

In our proposal, when a web site purchases a year-long certificate, the CA’s response is a URL that can be used to download on-demand short-lived certificates. The URL remains active for the year, but issues certificates that are valid for only a few days. Every day a server-side element fetches a new certificate that is active for the next few days. If this fetch fails, the web site is not harmed since the certificate obtained the previous day is active for a few more days giving the administrator and the CA ample

¹We suggest a certificate lifetime as short as four days, matching the average length of time that an OCSP response is cached [23]. We encourage CAs to configure an even shorter validity period to reduce the time gap that a stolen revoked certificate can possibly be used.

time to fix the problem. In this way the burden of validating certificates is taken off the critical path of HTTPS connection establishment, and instead is handled offline by the web site. We emphasize that short-lived certificates do not change the communication pattern between clients and web servers. Moreover, since clients typically fail closed when faced with an expired certificate, this approach is far more robust than the existing OCSP and CRL based approaches.

Although it is conceptually simple, many issues need to be addressed for this approach to work. First, we hope to use short-lived intermediate certificates, but this requires some additional steps at the CA. Second, we need to ensure that installing a new certificate on a web server does not force a web server restart. Third, for web sites with many servers, we need an architecture to ensure that only one request from the site is issued to the CA per day (as opposed to one request per server). We describe a certificate subscription system that automatically fetches and installs short-lived certificates periodically makes it practical for server administrators to deploy (and not need to manually install their certificates). Finally, a small optional change to the client can provide additional defense-in-depth against attacks on the CA.

5.2.1 Design

In what follows we assume a *server* provides a particular service such as a web-based email over HTTPS. A *client* is a user of the service, which will validate whether the server provided certificate is signed by a trusted source to determine the authenticity of the server. Both the client and server trust an intermediate party, known as the *certificate authority* (CA), who certifies the identity of the server. In practice, clients are pre-installed with the public keys of the trusted CAs, thus, clients can verify the trusted CA's signatures. We describe the modifications we make on the three components in this

scenario: the certificate authority, the server, and the client.

Certificate Authority The role of a certificate authority is to sign the certificates for subscribing servers. The certificate authority has two modes of operations: *on-demand* and *pre-signed*. What differentiates the two is how the certificates are generated.

- *On-demand mode.* When using the on-demand approach, the CA keeps its private key online, and signs new certificates when requested. In on-demand mode, the online CA keeps a template certificate — a certificate with static information, such as the common name and public-key already populated — which is loaded when the web server requests a new short-lived certificate. The validity period of the certificate is altered such that it begins at the time of the request and ends the configured amount of time in the future, typically within a few days. The certificate is signed using the CA's private key and sent back to the web site.

In on-demand mode the hardware boxes used at the CA to manage the CA's private key can be configured so that they will never issue a certificate with a validity period of more than a few days beyond the present date. Consequently, a single compromise of the CA will only expose certificates that will expire shortly.

- *Pre-signed mode.* With the pre-signed approach, the CA's private key is kept offline and certificates are signed in batches. When a server requests a certificate, the CA looks through its store of pre-signed certificates to find the appropriate one, in which the validity period begins before the request time and ends at least a day after the request time. The extra overlap allows the requester to not have to worry about automatically re-issuing the request were it to be issued closer to its expiration date. Similar to previous two-level digital signature schemes [110] (using an offline CA to

pre-issue certificates), this reduces the computation of the online CAs. It also allows that CA's private key to remain offline, as is often the case for root CAs.

In either case, should the server request its certificate be revoked, the certificates, either the template or pre-signed, are removed from the CA store. Subsequent requests will fail.

Server Plug-in The short-lived certificates themselves are usable by any server which supports standard X.509 certificates. What is required is a side-loaded program (or plug-in) that manages the server's certificates by requesting fresh certificates from the certificate authority as the expiration time of the current certificate approaches. Our server-side program is set-up to automatically execute after a certain interval of time. It is recommended that the interval is set to at least two days before the expiration date to ensure new certificates are installed in a timely fashion.

When the server certificate-downloading program wakes up, it checks the expiration date of the current installed certificates and if any are approaching expiration, the program issues an HTTP GET request to the CA for a new certificate. The server-side program checks that the only field that changed in the new certificate is the validity period (and in particular, the public-key did not change). If so, it stores the new certificate in the certificate store and alerts the server to load the new certificate. In Section 5.2.2 we explain how to load a new certificate in popular web server without restarting the server. If the retrieved certificate is corrupt, it is ignored and the site admin is alerted.

Client-Side Pinning In current popular browsers certificate validation fails for expired certificates. Therefore, no client-side modifications are needed to implement our short-lived proposal. Chrome's CRL approach complements this mechanism well in case there

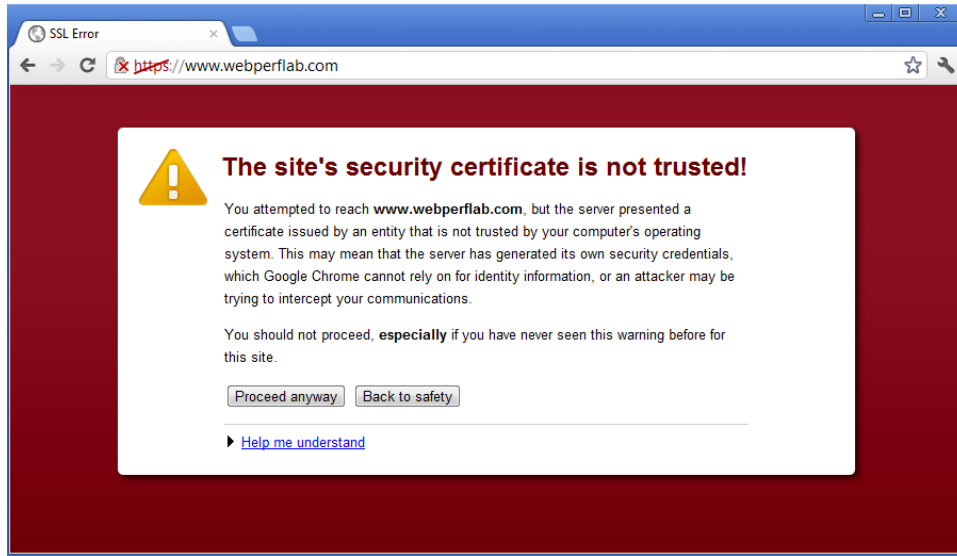


Figure 5.1: **Expired certificate warning in Google Chrome** *When encountering expired certificates, the browser blocks the page, removes the lock icon from the address bar, and presents a security warning dialog to the user.*

is a need to quickly revoke a compromised certificate (including root and intermediate certificates).

In practice, when encountering expired certificates, the browser blocks the page, removes the lock icon from the address bar, and presents a security warning dialog to the user (see Figure 5.1). We note that previous studies have shown that users may still click through the warnings [47] and therefore a strict hard-fail approach is suggested for better security, as implemented for HSTS [69] websites.

While no client-side modifications are required, the short-lived proposal can be strengthened with a small client-side extension. In particular, we propose to add a new X509 certificate extension to indicate that the certificate is a short-lived certificate. When a client sees such a short-lived certificate, it records this fact and blocks future connections to the server if the server presents a non-short-lived certificate. We call this *client-side pinning* for short-lived certificates, similar in a way to existing certificate authority pinning in

browsers [74, 111]. In addition to just pinning the CA used by a site, we are pinning the status of whether a site uses short-lived certificates. Client-side pinning ensures that the number of short-lived enabled certificates (including intermediate certificates) in a certificate chain should never decrease. This optional behavior should still allow short-lived certificates to be incrementally adopted on intermediate CAs. Client-side pinning prevents two attacks:

1. Suppose an attacker succeeds in compromising a CA *once* without being detected. Without short-lived pinning on the client, the attacker could simply request long-lived certificates from the CA's hardware and these certificates let the attacker man-in-the-middle all web sites that use this CA. With short-lived pinning, the attacker must request short-lived certificates from the CA's hardware, but by design the hardware will only issue short-lived certificates that expire in a few days. Therefore, a one-time compromise of the CA will not help the attacker. The attacker must repeatedly compromise the CA thereby increasing the chance of detection and revocation.
2. Consider a web site that currently uses long-lived certificates. If the server's secret key is stolen the site may ask the CA to revoke the long-lived certificate and then switch to using short-lived certificates. But an attacker can block revocation messages sent to clients and then use the stolen long-lived certificate to man-in-the-middle the web site. Clients would have no knowledge that revocation took place and will accept the revoked long-lived certificate. With short-lived pinning, if the client connects to the legitimate site after it switched to short-lived certificates, the long-lived certificate will no longer be accepted by the client.

If a website wishes to stop using short-lived certificates, the X.509 extension can

provide an option to disable client-side pinning.

