The Case for Prefetching and Prevalidating TLS Server Certificates

Emily Stark*
Massachusetts Institute of Technology
estark@mit.edu

Lin-Shung Huang
Carnegie Mellon University
linshung.huang@sv.cmu.edu

Dinesh Israni
Carnegie Mellon University
ddi@andrew.cmu.edu

Collin Jackson
Carnegie Mellon University
collin.jackson@sv.cmu.edu

Dan Boneh
Stanford University
dabo@cs.stanford.edu

Abstract

A key bottleneck in a full TLS handshake is the need to fetch and validate the server certificate before establishing a secure connection. We propose a mechanism by which a browser can prefetch and prevalidate server certificates so that by the time the user clicks on an HTTPS link, the server's certificate is immediately ready to be used. Combining this with a recent proposal called Snap Start reduces the TLS handshake to zero round trips. Prefetching and prevalidating certificates improves web security by making it less costly for websites to enable TLS and by removing time pressure from the certificate validation process. We implemented prefetching and prevalidation and studied the effects of four different prefetching strategies on server performance. Along the way we conducted a study of OCSP, a certificate validation mechanism. This data enabled us to evaluate the effectiveness of prefetching and prevalidating in reducing TLS handshake latency. In some cases we show a factor of four speed-up over a full TLS handshake.

1. Introduction

Transport Layer Security (TLS) [15] is used to secure and authenticate sensitive data in transit, but TLS often presents difficulties for both clients and servers. TLS misconfigurations and certificate warnings are common and can result in security vulnerabilities and usability problems [4, 41]. TLS-enabled servers face a heavier load [10] that discourages site-wide use of TLS, thereby exposing users to session hijacking and other exploits [7]. Serving websites over TLS also increases client latency, namely the time until a landing page is loaded and rendered. Even small additions to client latency can impact website traffic, usage, and revenue [40]. In this paper we address the client latency imposed by TLS.

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. The web browser must also validate the server's certificate using certificate revocation protocols such as the Online Certificate Status Protocol (OCSP) [36], adding more latency and leading clients to cache certificate validation results. Because high latency discourages websites from enabling TLS and forces browsers to compromise the freshness of certificate validation, there is a tradeoff between security and user experience. Decreasing TLS handshake latency can encourage wider use of TLS and improve web security.

Recent 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 [39]. More recent proposals work only when the client sends data first, as in the case of HTTP. One such proposal, TLS False Start, reduces the handshake to one round trip when whitelisted secure cipher suites are used [28]. 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 [24]. 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 from the handshake protocol. As we will see, the latency imposed by certificate validation greatly impacts the overall handshake time. Other work has analyzed TLS performance when clients and servers authenticate each other with a pre-shared symmetric key [9], but this is not the case for most

*Work done while at Stanford University.

web traffic.

In this paper, we introduce 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.

The Chromium browser uses *DNS prefetching*, in which DNS resolutions are done long before they are needed. Our work applies prefetching to certificates, which has the additional benefit of enabling certificate validation before a user clicks on a link.

1.1. Contributions

- We propose server certificate prefetching and prevalidation as a mechanism that significantly speeds up the full TLS handshake. We discuss four certificate prefetching strategies: (1) prefetch from DNS as part of a DNS domain-name resolution, (2) prefetch using an HTTP request to the server itself, (3) prefetch with an HTTP request to a content distribution network (CDN), and (4) prefetch using a truncated TLS handshake with the server. Once the server certificate is prefetched, the browser applies prevalidation to the certificate either by consulting a certificate revocation list (CRL) or by communicating with an online OCSP responder.

- We present detailed statistics from 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. This data shows the strong benefits of certificate prevalidation, which eliminates the expensive OCSP check from the critical path. We also identify an interesting attack on private browsing modes that results from the implementation of OCSP in all major browsers other than Firefox.

- We implemented two prefetching methods (HTTP and DNS) in the open-source browser Chromium. Our implementation integrates with an experimental implementation of Snap Start in Chromium to obtain a highly optimized zero round trip TLS handshake protocol. We also implemented server-side Snap Start in OpenSSL to study the effects of prefetching and prevalidation on a TLS server's performance. We present results of experiments comparing multiple prefetching and prevalidation strategies and demonstrate their benefits.

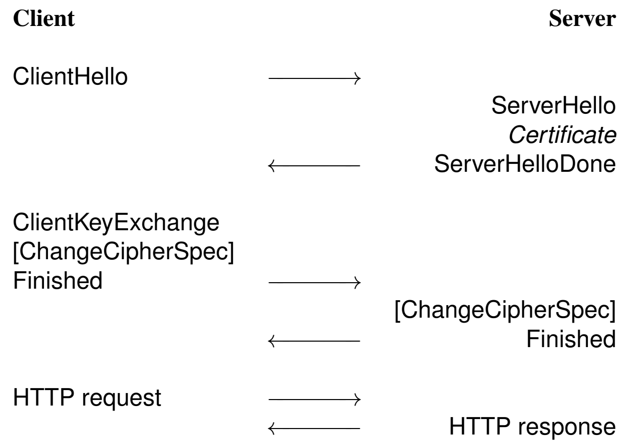

Figure 1. A standard TLS handshake, with RSA key exchange and no client certificate.

