Client Side Caching for TLS*

Hovav Shacham
hovav@cs.stanford.edu

Dan Boneh
dabo@cs.stanford.edu

Eric Rescorla
ekr@rtfm.com

Abstract

We propose two new mechanisms for caching handshake information on TLS clients. The "fast-track" mechanism provides a client side cache of a server's public parameters and negotiated parameters in the course of an initial, enabling handshake. These parameters need not be resent on subsequent handshakes. Fast-track reduces both network traffic and the number of round trips, and requires no additional server state. These savings are most useful in high latency environments such as wireless networks. The "client side session cache" mechanism allows the server to store an encrypted version of the session information on a client, allowing a server to maintain a much larger number of active sessions in a given memory footprint. Our design is fully backwards compatible with TLS: extended clients can interoperate with servers unaware of our extensions and vice versa. We have implemented our proposal to demonstrate the resulting efficiency improvements.

1 Introduction

TLS is a widely deployed protocol for securing network traffic. It is commonly used for protecting web traffic and some e-mail protocols such as IMAP and POP. Variants of TLS, such as WTLS [11], are used for securing wireless communication. In this paper we consider two modifications to the TLS (and WTLS) handshake protocols. The first, "fast-track," makes the protocol more efficient in terms of bandwidth and number of round trips. Improving the handshake protocol is especially relevant in bandwidth-constrained environments, such as wireless communications, where latency is high and small payload transfers are common. The second, "Client-side session caching" (CSSC), extends TLS' session resumption mechanism to reduce the load on the server. We hope that these extensions will promote

*This is the full version of a paper that appeared in NDSS '02 [9].

the use of TLS in high latency and high load situations and discourage the development of ad-hoc security protocols to address them.

Recall that the TLS protocol [5] incorporates two types of handshake mechanisms: a full handshake, and a resume handshake protocol. The resume handshake protocol is used to reinstate a previously negotiated TLS session between a client and a server. Compared to a full handshake, the resume mechanism significantly reduces handshake network traffic and computation on both ends. A session can only be resumed if the old session is present in the server's session cache. Unfortunately, heavily loaded servers can only store a session for a relatively short time before evicting it from the cache. As a result, a full handshake is often needed even though the client may be willing to resume a previously-negotiated session.

In contrast, clients rarely connect to numerous TLS servers, and could cache information about servers for a longer time. Both of our extensions take advantage of this situation to improve TLS efficiency. The fast-track extension (Section 3) allows the client to cache the server's long-lived parameters, thus bypassing the "discovery" phase of the TLS handshake. The CSSC extension (Section 4) allows the server to export the cost of session caching to the client, allowing the server to maintain a much larger session cache.

2 TLS handshake overview

A TLS handshake has three objectives: (1) to negotiate certain session parameters; (2) to authenticate the server to the client, and optionally the client to the server; and (3) to establish a shared cryptographic secret. The session parameters include the protocol version, the cipher suite, and the compression method. Authentication makes use of a certificate-based public key infrastructure (PKI): servers and clients identify themselves through certificate chains terminating in well-known Certification Authority (CA) certificates.

The standard TLS handshake is summarized in Figure 1. Messages sent by the client are on the left; by the server, the right. Messages appearing in slanted type are only sent in certain configurations; messages in brackets are sent out-of-band. The handshake proceeds, in four flows, as follows.

A client initiates a handshake by sending a ClientHello message. This message includes a suggested protocol version, a list of acceptable cipher suites and compression methods, a client random value used in establishing the shared secret, and (when TLS extensions [1] are used) other extension-

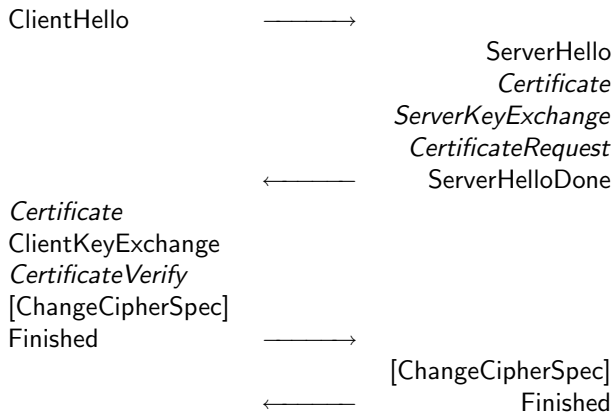

Figure 1: TLS handshake message diagram

specific parameters.

The server replies with a ServerHello message, which selects a protocol version, cipher suite, and compression method, and includes the server random, and extension-specific parameters. The server then sends its certificate chain in the Certificate message. In certain cases, it sends a ServerKeyExchange message with additional information required for establishing the shared secret (for example, the 512-bit export-grade RSA key for RSA export key-exchange). If the server wishes that the client authenticate itself, it sends a CertificateRequest message listing acceptable certificate types and CA names for the client's certificate chain. Finally, it sends a ServerHelloDone message to signal the end of the flow.

If the server requests client authentication, the client begins its response with a Certificate message that includes its certificate chain, and, after the ClientKeyExchange message, a CertificateVerify message that includes its signature on a digest of the handshake messages to that point. The ClientKeyExchange message includes the information necessary to determine the shared secret. (For example, in RSA key exchange, it includes the encryption of a "premaster secret" that is used to calculate the secret.)

Finally, the client sends a ChangeCipherSpec message (which is not a handshake message), signaling its switch to the newly-negotiated parameters and secret key, and sends an encrypted and compressed Finished message that includes a digest of the handshake messages.