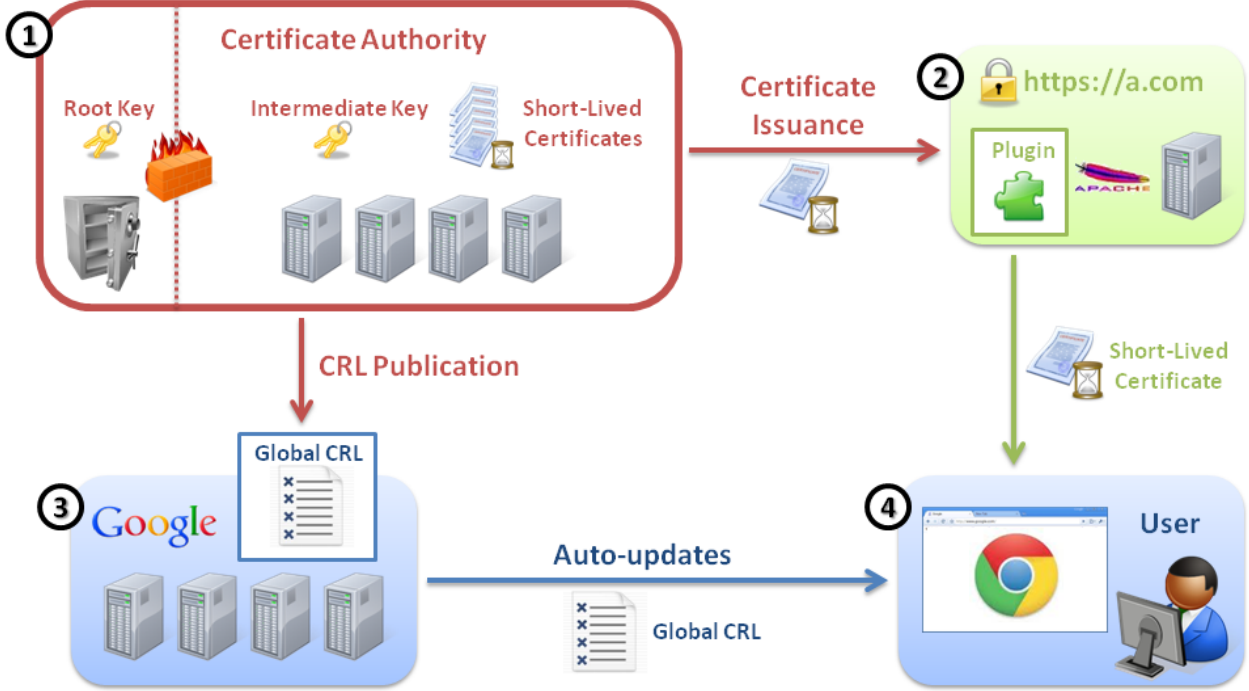


Figure 5.2: **Short-Lived Certificates and Chrome's CRL** 1. Certificate authorities pre-sign batches of short-lived certificates. 2. A plugin on the SSL website periodically fetches fresh short-lived certificates from the CA and loads it onto the server. 3. Google generates a list of revoked certificates, and automatically updates the global CRL to browsers. 4. When the user visits the SSL website, the certificate is validated against Chrome's CRL and also fails on expiration.

5.2.2 Implementation

We developed a prototype to enable and automatically update short-lived certificates for Apache web servers. We also implemented client-side pinning in Chromium web browser.

Certificate Authority Our certificate authority was implemented using Java and is served over Apache Tomcat as a web application. The web server issues an HTTP GET request to the CA server specifying the common name for the certificate it wishes to retrieve, as well as a unique certificate identifier. This identifier allows a web server to

have multiple certificates under the same common name stored by its one CA. These identifiers are chosen by the owners of the servers when they register with the CA for short-lived certificates. They allow a server to have multiple certificates under a common name, say if they wish to use a different private/public key pair or want a certificate that is a wild card certificate and one that is domain specific.

In either the pre-signed or on-demand mode, the CA's servlet looks for an appropriate certificate on the filesystem. In on-demand mode, the validity period of the matching template certificate is updated and signed with the CA's private key. The private key is stored encrypted on the CA's server, and is decrypted and brought into memory at start-up. The signing and general certificate handling is done using the IAIK cryptography libraries [112]. The pre-signed certificates are signed offline using a different key and are transferred to the servers manually. The batch can be set by the CA but will present a trade-off between security and ease-of-use. Pre-signing larger batches means less overhead of signing and transferring the certificates, but leaves more signed certificates on an online server and thus at the risk of being stolen. Each pre-signed and on-demand certificate is made valid for four days to match the length of time for which an OCSP response is cached [23].

Server Plug-in We implemented our server-side program in Java targeting Apache web servers. The program is set as a cron job executing every day. When the program runs, it checks to make sure the certificate is close to expiration. If true, it issues a GET request to our CA for either a pre-signed or on-demand certificate. Once the certificate is obtained it is stored to the filesystem in the standard PEM format.

Our Apache SSL configuration files are set such that the file locations of the certificates are symbolic links. When the new certificate is stored on the filesystem, all our program

has to do is re-point the symbolic link to the new certificate and optionally clean up old, expired ones. After this, the server certificate-downloading program issues a graceful restart command to Apache. This ensures the web server restarts and loads the new certificates without disrupting any existing connections.² Although we prototyped on the Apache web server, our proposal is applicable to other popular web servers such as Nginx and Jetty, without causing server downtime. Nginx supports graceful restart, similar to Apache, and Jetty supports reloading the Java trust store on-the-fly [113].

Client-Side Pinning We implemented a prototype of client-side pinning for short-lived certificate in Chromium (using revision 61348). Since Chromium utilizes the system cryptography libraries on each platform [114] to handle certificate verification, we implemented our code as a platform-independent function in the browser above the cryptography libraries, instead of modifying a specific library such as NSS. We reused the existing transport security state code in Chromium (for HSTS and public key pinning) to store our short-lived certificate pins persistently.

5.2.3 Deployment Incentives

The deployment incentives of web servers and CAs for short-lived certificates may be less obvious, which might discourage adoption. We discuss below the deployment costs of short-lived certificates from the perspectives of web servers and CAs, and explain why CAs and server operators are incentivized to deploy short-lived certificates.

- **Web server.** From the server’s point of view, manually updating short-lived certificates on a server is a tedious task, especially since they need to perform this rather frequently (once every day or every few hours). Fortunately, the method

²It is reasonable for a large site using Apache to use short-lived certificates. Apache can restart gracefully with no noticeable impact to end users as our benchmarking has shown.

we proposed (in Section 5.2.1) is an automatic certificate update system where the server can fetch new certificates from their CAs during their subscription period. We implemented scripts that automatically fetches new certificates from the CA, and performs the installation of downloaded certificates on-the-fly on an Apache web server. Our proposed system removes the burden of keeping server certificates up-to-date from human operators, to software (after an initial setup process by human), therefore it is no harder to deploy than traditional long-lived certificates.

- **CAs.** From a CA's point of view, supporting short-lived certificates introduces the extra costs of re-issuing certificates more frequently and making fresh certificates available to the subscribing web servers. This is necessary because web servers will automatically download fresh short-lived certificates from CAs every day (or every few hours). However, it is important to note that getting web servers to deploy short-lived certificates actually reduces the amount of OCSF request traffic being sent from web clients to the CA's OCSF responder. Since there are way more web clients than web servers on the Internet, it is safe to assume that the amount of OCSF requests coming from web clients (that could be saved) easily outweighs the amount of certificate downloads from the subscribing web servers. Therefore, deploying short-lived certificates actually reduces the resource requirements on the CAs' servers, and actually provides cost savings in bandwidth and equipments for CA operators.

Last but not least, deploying short-lived certificates offers better performance and security to web clients and servers: (1) in-band certificate revocation checks with no round-trip latencies result in faster web pages, and (2) in-band revocation checks are immune to network attackers that block or corrupt OCSF responses. As a result, CAs,

Table 5.1: Comparison of Certificate Revocation Approaches

	Chrome's CRL	Hard-Fail OCSP	Short-Lived Certificates	Chrome's CRL + Short-Lived Certificates
Untrust Rogue Root CAs	✓			✓
Revoke Misissued Certificates	✓	✓	✓	✓
Revoke Benign Certificates		✓	✓	✓
Support Captive Portals	✓		✓	✓
Avoid Page Load Delay	✓		✓	✓
Avoid User Tracking	✓		✓	✓
Avoid Single Point of Failure				✓
Support Legacy Clients			✓	

web servers, and browser vendors can proudly offer better performance and security to their users by supporting short-lived certificates.

5.3 Analysis of Post-OCSP World

In Section 5.1, we discussed the ineffectiveness of the existing soft-fail OCSP and CRL mechanisms, which have paved the way for recent proposals such as Chrome's browser-based CRLs. In this section, we discuss the benefits and shortcomings of various revocation approaches in a post-OCSP world (summarized in Table 5.1), including (1) Chrome's CRL, (2) hard-fail OCSP, (3) short-lived certificates, and (4) hybrid approach of short-lived certificates with Chrome's CRL.