2. Background

In this section, we review the features of TLS, Snap Start, DNS, and OCSP that are relevant to certificate prefetching.

2.1. Transport Layer Security

TLS is a protocol for encrypting and authenticating traffic between a client and a server [15]. To set up a secure connection, the client and server perform a handshake in which each party can authenticate itself by providing a certificate signed by a certificate authority. 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.

On the web, TLS provides privacy and data integrity for HTTP traffic between a web browser and a website. Figure 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. These initial messages also allow the client and server to exchange fresh random values used in deriving the session key, which prevents message replay. The client's random value includes the client's clock time. 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. The client only sends the first application data, in this case an encrypted HTTP request, after two round trips between the client and server.

TLS allows connections to be established by resuming previous sessions. If a session is to be resumed in the future, the server provides a session ID in the ServerHelloDone message of the full handshake. To resume a session, the client begins a resume handshake by sending the saved session ID in its ClientHello. An extension called TLS SessionTicket allows session resumption without server-side state [38]. If the client and server both include empty SessionTicket extensions in their Hello messages, then the server sends a NewSessionTicket message after receiving the client's Finished message. The NewSessionTicket contains encrypted and authenticated state that the server needs to resume the session. To resume a session, the client sends its cached session ticket in the SessionTicket extension in its ClientHello. Session tickets are used to enable Snap Start handshakes that can be resumed.

A proposal called TLS False Start, enabled by default in Google Chrome as of version 9 [5], removes one round trip from the TLS handshake [28]. In a False Start handshake, the client sends application data immediately after sending its Finished message, without waiting for the server's Finished message. The server buffers the encrypted application data until after it has sent its Finished message, and then it processes the encrypted record. The False Start proposal argues that, as long as the client has negotiated a secure cipher suite, the encrypted data can only be decrypted by the expected peer. If an attacker has interfered with the handshake, neither the server nor the attacker will be able to decrypt the data that the client sent preemptively.

The hidden costs of TLS handshakes. 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 [29]. 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. Our approach to prevalidating certificates greatly reduces the cost of these handshakes.

Second, some browsers, such as Firefox, are reluctant to cache HTTPS content unless explicitly told to do so using a `Cache-Control: public` header [6]. 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.

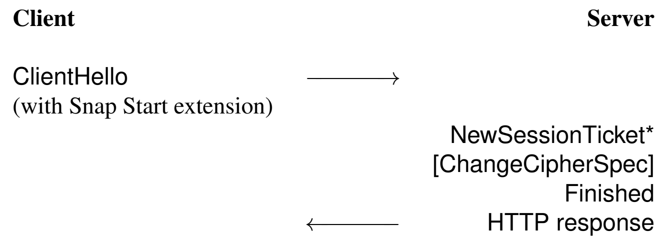

Figure 2. A TLS Snap Start handshake. The client's ClientKeyExchange, ChangeCipherSpec, and Finished messages, as well as the first HTTP request, are all sent in an extension in the ClientHello. Asterisk (*) indicates an optional message.

2.2. TLS Snap Start

Figure 2 shows the message flow of a TLS Snap Start handshake [24]. The client must have performed a full TLS handshake in the past. In this full handshake, the client sends an empty Snap Start extension, and the server echoes a Snap Start extension that includes a selected cipher suite and a value called an orbit. The orbit is made up of eight bytes chosen by the server, and it helps the server synchronize the rejection of replayed messages across multiple geographically separated locations. Since the server does not provide its own random value in a Snap Start handshake, it must keep track of client randoms that it has seen within a certain time interval, and it rejects handshakes that include an orbit different than its own or a client time older than its chosen allowable interval. By assigning a different orbit to each of its geographically separate server locations, a website can ensure that a handshake with one of its server locations cannot be replayed to a server in another location, since the latter server will reject the incorrect orbit.

By caching the server certificate, selected cipher suite, and orbit, the client can later perform a Snap Start handshake. In a Snap Start handshake, the client sends a Snap Start extension in its ClientHello. The extension includes the server's orbit value, twenty "suggested" random bytes, a hash of the server's handshake messages (which the client predicts using its cached information), and TLS ciphertext records, including ClientKeyExchange, ChangeCipherSpec, Finished, and the first HTTP request.