5.3.1 Chrome's CRL

Google has announced plans to disable OCSP checks completely and reuse its existing software update mechanism to maintain a list of revoked certificates on its clients.

Advantages

In the case of certificate misissuances during CA incidents, Google could push out new CRLs that will block the fraudulently issued certificates on the clients in less than a daily time frame. Due to using software updates, this approach even has the ability to remove

the misbehaving root certificates. Browser-based CRLs are updated periodically and not fetched at the time connecting a site, thus there is no additional latency during page load, nor privacy concerns of tracking the user's browsing history. Further, network filtering attacks become more difficult, an attacker would have to consistently block software updates from the time of revocation, instead of only at the time of visit.

Disadvantages

Due to space considerations of maintaining a global CRL, Google will not support a vast amount of revocations that are due to administrative reasons. Should OCSP be disabled, certificate authorities lose control over revocation, therefore unable to revoke certificates for billing, re-issuance, or other benign reasons. Google has become a single point of failure for certificate revocation. Another major disadvantage of this approach is that legacy clients are currently not supported, such as mobile devices and other browsers. Google may want to provide their global CRLs as a public service for other browsers and applications, in a way similar to their Safe Browsing API [115].

5.3.2 Hard-Fail OCSP

In attempt to fix the ineffectiveness of OCSP under existing CA infrastructures, some security researchers as well as CAs have suggested clients enforce hard-fail OCSP. Some browsers do allow users to opt-in to hard-fail for revocation checks, but this must be turned on by default to be effective.

Advantages

Unarguably, the security of a hard-fail OCSP is better than existing soft-fail mechanisms. Unlike new browser-based CRL proposals, this approach supports the existing revocation infrastructure managed by CAs.

Disadvantages

Unfortunately, there are legitimate reasons why browsers refuse to simply turn on hard-fail OCSP. For example, users may need to log in to captive portals of WiFi hotspots where the login page is protected by SSL. Often, the HTTP traffic is blocked, including OCSP, and will cause certificate validation to timeout. If hard-fail OCSP was implemented, users will not be able to connect to many WiFi hotspots, even though they are probably not under attack.

Also, this approach shares many disadvantages of existing OCSP approach, including the ability for third party to track the user's browsing history. Further, clients may suffer significant connection latency due to OCSP queries, which even worse may discourage sites on adopting SSL. Whenever an OCSP responder goes down, all sites that use their certificates will go down as well.

In the case where a Root CA has been completely compromised, OCSP does not provide any protection since the attacker could have easily generated a valid OCSP response with a far expiration date. Clients would need software updates to untrust the bad root certificate.

5.3.3 Short-Lived Certificates

We described the approach of using short-lived certificates in Chapter 5. In short, to revoke a certificate, the CA simply stops reissuing new certificates, as any old certificates either must have expired, or would expire shortly.

Advantages

By default, certificate expiration is always strictly validated on clients. All major browsers check for the validity period of certificates and present alerts to users when encounter-

ing expired certificates. No modifications on clients are required, thus will work on all platforms and devices without updates.

In the event of a key compromise or fraudulent issuance of site certificates, with short-lived certificates, the CA simply stops renewing the certificate for revocation. The window of vulnerability will be bounded by the short validity period, which should be no more than a few days. We note that short-lived certificates can potentially help if supposedly a large CA (e.g., VeriSign) is hacked — the *too big to fail* problem. In that case, there will be a need to revoke the stolen certificates. Short-lived certificates can make that easier.

Unlike OCSP, user's browsing habits are not reported to any third parties when short-lived certificates are used. Further, this approach does not require additional round-trips to a SSL connection setup, as browsers do not need to verify the certificate status with an OCSP responder.

Since short-lived certificates are regular certificates, they can be chained in exactly the same manner as the existing deployment of X.509 certificates. This allows websites, and even intermediate certificate authorities, to incrementally deploy and take advantage of short-lived certificates, without a large migration, or infrastructure change.

Disadvantages

Although short-lived certificates cannot solve the problem of a root CA compromise (neither does hard-fail OCSP), it does improve the security of intermediate certificates that are short-lived. In on-demand mode, the fact that the certificate authorities are online increases the risk of the CA's key being stolen and fraud certificates being generated. This is less of an issue with the pre-signed mode where certificates are signed by an offline CA in batches, allowing the key to be kept safe and isolated from the online servers. However, signing in batches implies that a CA break-in will provide the attacker with a larger pool

of pre-signed certificates and thus a longer time during which they can masquerade as the certificate owners. Fortunately, this only hinders security if the attackers have also compromised a client's private key.

One possible issue that could be raised is the fact that if a CA falls under DDoS attacks, servers can not get updated certificates and are thus forced into service outage as soon as their certificates expire. This requires that the attacker is able to take down the CA consistently for at least a few days. Fortunately, there are many mitigations. One approach for CAs is to filter traffic based on IP, only allowing well-known customer IPs through. If using pre-signed mode, the distribution of certificates can further be handled by third-party distributors that specialize in handling DDoS-level traffic.

We note that deploying short-lived certificates might cause errors on clients whose clocks are badly out of sync (e.g., off by a week). It is recommended that clients should sync their clocks periodically, which is critical for preventing replay attacks.

5.3.4 Hybrid approach

Lastly, we consider a hybrid approach of using short-lived certificates in cooperation with Chrome's CRL. First of all, by issuing short-lived certificates, CAs immediately regain control of certificate revocation that was disabled by Chrome. Once CAs start to issue short-lived certificates for sites (as well as intermediate CAs), these sites will benefit from the improved security. In the event of keys being stolen or certificates being misissued, short-lived certificates ensures a shorter window of vulnerability by warnings on expiration.

In addition, Chrome's CRL complements nicely with short-lived certificates. With short-lived certificates alone, we mentioned that users may still be vulnerable in the worst case, when a root certificate is deemed untrustworthy. Now that browser-based CRLs

provide protection against CA incidents, the combination of these two approaches allows a full spectrum of revocation (on supporting browsers). Browser vendors will be able to revoke fraudulent certificates and rogue CAs, while CAs may control administrative revocations for benign certificates.

This hybrid approach does not have the client side performance or privacy issues caused by OCSP, and does not block users behind captive portals. In contrast, we note that using hard-fail OCSP along with Chrome’s CRL would still suffer the compatibility issues with captive portals, as well as page load delay and privacy issues. Given Chrome’s CRL in place, we suggest that adopting short-lived certificates gives the maximum security without the obvious shortcomings of OCSP.

5.4 Related Work

5.4.1 OCSP Stapling

An update to OCSP is OCSP stapling, where the web server itself requests OCSP validation which it passes on the response to inquiring clients. Stapling removes the latency involved with OCSP validation because the client does not need an additional round trip to communicate with the OCSP responder to check the certificate’s validity. This also removes the privacy concern of OCSP because the responder does not have access to knowledge about a web site’s visitors. Unfortunately this is not widely implemented, only 3% of servers support OCSP stapling [116]. Also, current implementations do not support stapling multiple OCSP responses for the chained intermediate certificates [103]. Further, we note that users are still prone to attacks if clients do not implement hard-fail OCSP stapling (and pin the status of whether a site uses OCSP stapling). Proposals [117, 118] to let servers opt-in to hard-fail on clients if OCSP response is not stapled are still work in progress. In contrast, short-lived certificates automatically fail closed on existing clients

without modifications.

Chapter 6

Forward Secrecy Performance

The work in this chapter was done in collaboration with Shrikant Adhikarla, Collin Jackson, and Dan Boneh.

In a typical TLS setup, the client sends a random nonce (the *pre-master secret*) to the server, that is used to derive the shared session key used for encryption. Since the pre-master secret is encrypted with the server's public key, only the holder of the server's private key should be able to derive the session key. Therefore, an eavesdropper cannot simply decrypt the captured TLS traffic.

However, the confidentiality of the server's private key is not always as robust as one may wish. An attacker could possibly steal private keys from server administrators via social engineering, recover expired private keys from discarded storage devices (that might be less protected), or perform cryptanalysis on the encrypted traffic with future super computers. In fact, the Heartbleed OpenSSL bug [119] makes a point that private keys can be silently stolen from unpatched servers. A leaked document [120] even suggests that some governments have surveillance programs to capture backbone communications. If a government stores all of the captured traffic and later requests the server's private key, then past encrypted communications may be decrypted.

An important property in this context is *forward secrecy*, which ensures that short-term session keys cannot be recovered from the long-term secret key. Especially in the situation where Internet surveillance is a concern, forward secrecy lets enterprises argue that eavesdroppers simply cannot reveal secret data of past communications. Currently, websites can enable forward secrecy using TLS’s ephemeral Diffie-Hellman (DHE) or ephemeral Elliptic Curve Diffie-Hellman (ECDHE) key exchange methods. With these methods, the server’s long-term secret key is used to sign a short-lived Diffie-Hellman key exchange message. The resulting Diffie-Hellman secret is used as the session’s pre-master secret. Once the pre-master secret is discarded after the session, the session key cannot be reconstructed even if the server’s private key is given.