Upon receiving a full Snap Start extension in a ClientHello, the server forms its server random from the twenty suggested random bytes, the orbit, and the time included in the client random. Since the server does not choose its random value, it must prevent replay attacks by rejecting incorrect orbit values, requiring the time in the ClientHello to be within some interval of the server's current clock time, and keeping track of client-suggested random values that it has already seen within this interval. Since the server rejects client times that are older than its allowable interval, the client and server clocks must be synchronized to some degree for a Snap Start handshake to be successful. Since the client must be able to predict the contents of the server's handshake messages, the ServerHello cannot include a session ID. Instead, the connection can use session tickets, since NewSessionTicket is sent after the client's Finished message and the client only needs to predict up to the ServerHelloDone message.

If the client has sent a valid Snap Start extension, the server does not send its handshake messages. The server processes the records in the Snap Start extension to derive the master key and validate the handshake. It then sends its Finished message, processes the first HTTP request, and sends an encrypted HTTP response. Overall, no extra round trips are added to the interaction beyond what is needed to process an unencrypted HTTP request.

Before starting a Snap Start session, if the cached certificate's validation status from a previous handshake has expired, the browser must validate the server's certificate by consulting a CRL or by issuing an OCSP query. Hence, while Snap Start reduces round trips with the web server, the browser must in some cases still communicate with a certificate validation authority before setting up the connection. Our OCSP study (Section 3) shows that this step is quite costly and happens often. Our approach to prevalidation removes this costly step from the critical path, thus enabling the full benefits of Snap Start.

2.3. DNS prefetching

When establishing connections with web servers, the web browser relies on the Domain Name System (DNS) [34] 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 [12]. 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 [17]. 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 [37].

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.

We extend the DNS prefetching architecture in modern browsers to also prefetch and prevalidate TLS server certificates, as we describe in Section 4.3. Our experiments show significant improvements in TLS handshake performance.

2.4. Certificate validation

In the X.509 [14] public key infrastructure, a certificate issued by a certificate authority (CA) binds a public key with an individual, commonly a domain name. Web browsers determine the authenticity of a HTTPS website by validating the server certificate obtained via the TLS handshake. Fundamentally, a server 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 well-rooted, or bundled with a certificate chain leading to a trusted root CA.

To determine the validity period of a public key certificate, each certificate specifies the date it becomes valid, and the date it expires. 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) and the Online Certificate Status Protocol (OCSP).

2.4.1. CRL. A CRL [14] is a list that contains serial numbers of certificates that are revoked, signed by a CA. Web browsers may download CRLs published by CAs to verify the revocation status of a certificate. The location of the CRL for a certificate is indicated by the CRL distribution point extension. However, downloading a complete list of all unexpired certificates that have been revoked can be cumbersome, especially for large CAs, which can issue CRLs that are a megabyte in size [43]. Alternatively, the CAs may issue delta CRLs which only list the certificates whose revocation statuses have changed since a previous complete CRL cached by the client. Delta CRL requires support on both CAs and clients and has not been widely deployed in practice.

2.4.2. OCSP. OCSP [36] was introduced as an alternative to CRLs. Web browsers can check whether a specific certificate has been revoked by asking the OCSP responder for that certificate. The location of the OCSP responder for each certificate is indicated by the authority information access (AIA) extension. Since an OCSP response is typically smaller than a CRL, it is more feasible for a CA (or the delegated OCSP signing authority) to issue OCSP responses with shorter validity intervals (10 days maximum recommended by Mozilla [35], and 2 weeks recommended by Microsoft [32]), defined with the thisUpdate and nextUpdate fields.

In practice, we observe that the actual OCSP response caching behaviors may vary on different web browsers and operating systems. On Windows, Internet Explorer, Safari, and Google Chrome all use CryptoAPI to perform certificate validation, which shares OCSP response caches maintained by the operating system and cleared on expiration. Similarly on Mac OS X, Safari and Google Chrome both use Security Framework API and share OCSP response caches maintained by the operating system and cleared on expiration. For Opera on all platforms, OCSP responses are cached by the browser, which are cleared on expiration and also when the user clears private data. For Firefox on all platforms, OCSP responses are cached by the browser using NSS, independent of operating system caches. In particular, the OCSP cache is stored in memory and cleared when the program closes, or on expiration. In addition, Firefox forces a maximum OCSP response lifetime of 24 hours regardless of longer expiration times. On Linux, Google Chrome also uses NSS and stores OCSP caches in memory. Note that shorter OCSP lifetimes may provide better freshness, but induce more frequent OCSP lookups. Furthermore, we discovered that when OCSP checking is performed for the whole certificate chain, multiple OCSP requests are not performed in parallel, which may result in longer delays [21].

Although all major browsers support OCSP checking, recent studies have revealed that the implementations of OCSP checking are inconsistent, in particular the warning prompts and fallback mechanisms on status check failures [16]. Some browsers ignore bogus OCSP responses, while all avoid treating such errors as fatal; otherwise, websites would have to rely on the availability of OCSP responders. Researchers have suggested that current implementations of certificate revocation mechanisms in browsers are flawed due to lenient checking [26], as evidenced during the Comodo and DigiNotar CA security breaches [13, 46], which caused browser vendors to patch their browsers instead of relying on revocation. One possible solution would be OCSP stapling, in which the TLS server provides the OCSP response during the TLS handshake. This would effectively provide fresh OCSP responses and avoid additional OCSP lookups on the client. However, current implementations of OCSP stapling do not support multistapling, needed for intermediate CAs. Even if allowed, the responses might be too large to fit in the server's initial congestion window and result in additional round trips [27].

OCSP is mandatory for extended validation (EV) certificates [8] and EV certificates use dedicated OCSP responders. If both CRL and OCSP extensions are present in the certificates, web browsers will generally prefer to use OCSP rather than download a large CRL.

Regardless of using CRL, OCSP, or OCSP stapling, we propose performing certificate validation during the prefetching phase, such that more strict and frequent validation checking can be obtained without impacting user experience. We note that some browsers do implement prevalidation, either by periodically validating certificates in the disk cache in CryptoAPI [33], or by concurrently validating certificates during the DNS lookup phase for previously visited HTTPS websites in Google Chrome. However, existing prevalidation mechanisms are not effective for unvisited websites, therefore we propose prefetching server certificates in advance. In the case that OCSP checking may be removed in the future due to wider use of short-lived certificates, certificate prefetching will still be beneficial, simply because certificates will expire more frequently and full TLS handshakes will more often be required.

OCSP and private browsing. Most modern browsers implement a private browsing mode, designed to let users visit websites without leaving traces of their visits to these sites on their computer [3]. An attacker who takes control of the user's machine after the user exits private browsing should not be able to determine what the user did while in private mode.

OCSP permits an attack on private browsing modes in all major browsers, except Firefox, on both Windows and Mac OSX. As mentioned before, IE, Chrome, and Safari use the Windows CryptoAPI for certificate validation. When Windows issues an OCSP query, it caches the result as specified by the nextUpdate field. Unfortunately, CryptoAPI provides no interface for removing specific entries from the cache. As a result, when the browser exits private browsing mode it does not remove the newly acquired OCSP responses from the cache. An attacker who wishes to learn what the user did while in private browsing mode need only dump the Windows OCSP cache. The contents of the cache divulge the identity of HTTPS websites visited. A similar attack applies to browsers on Mac OSX who use Apple's Security Framework API.

To give an example we use the Windows certutil tool [31] that can be used to manipulate the OCSP cache. To view the cache, the attacker issues the command

```
certutil -URLcache ocsp
```

on the browser's machine. A truncated sample output is

```
http://ocsp.thawte.com/MFEwTzBNMEswSTAJBgUrDgM...
http://ocsp.thawte.com/MEUwQzBBMD8wPTAJBgUrDgM...
```

In this example, Thawte's OCSP responder was queried twice, and the query paths contain the certificates' serial numbers. The attacker can search for a web site whose certificate's serial number matches the query and learn what web sites were visited while the user was in private mode. Fixing this problem may be difficult since it requires changes to CryptoAPI. Once fixed, browsers will need to obtain fresh OCSP responses after switching from private browsing mode to normal mode, and certificate prefetching can help mitigate the impact of these extra checks.

3. OCSP measurements

3.1. Experimental setup

To collect statistics of OCSP responses in the wild, we ran experiments on the Perspectives system [47]. 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.

3.2. Results

3.2.1. OCSP response validity lifetime. Table 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.

3.2.2. OCSP lookup response time. Figure 3 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 measurements, the median OCSP lookup time is 291 ms and the mean is 497.55 ms. Table 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.

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 OCSP lookup (usually blocking the completion of the TLS handshake), and finally the HTTP request-response protocol. As previously

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 1. Validity lifetime of OCSP responses.