Traditionally, the argument against deploying forward secrecy is that forward-secure cipher suites incur a significant performance cost. In our work, we evaluated the performance of TLS cipher suites and found that ECDHE-based forward secrecy is not much slower than RSA-based setups with no forward secrecy. The reason is that with the RSA key exchange, the server must perform an expensive RSA decryption on *every* key exchange. With the ECDHE key exchange, the server can RSA-sign its ECDH parameters once and re-use that signature across several connections. The server-side online cryptographic operation is then just one elliptic curve multiplication which can be faster than a single RSA decryption (of equivalent key strength). Furthermore, ECDHE outperforms DHE since the parameter sizes are significantly smaller while providing the same level of security. Finally, the ECDHE key exchange can even be faster than RSA key exchange if the ECDH parameters are signed with ECDSA.

We evaluated the performance costs of TLS forward secrecy on the server side and the client side. First, we conducted a *controlled experiment* where we load tested our TLS servers on an internal network. Second, we ran an *Ad experiment* to measure the

client-side TLS latencies on real-world clients.

6.1 Controlled Experiment - Server throughput

6.1.1 TLS server setup

We setup our TLS servers using Apache 2.4.4 compiled with OpenSSL 1.0.1e (with 64-bit elliptic curve optimizations). We applied Abalea’s `mod_ssl` patch [121] to enable 2048-bit Diffie-Hellman parameters.¹ We used Rackspace virtual private servers running Debian Linux 2.6.32-5 on AMD Opteron 4170 HE 2.1 GHz CPU, 512 MB RAM and 40 Mbps network bandwidth. We disabled TLS session resumption and HTTP persistent connections.

We evaluated five representative TLS cipher suites, including (1) RSA-RSA: RSA-2048 key exchange with RSA-2048 signatures, (2) DHE-RSA: DHE-2048 key exchange with RSA-2048 signatures, (3) ECDHE-RSA: ECDHE-256 key exchange with RSA-2048 signatures, (4) ECDHE-ECDSA: ECDHE-256 key exchange with ECDSA-256 signatures, and (5) DHE-DSA: DHE-2048 key exchange with DSA-2048 signatures. All of the cipher suites in our experiments were uniformly configured to use 128-bit AES-CBC encryption with SHA-1 HMAC. The security strengths of these cipher suites were not necessarily equivalent since we used commercial certificate authorities which do not issue certificates with arbitrary key strengths. Note that self-signed certificates would trigger SSL certificate warnings on real-world clients, thus does not suffice for our ad experiments. Table 6.1 describes the 3 production TLS certificate chains used in our evaluation, listing the signature algorithms, signature hash algorithms and chain sizes. We point out that there is a roughly one kilobyte size difference between the RSA and ECDSA certificate chains. This is because there are two certificates (leaf and intermediate) transmitted per chain,

¹Abalea’s patch is obsolete as of Apache 2.4.7.

	Leaf certificate	Intermediate certificate	Root certificate (not transmitted)	Chain size (bytes)
1.	RSA-2048, SHA-256	RSA-2048, SHA-1	RSA-2048, SHA-1	3,119
2.	DSA-2048, SHA-256	DSA-2048, SHA-256	DSA-2048, SHA-256	3,343
3.	ECDSA-256, SHA-256	ECDSA-256, SHA-384	ECDSA-384, SHA-384	2,104

Table 6.1: TLS certificate chains for evaluation issued by Symantec CA

and each ECDSA-256 public key along with a signature is roughly 500 bytes smaller than a RSA-2048 public key along with a signature.

We setup three different web pages of varying complexity for our experiments.

- Simple page - a copy of one of our author's home page. The page was static and hosted on a single domain.
- Complex page - a copy of Amazon.com's landing page. We hosted the page and all sub-resources (e.g., images, stylesheets and scripts) on a single domain.
- Multi-domain page - a copy of Salon.com's landing page. We setup 10 additional sub-domains on our site to host the sub-resources.

6.1.2 Methodology

We measured the average server throughput of each TLS server setup by generating large amounts of synthetic TLS traffic towards the server, from two client machines over a 40 Mbps private network. We used the ApacheBench tool to send HTTPS requests continuously, and concurrently (1,000 requests at the same time), from each client machine. We monitored the server throughput (number of requests per second) and took the average value over 5 minutes. For sanity check, we tested each TLS server configuration using GET requests and HEAD requests, separately.

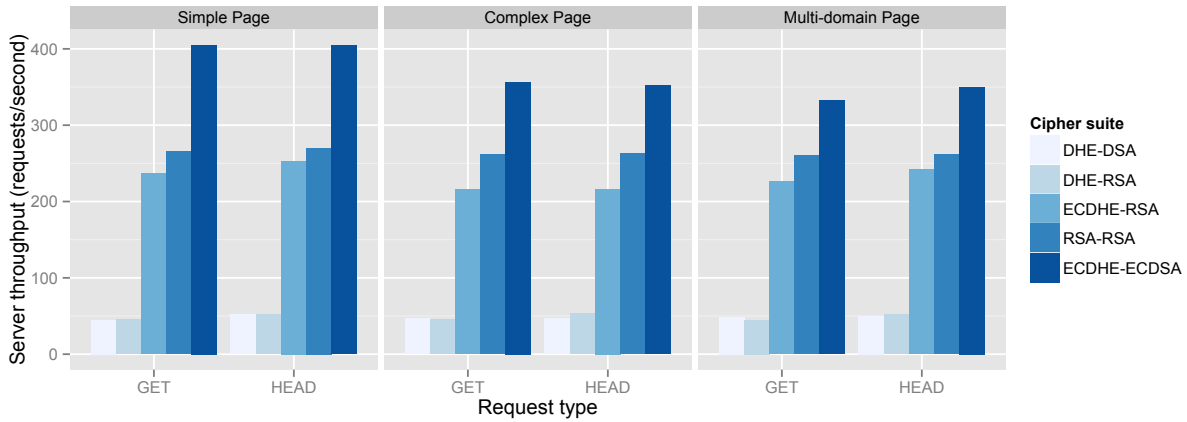


Figure 6.1: Server throughput of different configurations under synthetic traffic

6.1.3 Results

Figure 6.1 shows the average number of requests per second that the web server can serve when fully loaded under each server configuration. We compare the three cipher suites (RSA-RSA, DHE-RSA and ECDHE-RSA) that use the same signature algorithm (RSA-2048) with different key exchanges (RSA, DHE and ECDHE). First of all, RSA-RSA (with no forward secrecy) was clearly the fastest of the three regardless of the type of web page, peaking at 265.4 GET requests per second when serving simple pages.

Forward secrecy with DHE is costly. DHE-RSA performed the slowest of all, averaging only 45.7 requests per second in the best case. This should be due to the extra computation required for generating the ephemeral DH key (and RSA-signing it) for each `ServerKeyExchange` message.

Forward secrecy with ECDHE is basically free. Interestingly, the performance cost of ECDHE-RSA (that averaged 237 requests per second when serving simple pages) is dramatically cheaper than DHE, and even almost free compared to RSA (with no forward secrecy). The reason is that ECDHE requires significantly smaller parameters than DHE

to achieve equivalent security strengths. Further, with the RSA key exchange, the server must perform an expensive RSA decryption on every key exchange. With the ECDHE key exchange, the server can RSA-sign its ECDH parameters once and re-use that signature across several connections.

Forward secrecy with ECDHE-ECDSA actually improves performance. The performance of ECDHE-ECDSA is the fastest (peaking at 405 requests per second when serving simple pages). ECDHE-ECDSA is not only faster than ECDHE-RSA, but even faster than RSA-RSA, which does not provide forward secrecy. Moreover, ECDSA-256 has a higher security strength than RSA-2048, thus one could expect a larger difference if comparing equivalent strengths.

6.2 Ad-based experiment - Client latencies

6.2.1 Methodology

We conducted an ad-based experiment to measure TLS latencies of different cipher suites on real-world clients. Our experiment setup consisted of two machines (with separate IP addresses and domain names), one which runs the TLS servers to be tested (as described in Section 6.1.1), and the other which hosts our advertisement banner page (mainly a blank image and JavaScript code). We purchased ad impressions (see Section 6.2.2) to recruit real-world clients to view our banner. When our ad banner is rendered on each client, our code will create TLS connections to our TLS servers by loading each HTTPS test link in an IFRAME. For each HTTPS test, our script collected the following two measurements using the HTML5 Navigation Timing API:

- *TLS setup time*: The amount of time used to establish a SSL/TLS connection, including the TLS handshake time and the certificate validation time on the client.

Unfortunately, this measurement is currently only supported in Chrome.