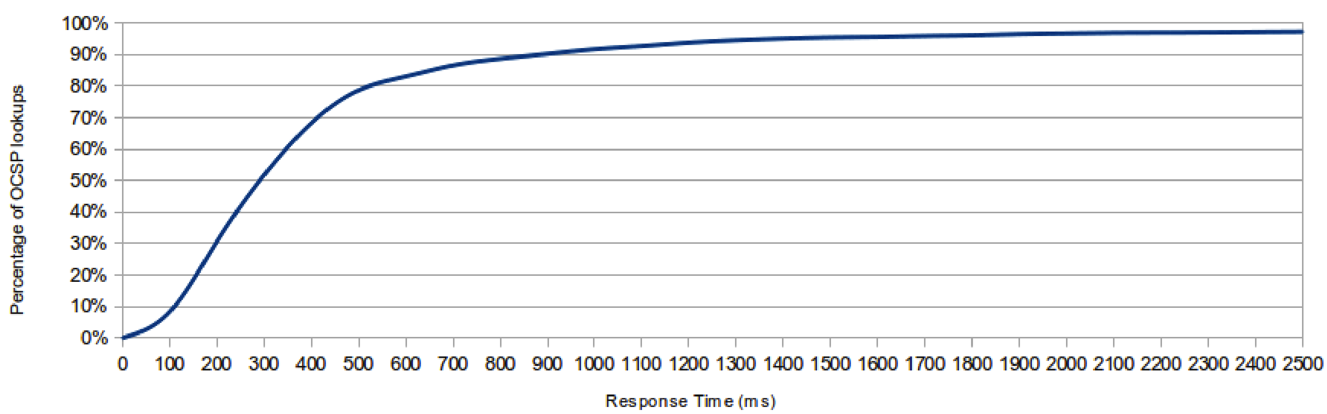

Figure 3. Cumulative distribution of OCSP lookup response times.

OCSP 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 2. Response times of OCSP responders.

introduced, DNS prefetching removes round trips for DNS lookups at the time of user navigation, while TLS False Start removes a round trip for the first HTTP request. In this paper, we propose certificate prefetching along with prevalidating to effectively remove the round trips for the TLS handshake and the OCSP lookup, which may reduce hundreds of milliseconds of perceived latency on average.

3.3. Lessons

Our measurements show that OCSP validation is a significant source of user-perceived latency. Without certificate prefetching and prevalidation, reducing latency requires weakening security (for example, by extending validity periods, or by using cached responses with no nonces). Further, browsers often refuse to treat negative or nonexistent OCSP responses as fatal, meaning that OCSP has little security benefit. For these reasons, Google has recently floated plans to phase out OCSP in its Chrome browser, reverting to pushing CRLs via frequent browser patches [25]. Future directions for certificate validation may also benefit from certificate prefetching. As discussed in Section 2.4.2, short-lived certificates will require more frequent full TLS handshakes. Like OCSP, notary systems such as Perspectives [47] and Convergence [30] require round trips for validation, and the latency of these round trips can be hidden from the user by doing them in advance of the page load.

4. Server certificate prefetching and prevalidation

To enable this reduced latency, a server must 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.

4.1. Benefits of prefetching

4.1.1. 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 [2, 18, 11, 42]. 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.

4.1.2. 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 3 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.

As discussed in Section 2.4.2, modern browsers commonly cache OCSP responses across public and private browsing modes. Further, Opera is the only one of the five browsers that clears the OCSP cache when the user opts to clear all private data. The persistence of OCSP responses is a privacy leak, and we note that, once fixed, certificate prevalidation will become more important because OCSP responses will be cached less frequently.

4.2. 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.

4.2.1. 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.

4.2.2. 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.

4.2.3. 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 [20]. 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.

4.2.4. Prefetching from DNS. Alternatively, 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 [22], 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 A records, 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) [48] 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 [19].

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. Projects such as Perspectives [47] can also help users make correct trust decisions if they receive an invalid certificate.

4.3. 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 preresolves 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. Our patches for Chromium and OpenSSL are available online [1].

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 2.3. 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.

5. Prefetching experiments

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 prevalidated 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.

5.1. 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.

5.1.1. 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.

5.1.2. 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` [45] 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.

5.2. Results

Table 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**

	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 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.