- *TCP + TLS setup time*: The amount of time used to establish the transport connection, which includes the TCP handshake and TLS handshake. This measurement is currently supported in three major browsers (including Chrome 6+, Firefox 7+ and IE 9+).

Upon completion, the timing measurements were sent back to our log server, where the client’s IP address (and other personally identifiable information) were discarded. Our ad experiment did not require any user involvement. If the user navigated away (e.g., closed the tab) during the experiment, or if the TLS connection failed, our servers received partial results.

We note that modern web clients cache the validity statuses of certificates for performance reasons, thus our measurement results may be biased by the testing order (subsequent tests that share the same RSA certificate may load faster). As a workaround, two “cold” tests were added to warm the client-side caches, labeled as RSA-RSA_COLD and ECDHE-ECDSA_COLD, such that all of the subsequent tests would be equally evaluated under a warmed cache.

6.2.2 Results

We purchased 273,533 advertisement impressions from 23 January 2014 to 29 January 2014. We spent \$167.75 in total, including \$122.23 on a run-of-network campaign (195,214 impressions), and \$45.52 targeted on mobile devices (78,319 impressions). Not all ad impressions converted to valid measurements. We discarded impressions with clients that do not support HTML5 Navigation Timing, and clients that are not viewing our ad for the first time. Also, users may leave the web page before completion of tests. We indicate the number of unique clients that successfully performed each test in Figures 6.2 and 6.3.

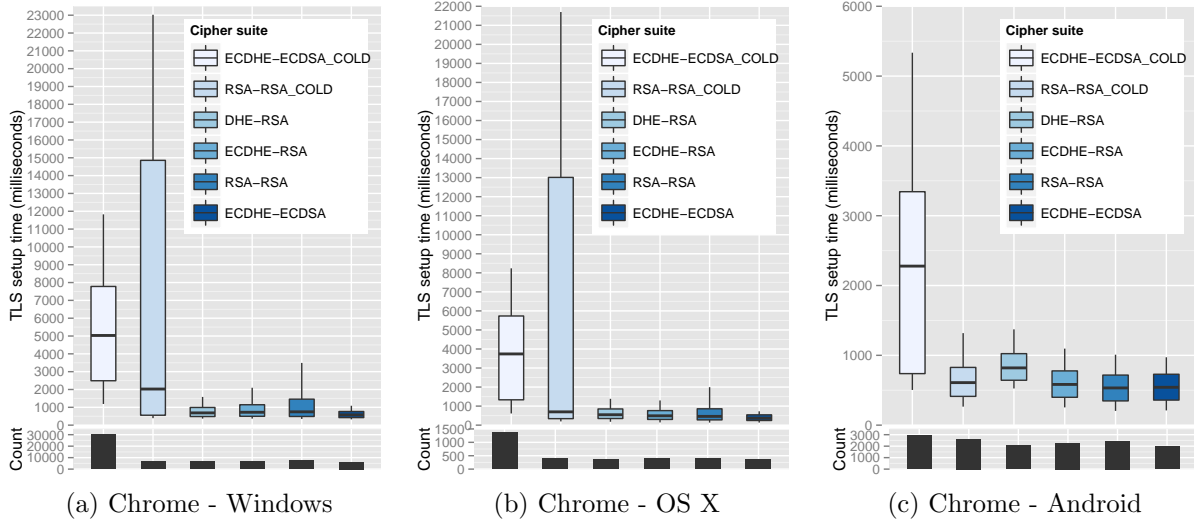


Figure 6.2: Comparison of **TLS setup times** in Chrome browsers on Windows, OS X and Android. The box plots show the 10th, 25th, 50th, 75th and 90th percentiles of measured TLS setup times for each cipher suite. The corresponding bar charts show the number of unique clients that successfully completed each test.

TLS setup times. Figures 6.2a-c show the TLS setup times of different cipher suites in Chrome browsers on Windows, OS X and Android. When comparing client-side latencies, smaller values mean better performance. Upon first glance at Figure 6.2a, the medians of ECDHE-ECDSA_COLD and RSA-RSA_COLD are both substantially higher than the other configurations. Unsurprisingly, the two “cold” connections result in longer latencies possibly due to performing Online Certificate Status Protocol (OCSP) lookups to check validity, while the subsequent tests may enjoy a warm OCSP cache. The number of unique clients that completed ECDHE-ECDSA_COLD appeared to be the highest because it was always tested first (and many users do not stay on the page long enough for other tests to complete).

While the performances of DHE-RSA, ECDHE-RSA, RSA-RSA and ECDHE-ECDSA were similar, we noticed that the ECDHE-ECDSA setup consistently performed the fastest of all setups, resulting in a median of 366 milliseconds (and a 90th percentile of 1088 mil-

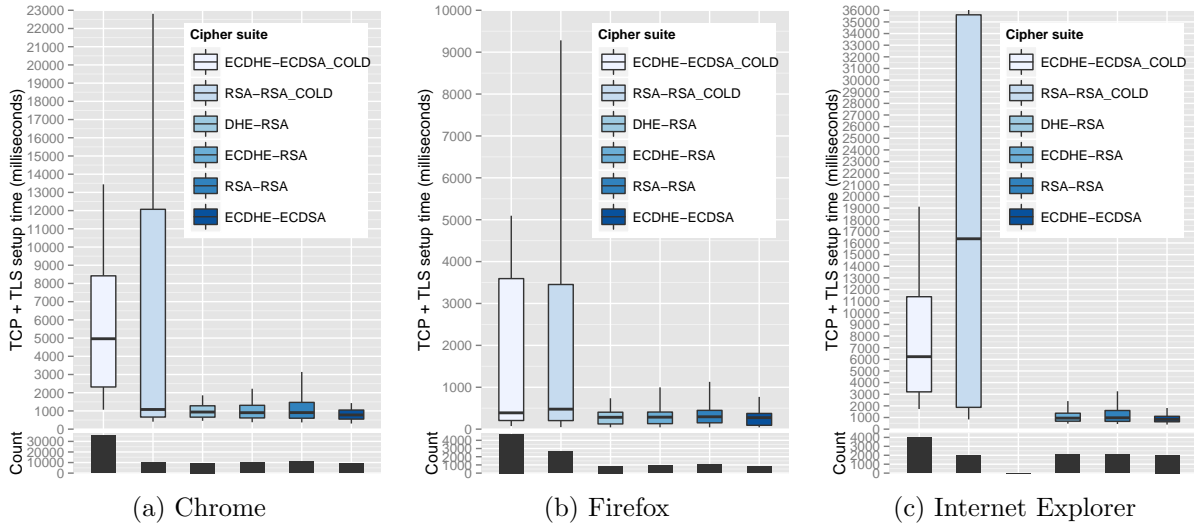


Figure 6.3: Comparison of **TCP + TLS setup times** in Chrome, Firefox and Internet Explorer browsers (on all platforms).

liseconds). This suggests that deploying forward secrecy actually improves performance on the client over RSA-based setups with no forward secrecy. Encouragingly, we observe very similar trends on OS X (in Figure 6.2b) and Android (in Figure 6.2c), where the medians of TLS setup times for ECDHE-ECDSA were consistently the smallest. The results for Android show that mobile devices (typically with less computational power) might also benefit from ECC-based forward secrecy. On the other hand, DHE-RSA performed the slowest on Android mobile with a median of 820 milliseconds.

TCP + TLS setup times. We compare the TCP + TLS setup times of different cipher suites in Figures 6.3a-c for Chrome, Firefox and Internet Explorer browsers on all platforms. The coarser TCP + TLS setup time includes possibly more noise incurred by the extra round-trips of the TCP handshake. In Figure 6.3c, we do not have any results for DHE-RSA since it was not supported in Internet Explorer.

As a sanity check, the TCP + TLS results for Chrome in Figure 6.3a were basically in line with the TLS results in Figure 6.2a, where the ECDHE-ECDSA was the fastest of

all cipher suites. For Firefox, we did not observe significant differences between the client latency medians for each cipher suite in Figure 6.3b. We did not find any cipher suite that performed particularly slower. Unlike Chrome (where ECDHE-ECDSA_COLD took roughly 4 seconds longer than other cipher suites in median), the performance of ECDHE-ECDSA_COLD in Firefox (a median of only 389 milliseconds) was not significantly slower than ECDHE-ECDSA. Upon further investigation, we believe that “cold” connections in Firefox were often faster than other browsers because Firefox maintains its own root CA store (rather than rely on the operating system’s root CA store). The underlying cause is that not all legitimate root CA certificates are pre-installed on the system’s CA root store, in particular on Windows (we verified that Symantec’s ECDSA root certificate is not pre-installed on Windows 8 or Windows Server 2012). When encountering an unseen root CA certificate (in non-Firefox browser), the system attempts to fetch the root certificate over-the-air. As a result, the ECDHE-ECDSA_COLD setup performs faster in Firefox browsers. Nevertheless, fetching a new root certificate is a one-time cost. As more TLS servers deploy ECDSA certificate chains, clients will eventually have downloaded the ECDSA root certificate after visiting any of those sites and will have paid off this one-time cost.

6.3 Discussion

6.3.1 Forward secrecy is free

Our experiments suggest that the performance-based arguments against deploying forward secrecy are no longer valid. ECDHE-based key exchange, which provides forward secrecy, can be faster than basic RSA-2048 key exchange which does not. The reason for the performance improvement is the replacement of an expensive RSA-2048 decryption with faster `secp256r1` elliptic curve operations. As we transition to longer RSA keys, such as

RSA-3072 or RSA-4096, the performance advantage of ECDHE will become even more pronounced. These results suggest that sites should migrate to ECDHE for both security and performance reasons.

6.3.2 RSA vs. ECDSA authentication

A common practice today for deploying ECDHE-based forward secrecy is to use elliptic curves for key exchange, but use RSA signatures for server-side authentication. From a security standpoint this is an undesirable setup: a weakness discovered in *either* algorithm will defeat the security of TLS at the site. A-priori, the likelihood of a weakness discovered in one of two algorithms is far greater than the likelihood of an attack on a single algorithm. Consequently, due to the desire to move to ECDHE key exchange, there is a strong argument for sites to move to certificates for ECDSA public keys.

To understand the risk of using both RSA and ECDHE (called ECDHE-RSA) compared to only relying on elliptic curve cryptography (as in ECDHE-ECDSA), consider the following three possibilities:

1. both RSA and the NIST curve `secp256r1` provide adequate security,
2. curve `secp256r1` is secure, but RSA is not,
3. RSA is secure, but curve `secp256r1` is not.

In Case 1, both ECDHE-RSA and ECDHE-ECDSA are secure. In Case 3, both ECDHE-RSA and ECDHE-ECDSA are insecure. However, in Case 2, ECDHE-RSA is insecure but ECDHE-ECDSA is still secure. Table 6.2 lists the resulting security of ECDHE-RSA and ECDHE-ECDSA in each of the three cases. The table suggests that ECDHE-ECDSA incurs less risk than ECDHE-RSA since there is a scenario where ECDHE-ECDSA is secure, but ECDHE-RSA is not. The converse cannot happen. Given the desire to

	ECDHE-RSA	ECDHE-ECDSA
RSA and secp256r1 both secure	secure	secure
secp256r1 secure, RSA insecure	insecure	secure
RSA secure, secp256r1 insecure	insecure	insecure

Table 6.2: Comparing ECDHE-RSA and ECDHE-ECDSA

use ECDHE, this is an argument for moving to elliptic curve public keys for server-side authentication.

To properly move to ECDSA signatures, CAs will need to sign those certificates with ECDSA signatures along the entire certification chain. The security of TLS key exchange will then only depend on the hardness of a single algebraic problem instead of two. Only time will tell whether the elliptic curve discrete logarithm problem (on the NIST curve **secp256r1**) is indeed as hard as we currently believe.

Note that moving to ECDSA public keys means that during the ECDHE key exchange the server will need to generate an ECDSA signature. The ECDSA signature algorithm requires strong randomness: bias in the random generator can lead to exposure of the secret signing key [122]. Therefore, when moving to ECDSA public keys servers will need to ensure an adequate source of randomness. An alternative proposal, which is not frequently used, is to derive the ECDSA randomness by applying a pseudo-random function such as HMAC to the message to be signed, where the PRF secret key is stored along with the signing key.

6.4 Related Work

Coarfa et al. [25] profiled TLS web servers with trace-driven workloads in 2002, showing that the largest performance cost on the TLS web server is from the RSA operations, and suggested that TLS overhead will diminish as CPUs become faster. Gupta et al. [123]

showed that TLS server throughput can increase by 11% to 31% when using ECDSA signatures (with fixed ECDH key exchange) over RSA signatures (with RSA key exchange). Note that none of the measured cipher suites support forward secrecy. Bernat [124] evaluated RSA, DHE and ECDHE key exchanges over 1,000 handshakes and reported a 15% server overhead for using ECDHE-256 over RSA-2048 key exchange. All of the measurements used RSA signatures, and not ECDSA signatures. Our controlled experiment concurs with previous studies while providing data points for a wider range of cipher suites that are currently recommended. Our ad experiment is the first to measure client-side TLS latencies in real-world.

Chapter 7

Conclusion

In this thesis, we focused on new real-world measurements of misbehaving intermediaries in the wild, and new efforts to improve web security by reducing the cost of TLS. The contributions of this thesis include:

- We demonstrate a new class of attacks that poisons the HTTP caches of transparent proxies. Our ad experiments indicate that roughly 7% of Internet users are vulnerable to Auger’s IP hijacking attacks, while 0.2% are vulnerable to our cache poisoning attacks. In response, the HTML5 WebSocket protocol has adopted a variant of our proposal to prevent these attacks.
- We introduce a method for websites to detect TLS man-in-the-middle attacks. We conduct the first analysis of forged TLS certificates in the wild, by measuring over 3 million TLS connections on a popular global website. Our results indicate that 0.2% of the TLS connections analyzed were tampered with forged certificates.
- We propose certificate prefetching and prevalidation in browsers to significantly speed up the full TLS handshake. We evaluate OCSP responders in the wild, including measurements of the response times and validity durations.

- We demonstrate the feasibility of short-lived certificates, which ensures that certificate authorities can control revocation, without imposing a performance penalty.
- We provide a performance evaluation of various TLS cipher suites for servers that support forward secrecy. We conduct the first ad experiment to measure client-side TLS connection times of various TLS cipher suites.

Bibliography

- [1] J. Ruderman, “Same-origin policy.” https://developer.mozilla.org/en-US/docs/Web/JavaScript/Same_origin_policy_for_JavaScript.
- [2] E. Butler, “Firesheep.” <http://codebutler.com/firesheep>.
- [3] R. Auger, “Socket capable browser plugins result in transparent proxy abuse,” 2010. http://www.thesecuritypractice.com/the_security_practice/TransparentProxyAbuse.pdf.
- [4] L.-S. Huang, E. Y. Chen, A. Barth, E. Rescorla, and C. Jackson, “Talking to yourself for fun and profit,” in *Proceedings of the Web 2.0 Security and Privacy*, 2011.
- [5] T. Dierks and E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.2.” RFC 5246 (Proposed Standard), Aug. 2008.
- [6] S. Souders, “WPO – Web Performance Optimization,” 2010. <http://www.stevesouders.com/blog/2010/05/07/wpo-web-performance-optimization/>.
- [7] C. Jackson, “Improving browser security policies.” PhD thesis, Stanford University, 2009.
- [8] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh, “Protecting browsers from DNS rebinding attacks,” in *Proceedings of the ACM Conference on Computer and Communications Security*, 2007.

- [9] D. D. Edward, E. W. Felten, and D. S. Wallach, “Java security: From hotjava to netscape and beyond,” in *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, 1996.
- [10] H. Wang, X. Fan, J. Howell, and C. Jackson, “Protection and communication abstractions for web browsers in mashups,” in *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007.
- [11] P. Uhley, “Setting up a socket policy file server.” http://www.adobe.com/devnet/flashplayer/articles/socket_policy_files.html, Apr. 2008.
- [12] Adobe, “White paper: Adobe flash player 10 security,” 2008. http://www.adobe.com/devnet/flashplayer/articles/flash_player10_security_wp.html.
- [13] M. Zalewski, “Browser security handbook.” <http://code.google.com/p/browsersec/wiki/Main>.
- [14] A. van Kesteren, “Cross-Origin Resource Sharing,” 2010. <http://www.w3.org/TR/cors/>.
- [15] I. Hickson, “The Web Sockets API,” 2009. <http://www.w3.org/TR/websockets/>.
- [16] A. Russell, “Comet: Low Latency Data for the Browser,” 2006. <http://infrequently.org/2006/03/comet-low-latency-data-for-the-browser/>.
- [17] M. Nottingham, “What proxies must do.” https://www.mnot.net/blog/2011/07/11/what_proxies_must_do, 2011.
- [18] H. Adkins, “An update on attempted man-in-the-middle attacks.” <http://googleonlinesecurity.blogspot.com/2011/08/update-on-attempted-man-in-middle.html>.