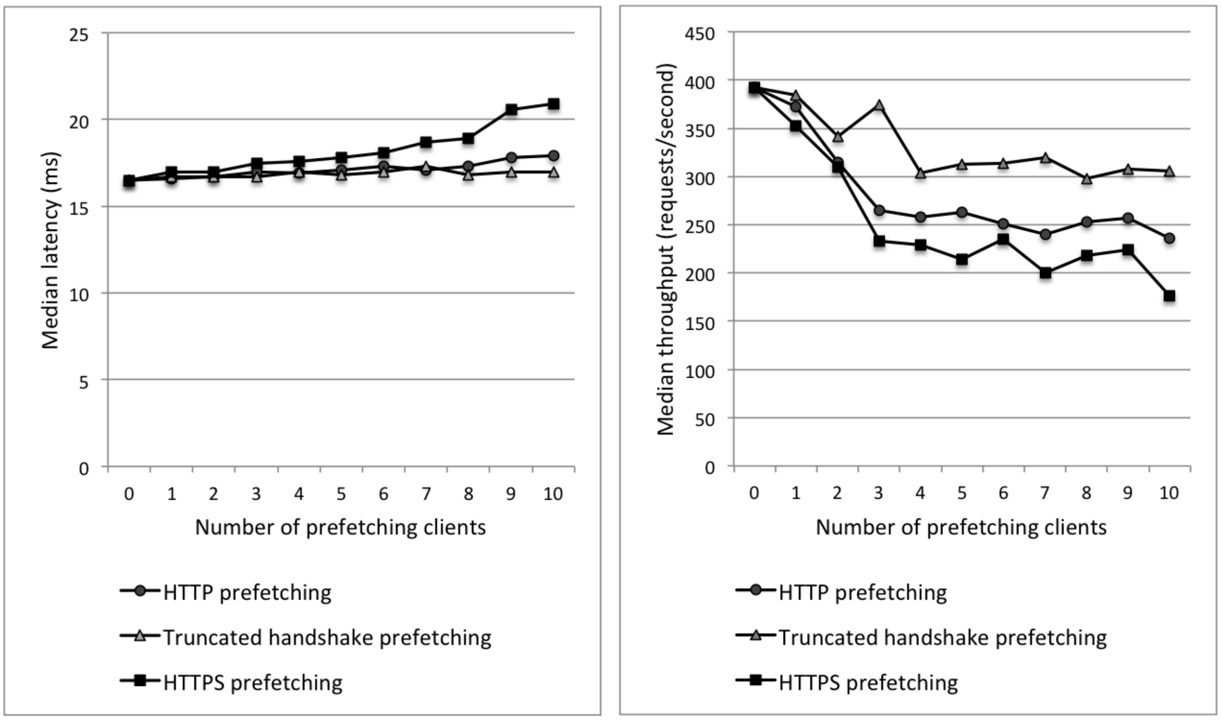

Figure 4. Median latency and throughput for HTTP HEAD requests with different types of prefetching traffic.

three.

Figure 4 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.

In Appendix A, we give data that shows how ten prefetching clients affected the server's performance (latency and throughput) on normal TLS handshakes, Snap Start handshakes with unvalidated certificates, and Snap Start handshakes with prevalidated certificates.

Figure 5 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.

6. 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 certifi-

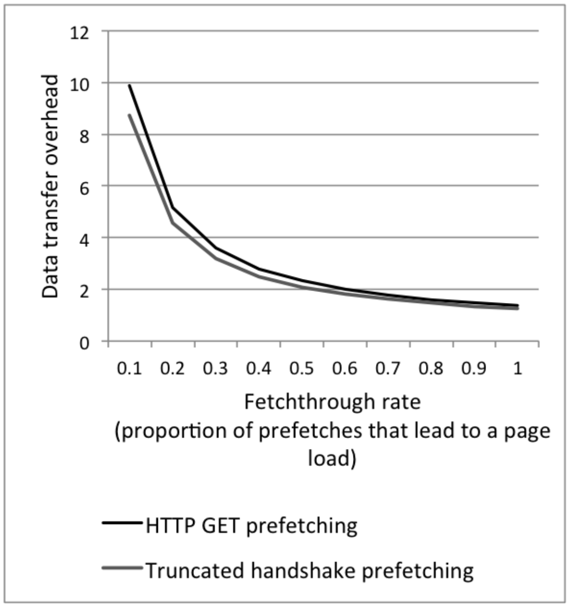

Figure 5. Data transfer overhead for certificate prefetching.

cate. 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 3, show that prevalidating certificates will reduce latency even more in a real-world setting. 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 handshakes 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 5, 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.

6.1. Pros and cons of prefetching methods

Having observed the performance impact of each prefetching method, we consider the benefits and drawbacks of each method and discuss how browsers and servers might choose which method to implement.

6.1.1. Prefetching with a truncated handshake. 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. A truncated handshake requires slightly more client-side code complexity than prefetching via a HTTP GET request directly to the server, since the TLS implementation must be modified to truncate the handshake when prefetching. (For example, in Chromium, we used a `URLFetcher` interface to prefetch a certificate via HTTP, but making a HTTPS request that truncates after receiving ServerHelloDone requires going below this abstraction to modify the TLS implementation.) 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.

6.1.2. Prefetching via HTTP GET. Prefetching via a HTTP request directly to the server is the simplest prefetching method to implement in a browser, but for clients to be able to prefetch via HTTP, the server must explicitly enable it by creating a file with its certificate, orbit, and supported cipher suites.

6.1.3. Prefetching from a CDN. Prefetching certificates from a CDN has no impact on server performance, and for a server that already uses a CDN to distribute static content, enabling prefetching via CDN will be no more difficult than enabling prefetching for the HTTP GET method. However, the main drawback of prefetching from CDNs, as discussed in Section 4.2.3, is that the browser cannot know from which CDN to prefetch the certificate for a particular website, so the browser must send requests to multiple CDNs to increase its probability of a successful prefetch. These requests can be performed asynchronously, but still use more client bandwidth than the other methods. As a compromise, we suggest that a DNS TXT record can hold the location of a server's certificate (whether it is on a CDN or on the server itself), which allows web browsers to prefetch certificates from CDNs without making requests to multiple CDNs.