- [19] P. Eckersley, “A Syrian man-in-the-middle attack against Facebook.” <https://www.eff.org/deeplinks/2011/05/syrian-man-middle-against-facebook>, May 2011.
- [20] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, “Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile.” RFC 5280 (Proposed Standard), May 2008.
- [21] A. O. Freier, P. Karlton, and P. C. Kocher, “The Secure Sockets Layer (SSL) Protocol Version 3.0.” RFC 6101 (Historic), Aug. 2011.
- [22] M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams, “X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP.” RFC 2560 (Proposed Standard), June 1999.
- [23] E. Stark, L.-S. Huang, D. Israni, C. Jackson, and D. Boneh, “The case for prefetching and prevalidating TLS server certificates,” in *Proceedings of the 19th Network and Distributed System Security Symposium*, 2012.
- [24] I. Grigorik, *High Performance Browser Networking*. ”O’Reilly Media, Inc.”, 2013. http://chimera.labs.oreilly.com/books/12300000000545/ch04.html#TLS_COMPUTATIONAL_COSTS.
- [25] C. Coarfa, P. Druschel, and D. S. Wallach, “Performance analysis of TLS Web servers,” *ACM Trans. Comput. Syst.*, vol. 24, pp. 39–69, Feb. 2006.
- [26] E. Lawrence, “Https caching and internet explorer.” EricLaw’s IEInternals blog, Apr. 2010.

- [27] C. Biesinger, “Bug 531801,” Nov. 2009. bugzilla.mozilla.org/show_bug.cgi?id=531801.
- [28] W. Chan, “HTTP/2 considerations and tradeoffs,” 2014. <https://insouciant.org/tech/http-slash-2-considerations-and-tradeoffs/>.
- [29] “Comodo Report of Incident - Comodo detected and thwarted an intrusion on 26-MAR-2011.” <http://www.comodo.com/Comodo-Fraud-Incident-2011-03-23.html>, Mar. 2011.
- [30] “DigiNotar reports security incident.” http://www.vasco.com/company/about_vasco/press_room/news_archive/2011/news_diginotar_reports_security_incident.aspx, Aug. 2011.
- [31] J. Roskind, “DNS Prefetching (or Pre-Resolving),” 2008. <http://blog.chromium.org/2008/09/dns-prefetching-or-pre-resolving.html>.
- [32] R. Khare and S. Lawrence, “Upgrading to TLS Within HTTP/1.1.” RFC 2817 (Proposed Standard), May 2000.
- [33] V. Anupam, A. Mayer, K. N. an Benny Pinkas, and M. K. Reiter, “On the security of pay-per-click and other web advertising schemes,” in *Proceedings of the 8th International Conference on World Wide Web*, 1999.
- [34] BuiltWith, “Google Analytics Usage Statistics,” 2011. <http://trends.builtwith.com/analytics/Google-Analytics>.
- [35] E. Rescorla, “HTTP Over TLS.” RFC 2818 (Informational), May 2000. Updated by RFC 5785.

- [36] D. Eastlake 3rd and A. Panitz, “Reserved Top Level DNS Names.” RFC 2606 (Best Current Practice), June 1999.
- [37] M. Stachowiak, “Re: [hybi] handshake was: The websocket protocol issues,” 2010. <http://www.ietf.org/mail-archive/web/hybi/current/msg04379.html>.
- [38] J. Tamplin, “Sample code for evaluation of WebSocket draft proposals,” 2011. <http://code.google.com/p/websocket-draft-eval/>.
- [39] C. Heilmann, “WebSocket disabled in Firefox 4,” 2010. <http://hacks.mozilla.org/2010/12/websockets-disabled-in-firefox-4/>.
- [40] A. van Kesteren, “Disabling the WebSocket Protocol,” 2010. <http://annevankesteren.nl/2010/12/websocket-protocol-vulnerability>.
- [41] C. Caldato, “The Updated WebSockets Prototype,” 2011. <http://blogs.msdn.com/b/interoperability/archive/2011/02/09/the-updated-websockets-prototype.aspx>.
- [42] I. Fette and A. Melnikov, “The WebSocket Protocol.” RFC 6455 (Proposed Standard), Dec. 2011.
- [43] J. Topf, “Html form protocol attack,” 2001. <http://www.remote.org/jochen/sec/hfpa/hfpa.pdf>.
- [44] C. Bueno, “HTTP Cache Poisoning via Host Header Injection,” 2008. <http://carlos.bueno.org/2008/06/host-header-injection.html>.
- [45] A. Klein, “Divide and conquer - HTTP response splitting, web cache poisoning attacks, and related topics,” 2004. http://packetstormsecurity.org/papers/general/whitepaper_httpresponse.pdf.

- [46] S. E. Schechter, R. Dhamija, A. Ozment, and I. Fischer, “The emperor’s new security indicators,” in *Proceedings of the IEEE Symposium on Security and Privacy*, 2007.
- [47] J. Sunshine, S. Egelman, H. Almuhiemedi, N. Atri, and L. F. Cranor, “Crying wolf: an empirical study of SSL warning effectiveness,” in *Proceedings of the 18th USENIX Security Symposium*, 2009.
- [48] D. Akhawe and A. P. Felt, “Alice in warningland: A large-scale field study of browser security warning effectiveness,” in *Proceedings of the 22nd USENIX Security Symposium*, 2013.
- [49] A. P. Felt, H. Almuhiemedi, S. Consolvo, and R. W. Reeder, “Experimenting at scale with Google Chrome’s SSL warning,” in *Proceedings of the ACM Conference on Human Factors in Computing Systems*, 2014.
- [50] “TURKTRUST Public Announcements.” <http://turktrust.com.tr/en/kamuoyu-aciklamasi-en.html>, Jan. 2013.
- [51] Team Furry, “TOR exit-node doing MITM attacks.” <http://www.teamfurry.com/wordpress/2007/11/20/tor-exit-node-doing-mitm-attacks/>.
- [52] Electronic Frontier Foundation, “The EFF SSL Observatory.” <https://www.eff.org/observatory>.
- [53] R. Holz, L. Braun, N. Kammenhuber, and G. Carle, “The SSL landscape: a thorough analysis of the x.509 PKI using active and passive measurements,” in *Proceedings of the 11th ACM SIGCOMM Conference on Internet Measurement*, 2011.

- [54] D. Akhawe, B. Amann, M. Vallentin, and R. Sommer, “Here’s my cert, so trust me, maybe? Understanding TLS errors on the web,” in *Proceedings of the International Conference on World Wide Web*, 2013.
- [55] Z. Durumeric, J. Kasten, M. Bailey, and J. A. Halderman, “Analysis of the HTTPS certificate ecosystem,” in *Proceedings of the 13th ACM SIGCOMM Conference on Internet Measurement*, 2013.
- [56] S. Chen, Z. Mao, Y.-M. Wang, and M. Zhang, “Pretty-Bad-Proxy: An overlooked adversary in browsers’ HTTPS deployments,” in *Proceedings of the IEEE Symposium on Security and Privacy*, 2009.
- [57] M. Marlinspike, “sslsniff.” <http://www.thoughtcrime.org/software/sslsniff>.
- [58] “StatOwl.” <http://www.statowl.com>.
- [59] S. Fahl, M. Harbach, H. Perl, M. Koetter, and M. Smith, “Rethinking SSL development in an appified world,” in *Proceedings of the ACM Conference on Computer and Communications Security*, 2013.
- [60] R. A. Sandvik, “Security vulnerability found in Cyberoam DPI devices (CVE-2012-3372).” <https://blog.torproject.org/blog/security-vulnerability-found-cyberoam-dpi-devices-cve-2012-3372>, July 2012.
- [61] Y. N. Pettersen, “Suspected malware performs man-in-the-middle attack on secure connections.” <http://my.opera.com/securitygroup/blog/2012/05/16/suspected-malware-performs-man-in-the-middle-attack-on-secure-connections>, May 2012.

- [62] C. Reis, S. D. Gribble, T. Kohno, and N. C. Weaver, “Detecting in-flight page changes with web tripwires,” in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, 2008.
- [63] S. Stamm, B. Sterne, and G. Markham, “Reining in the web with content security policy,” in *Proceedings of the 19th International Conference on World Wide Web*, 2010.
- [64] D. Wendlandt, D. Andersen, and A. Perrig, “Perspectives: Improving SSH-style host authentication with multi-path probing,” in *Proceedings of the USENIX Annual Technical Conference*, 2008.
- [65] M. Marlinspike, “SSL and the future of authenticity,” in *Black Hat USA*, 2011.
- [66] K. Engert, “DetecTor.” <http://detector.io/DetecTor.html>.
- [67] M. Alicherry and A. D. Keromytis, “Doublecheck: Multi-path verification against man-in-the-middle attacks,” in *IEEE Symposium on Computers and Communications*, 2009.
- [68] R. Holz, T. Riedmaier, N. Kammenhuber, and G. Carle, “X. 509 forensics: Detecting and localising the SSL/TLS men-in-the-middle,” in *Proceedings of the European Symposium on Research in Computer Security*, 2012.
- [69] J. Hodges, C. Jackson, and A. Barth, “HTTP Strict Transport Security (HSTS).” RFC 6797 (Proposed Standard), Nov. 2012.
- [70] C. Jackson and A. Barth, “ForceHTTPS: protecting high-security web sites from network attacks,” in *Proceedings of the 17th International Conference on World Wide Web*, 2008.