6.1.4. Prefetching from DNS. Like CDN prefetching, DNS certificate prefetching places no additional load on the server, but DNS also uses minimal client bandwidth and it is also a more accessible option for servers that don't already use a CDN. DNS certificate prefetching may be slightly limited by the fact that not all domain registrars and DNS providers support DNS TXT records [23] [44]. DNS prefetching also has the undesirable effects of swelling DNS records and overloading the meaning of TXT records.

7. Conclusion

Client latency from TLS handshakes costs websites in traffic and revenue, and discourages websites from using TLS. Server certificate prefetching and prevalidation can enable abbreviated TLS handshakes and remove certificate validation latency. In our tests, a Snap Start handshake with a prevalidated certificate can be as much as four times faster than a normal TLS handshake. We also found that 74.8% of OCSP lookups took between 100 ms and 600 ms, so for many users in the wild, prefetching enables an even more dramatic speed-up over standard TLS.

Web browsers can prefetch server certificates either from the server itself (via a truncated TLS handshake or a HTTP GET request) or from a third party (a CDN or DNS). While each method of prefetching has benefits and drawbacks, we suggest that using a DNS record to notify the web browser of the server's certificate location may be a flexible and effective compromise.

Certificate prefetching, in addition to decreasing client latency, allows browsers to validate certificates more frequently, since prevalidation does not increase client latency. We hope that certificate prefetching further encourages deployment of Snap Start in web browsers and servers: prefetching makes Snap Start applicable more often, and enables websites to use TLS more widely.

Acknowledgments

We are grateful to Adam Langley, Jim Roskind, and Scott Hollenbeck for reviewing and providing feedback on this work. This work was funded by NSF and a grant from VeriSign.

References

[1] Chromium and openssl source code patches. `http://web.mit.edu/estark/www/prefetching/`.

[2] 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.

[3] G. Aggarwal, E. Bursztein, C. Jackson, and D. Boneh. An analysis of private browsing modes in modern browsers. In *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, Berkeley, CA, USA, 2010. USENIX Association.

[4] D. Balfanz, G. Durfee, P. Alto, and R. E. In search of usable security: Five lessons from the field. *Proceedings of the IEEE Symposium on Security and Privacy*, 2004.

[5] M. Belshe. SSL FalseStart Performance Results, 2011. `http://blog.chromium.org/2011/05/ssl-falsestart-performance-results.html`.

[6] C. Biesinger. Bug 531801, Nov. 2009. `bugzilla.mozilla.org/show_bug.cgi?id=531801`.

[7] E. Butler. Firesheep, 2010. `http://codebutler.com/firesheep`.

[8] CA/Browser Forum. Guidelines for the issuance and management of extended validation certificates, November 2010. `http://www.cabforum.org/Guidelines_v1_3.pdf`.

[9] F. chun Kuo, H. Tschofenig, F. Meyer, and X. Fu. Comparison studies between pre-shared key and public key exchange mechanisms for transport layer security. Technical report, 2006.

[10] C. Coarfa, P. Druschel, and D. S. Wallach. Performance Analysis of TLS Web Servers. In *Proceedings of the Network and Distributed Systems Security Symposium '02*, 2002.

[11] A. Cockburn and B. Mckenzie. What do web users do? an empirical analysis of web use. *International Journal of Human-Computer Studies*, 54:903–922, 2000.

[12] 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.

[13] Comodo. Comodo report of incident - comodo detected and thwarted an intrusion on 26-mar-2011, 2011. `http://www.comodo.com/Comodo-Fraud-Incident-2011-03-23.html`.

[14] 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.

[15] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), Aug. 2008. Updated by RFCs 5746, 5878, 6176.

[16] A. Dimcev. Random SSL/TLS 101 - OCSP/CRL in practice, 2011. `http://www.carbonwind.net/blog/post/Random-SSLTLS-101-OCSPCRL-in-practice.aspx`.

[17] Google Chrome Team. DNS Prefetching. `http://www.chromium.org/developers/design-documents/dns-prefetching`.

[18] E. Herder. Characterizations of user web revisit behavior. In *Proceedings of Workshop on Adaptivity and User Modeling in Interactive Systems*, 2005.

[19] P. Hoffman and J. Schlyter. Using Secure DNS to Associate Certificates with Domain Names For TLS, 2011. IETF Internet Draft.

[20] 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.

[21] L.-S. Huang. Multiple ocsp requests should be performed in parallel, 2010. `https://bugzilla.mozilla.org/show_bug.cgi?id=579606`.

[22] S. Josefsson. Storing Certificates in the Domain Name System (DNS). RFC 4398 (Proposed Standard), Mar. 2006.

[23] S. Kitterman. Domain registrars and dns providers that support txt records, 2008. `http://www.kitterman.com/spf/txt.html`.

[24] A. Langley. Transport Layer Security (TLS) Snap Start. Working Draft, 2010. IETF Internet Draft.

[25] A. Langley. Personal Communication, 2011.