- [71] M. Marlinspike, “New techniques for defeating SSL/TLS,” in *Black Hat DC*, 2009.
- [72] N. Nikiforakis, Y. Younan, and W. Joosen, “Hproxy: Client-side detection of ssl stripping attacks,” in *Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 200–218, Springer, 2010.
- [73] C. Evans and C. Palmer, “Public Key Pinning Extension for HTTP,” Internet-Draft draft-ietf-websec-key-pinning-03, IETF, Oct. 2012.
- [74] A. Langley, “Public key pinning.” <http://www.imperialviolet.org/2011/05/04/pinning.html>.
- [75] C. Paya, “Certificate pinning in Internet Explorer with EMET.” <http://randomoracle.wordpress.com/2013/04/25/certificate-pinning-in-internet-explorer-with-emet/>.
- [76] M. Marlinspike and E. T. Perrin, “Trust Assertions for Certificate Keys,” Internet-Draft draft-perrin-tls-tack-02, IETF, Jan. 2013.
- [77] I. Dacosta, M. Ahamad, and P. Traynor, “Trust no one else: Detecting MITM attacks against SSL/TLS without third-parties,” in *Proceedings of the European Symposium on Research in Computer Security*, 2012.
- [78] “Certificate Patrol.” <http://patrol.psyced.org>.
- [79] C. Soghoian and S. Stamm, “Certified lies: detecting and defeating government interception attacks against SSL,” in *Proceedings of the 15th International Conference on Financial Cryptography and Data Security*, 2011.
- [80] “The Sovereign Keys Project.” <https://www.eff.org/sovereign-keys>.

- [81] B. Laurie, A. Langley, and E. Kasper, “Certificate Transparency,” Internet-Draft draft-laurie-pki-sunlight-02, IETF, Oct. 2012.
- [82] T. H.-J. Kim, L.-S. Huang, A. Perrig, C. Jackson, and V. Gligor, “Accountable Key Infrastructure (AKI): A proposal for a public-key validation infrastructure,” in *Proceedings of the International Conference on World Wide Web*, 2013.
- [83] R. Sleevi, “[cabfpub] Upcoming changes to Google Chrome’s certificate handling.” <https://cabforum.org/pipermail/public/2013-September/002233.html>, 2013.
- [84] P. Hoffman and J. Schlyter, “Using Secure DNS to Associate Certificates with Domain Names For TLS,” 2011. IETF Internet Draft.
- [85] P. Hallam-Baker and R. Stradling, “DNS Certification Authority Authorization (CAA) Resource Record.” RFC 6844 (Proposed Standard), Jan. 2013.
- [86] H. Shacham and D. Boneh, “Fast-Track Session Establishment for TLS,” in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2002.
- [87] A. Langley, N. Modadugu, and B. Moeller, “Transport Layer Security (TLS) False Start.” Working Draft, 2010. IETF Internet Draft.
- [88] A. Langley, “Transport Layer Security (TLS) Snap Start.” Working Draft, 2010. IETF Internet Draft.
- [89] P. Mockapetris, “Domain names - implementation and specification.” RFC 1035 (Standard), Nov. 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2845, 3425, 3658, 4033, 4034, 4035, 4343, 5936, 5966.

- [90] E. Cohen and H. Kaplan, “Prefetching the means for document transfer: A new approach for reducing web latency,” in *Proceedings of the IEEE INFOCOM’00 Conference*, 2000.
- [91] Google Chrome Team, “DNS Prefetching.” <http://www.chromium.org/developers/design-documents/dns-prefetching>.
- [92] E. Adar, J. Teevan, and S. T. Dumais, “Large scale analysis of web revisitation patterns,” in *Proceedings of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, 2008.
- [93] E. Herder, “Characterizations of user web revisit behavior,” in *Proceedings of Workshop on Adaptivity and User Modeling in Interactive Systems*, 2005.
- [94] A. Cockburn and B. Mckenzie, “What do web users do? an empirical analysis of web use,” *International Journal of Human-Computer Studies*, vol. 54, pp. 903–922, 2000.
- [95] L. Tauscher and S. Greenberg, “How people revisit web pages: empirical findings and implications for the design of history systems,” *International Journal of Human Computer Studies*, vol. 47, pp. 97–137, 1997.
- [96] K. Hosanagar, R. Krishnan, M. Smith, and J. Chuang, “Optimal pricing of content delivery network (CDN) services,” in *Proceedings of the 37th Annual Hawaii International Conference on System Sciences*, 2004.
- [97] S. Josefsson, “Storing Certificates in the Domain Name System (DNS).” RFC 4398 (Proposed Standard), Mar. 2006.

- [98] M. Wong and W. Schlitt, “Sender Policy Framework (SPF) for Authorizing Use of Domains in E-Mail, Version 1.” RFC 4408 (Experimental), Apr. 2006.
- [99] F. van Heusden, “httping,” 2010. <http://www.vanheusden.com/httping/>.
- [100] A. Langley, “Revocation checking and Chrome’s CRL.” <http://www.imperialviolet.org/2012/02/05/crlsets.html>, 2012.
- [101] Mozilla, “OneCRL.” <https://wiki.mozilla.org/CA:RevocationPlan#OneCRL>, 2014.
- [102] C. Ma, N. Hu, and Y. Li, “On the release of CRLs in public key infrastructure,” in *Proceedings of the 15th USENIX Security Symposium*, 2006.
- [103] A. Langley, “Revocation doesn’t work.” <http://www.imperialviolet.org/2011/03/18/revocation.html>, 2011.
- [104] P. Kehrer, “Defective By Design? - Certificate Revocation Behavior In Modern Browsers.” <http://blog.spiderlabs.com/2011/04/certificate-revocation-behavior-in-modern-browsers.html>, 2011.
- [105] E. Turkal, “Securing Certificate Revocation List Infrastructures,” 2001. http://www.sans.org/reading_room/whitepapers/vpns/securing-certificate-revocation-list-infrastructures_748.
- [106] K. McArthur. <https://twitter.com/#!/KevinSMcArthur/status/110810801446727681>, 2011.
- [107] M. Marlinspike, “New Techniques for Defeating SSL/TLS.” Black Hat DC 2009.

- [108] R. Rivest, “Can we eliminate certificate revocation lists?,” in *Financial Cryptography*, 1998.
- [109] A. Herzberg and H. Yochai, “Minipay: charging per click on the web,” in *Selected papers from the sixth international conference on World Wide Web*, 1997.
- [110] M. Naor and K. Nissim, “Certificate revocation and certificate update,” in *Proceedings of the 7th USENIX Security Symposium*, 1998.
- [111] M. Marlinspike, “Your App shouldn’t suffer SSL’s problems.” <http://blog.thoughtcrime.org/authenticity-is-broken-in-ssl-but-your-app-ha>, 2011.
- [112] IAIK, “CRYPTO Toolkit.” <http://jce.iaik.tugraz.at/>.
- [113] J. Calcote, “Managing a Dynamic Java Trust Store.” <http://jcalcote.wordpress.com/2010/06/22/managing-a-dynamic-java-trust-store/>, 2010.
- [114] Chromium, “SSL Stack.” <http://www.chromium.org/developers/design-documents/network-stack/ssl-stack>, 2010.
- [115] Google, “Safe Browsing API.” <https://developers.google.com/safe-browsing/>.
- [116] Y. N. Pettersen, “New CNNIC EV Root, pubsuffix update, and some blacklisted certificates.” <http://my.opera.com/rootstore/blog/2011/03/31/new-cnnic-ev-root-pubsuffix-update-and-some-blacklisted-certificates>, 2011.
- [117] P. Hallam-Baker, “X.509v3 TLS Feature Extension,” Internet-Draft draft-hallambaker-tlsfeature-04, IETF, 2014.

- [118] B. Smith, “[TLS] OCSP Must Staple.” <http://www.ietf.org/mail-archive/web/tls/current/msg10351.html>, Oct. 2014.
- [119] Codenomicon, “The HeartBleed bug.” <http://heartbleed.com/>, 2014.
- [120] J. Ball, “NSA’s Prism surveillance program: how it works and what it can do.” <http://www.theguardian.com/world/2013/jun/08/nsa-prism-server-collection-facebook-google>, 2013.
- [121] E. Abalea, “Bug 49559 - Patch to add user-specified Diffie-Hellman parameters.” https://issues.apache.org/bugzilla/show_bug.cgi?id=49559.
- [122] P. Nguyen and I. Shparlinski, “The insecurity of the elliptic curve digital signature algorithm with partially known nonces,” *Design Codes Cryptography*, vol. 30, no. 2, pp. 201–217, 2003.
- [123] V. Gupta, D. Stebila, S. Fung, S. C. Shantz, N. Gura, and H. Eberle, “Speeding up secure web transactions using elliptic curve cryptography,” in *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, 2004.
- [124] V. Bernat, “SSL/TLS & Perfect Forward Secrecy.” <http://vincent.bernat.im/en/blog/2011-ssl-perfect-forward-secrecy.html>, 2011.