[26] A. Langley. Revocation doesn't work, 2011. `http://www.imperialviolet.org/2011/03/18/revocation.html`.

[27] A. Langley. [websec] revocation check failures for HSTS sites, 2011. `http://www.ietf.org/mail-archive/web/websec/current/msg00296.html`.

[28] A. Langley, N. Modadugu, and B. Moeller. Transport Layer Security (TLS) False Start. Working Draft, 2010. IETF Internet Draft.

[29] E. Lawrence. Https caching and internet explorer. EricLaw's IEInternals blog, Apr. 2010.

[30] M. Marlinspike. Convergence, 2011. `http://convergence.io`.

[31] Microsoft. How certificate revocation works, 2009. `technet.microsoft.com/en-us/library/ee619754(WS.10).aspx`.

[32] Microsoft. Optimizing the Revocation Experience, 2009. `http://technet.microsoft.com/en-us/library/ee619783(WS.10).aspx`.

[33] Microsoft. Pre-Fetching, 2009. `http://technet.microsoft.com/en-us/library/ee619723(WS.10).aspx`.

[34] 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.

[35] Mozilla. CA:Recommended Practices, 2010. `https://wiki.mozilla.org/CA:Recommended_Practices#OCSP`.

[36] 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.

[37] J. Roskind. DNS Prefetching (or Pre-Resolving), 2008. `http://blog.chromium.org/2008/09/dns-prefetching-or-pre-resolving.html`.

[38] J. Salowey, H. Zhou, P. Eronen, and H. Tschofenig. Transport Layer Security (TLS) Session Resumption without Server-Side State. RFC 5077 (Proposed Standard), Jan. 2008.

[39] H. Shacham and D. Boneh. Fast-Track Session Establishment for TLS. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2002.

[40] S. Souders. WPO – Web Performance Optimization, 2010. `http://www.stevesouders.com/blog/2010/05/07/wpo-web-performance-optimization/`.

[41] J. Sunshine, S. Egelman, H. Almuhimedi, N. Atri, and L. F. Cranor. Crying wolf: An empirical study of ssl warning effectiveness. In *Proceedings of the 18th USENIX security symposium*, USENIX Security'09, 2009.

[42] 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*, 47:97–137, 1997.

[43] E. Turkaly. Securing certificate revocation list infrastructure, 2001. `http://www.sans.org/reading_room/whitepapers/vpns/securing-certificate-revocation-list-infrastructures_748`.

[44] Domain registrar limitations for creating txt records. `http://www.google.com/support/a/bin/answer.py?answer=172167`.

[45] F. van Heusden. httping, 2010. `http://www.vanheusden.com/httping/`.

[46] VASCO Data Security International. DigiNotar reports security incident, 2011. `http://www.vasco.com/company/press_room/news_archive/2011/news_diginotar_reports_security_incident.aspx`.

[47] D. Wendlandt, D. G. Andersen, and A. Perrig. Perspectives: Improving ssh-style host authentication with multi-path probing. In *Proceedings of the USENIX Annual Technical Conference (Usenix ATC)*, 2008.

[48] M. Wong and W. Schlitt. Sender Policy Framework (SPF) for Authorizing Use of Domains in E-Mail, Version 1. RFC 4408 (Experimental), Apr. 2006.

A. Additional data from prefetching experiments

Tables 4 and 5 give the data from the prefetching experiments discussed in Section 5. We measured mean and median latency (in milliseconds) and throughput (requests/second). Section 5 shows the effects of prefetching traffic on HTTP GET requests, and these tables show the effects of 10 prefetching clients on different types of TLS handshakes.

	No prefetching traffic		HTTP		Truncated handshake		HTTPS	
	Median	Mean	Median	Mean	Median	Mean	Median	Mean
Snap Start, prevalidated certificate	30.45	35.58	37.65	57.77	32.00	61.82	42.25	61.53
Snap Start	83.40	99.86	84.76	101.14	82.64	104.11	87.20	103.05
Normal TLS	121.82	124.11	126.69	137.00	125.60	213.94	130.84	273.96

Table 4. Latencies, in milliseconds, for different types of TLS handshakes with no prefetching traffic and with ten clients generating HTTP, truncated handshake, and HTTPS prefetching traffic.

	No prefetching traffic		HTTP		Truncated handshake		HTTPS	
	Median	Mean	Median	Mean	Median	Mean	Median	Mean
Snap Start, prevalidated certificate	75.00	72.25	77.00	72.79	72.50	63.89	66.00	62.88
Snap Start	76.00	76.67	70.00	68.54	72.50	70.45	61.50	56.98
Normal TLS	76.00	76.50	74.50	76.81	77.50	77.05	68.50	74.89

Table 5. Throughput, in requests per second, for different types of TLS handshakes with no prefetching traffic and with ten clients generating HTTP, truncated handshake, and HTTPS prefetching traffic.