The server, in turn, also sends a ChangeCipherSpec message and a Finished message that includes a digest of the handshake messages (up to the client's

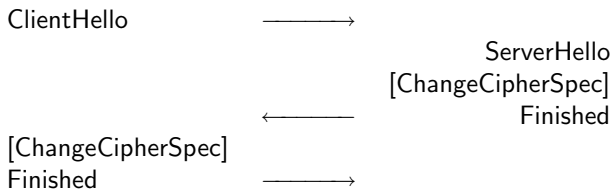

Figure 2: Message diagram for a TLS session resume

Finished message). After this, the client and server can exchange application data over the encrypted, authenticated, and possibly compressed link that has been established.

A server can identify a particular connection by a "session ID," a field in the ServerHello message. By mutual consent, a client and server can resume a connection. The client includes the ID of the session it wishes to resume in its hello message, and the server accepts by including the same ID in its hello. The client and server proceed directly to the ChangeCipherSpec and Finished messages (with the previously-agreed-upon parameters and secrets). This exchange is summarized in Figure 2.

Relative to establishing a new session, resuming a previously-negotiated session saves bandwidth, flows, and computation (since the handshake's expensive cryptographic operations are avoided). However, heavily loaded servers typically keep session IDs in their session cache for only a relatively short while.

We note that our fast-track optimization applies only to full handshakes, not session-resuming handshakes. Hence, fast-track is most effective in environments where short-lived TLS sessions are common, and full handshakes, not resumes, are the norm.

3 Fast-Track

The TLS handshake is optimized for two basic situations. In the case where the peers have never communicated, the full handshake is required for the client to discover the server's parameters. When the peers have communicated very recently, then the resumed handshake can be used. However, an intermediate case, in which the server and the client have communicated at some point in the past but the session has expired or been purged from the server's cache, is quite common. Since server parameters are essentially static, the discovery phase is unnecessary. Fast-track takes advantage of this

observation to improve full TLS handshake efficiency.

Fast-track clients maintain a cache of long-lived server information, such as the server certificate, and long-lived negotiated information, such as the preferred cipher suite. The long-lived cached information allows a reduction in handshake bandwidth: The handshake messages by which a server communicates this information to the client are obviated by the cache, and omitted from the fast-track handshake. Moreover, the remaining messages are reordered, so a fast-track handshake has three flows rather than four. Hence, our fast-track mechanism reduces both network traffic and round trips in the TLS handshake protocol.

By a flow we mean an uninterrupted sequence of messages from one participant in a connection to the other. An ordinary TLS handshake has four flows; our fast-track handshake has three. Because of the design of the TLS protocol, multiple consecutive handshake messages can be coalesced into a single TLS transport-layer message. Thus, when network latency is high, a savings in flows can translate into a savings in time.

The use of fast-track, along with the particular fast-track parameters, is negotiated between clients and servers by means of TLS extensions [1]. Care is taken to ensure interoperability with non-fast-track clients and servers, and to allow graceful fallback to ordinary TLS handshakes when required.

The use of fast-track session establishment gives savings in handshake bandwidth and flows, but does not provide a significant computational speedup relative to ordinary TLS handshakes. It is most useful for bandwidth-constrained, high-latency situations, and those in which application message payloads are small. Fast-track, via a relatively simple, and fully backwards-compatible change to the TLS protocol, improves performance and makes TLS more usable in wireless environments.

We enumerate the long-lived, cacheable items and describe the manner in which they are used in Section 3.1. We discuss some design criteria in Section 3.2. We describe the fast-track handshake protocol in Section 3.3. We then discuss performance, implementation, and security consideration in Sections 3.4, 3.5, and 3.6.

3.1 Cacheable handshake parameters

The savings we achieve through fast-track depend on a client's caching certain long-lived handshake parameters. "Long-lived," in this context, means, first, that they do not change between handshakes (as does, e.g., the server random), and, second, that they are expected not to change except when either the server or client is reconfigured. A client collects these parameters

in the course of an ordinary TLS handshake. In a fast-track handshake, it uses these parameters to craft its messages.

The particular values which a client uses in a fast-track handshake are called the determining parameters for that connection. A server uses information in the client hello message and its own configuration to come up with its own version of the determining parameters for the connection. The two versions must match for the handshake to be successful. Therefore, a fast-track-initiating hello message includes a hash of the determining parameters to allow the server to verify this match, as described in Section 3.3.2.

The long-lived parameters fall into two general categories: (1) those that are features of the server's configuration alone; and (2) those that properly depend on the interaction of the server's configuration with the client's.
In the first category, we include:

- The server's certificate chain;
- The server's Diffie-Hellman group, if any; and
- Whether client authentication is required; if so,
  - Acceptable client certificate types; and
  - Acceptable certificate authorities.

These features of a TLS server's configuration are assumed to change infrequently and thus to be capable of being cached on the client.
In the second category, we include parameters such as:

- The preferred client-server cipher suite; and
- The preferred client-server compression method.

(The cipher suite comprises a key-exchange algorithm, a bulk encryption algorithm, and a MAC algorithm.) These are a function of both the server and client configurations, and are negotiated in a TLS handshake: the client proposes a list for each, and the server chooses.

A client in possession of the above information knows enough to be able to compute a key-exchange message, without any additional input from the server (with one exception discussed below). It is this fact that allows the reordering of the handshake messages.

To participate in ephemeral Diffie-Hellman (EDH) key exchange, a client needs to know the group modulus and generator relative to which the Diffie-Hellman exchange will operate. The description of this group is part of the

ServerKeyExchange message when EDH is used. It is assumed that the server will not often change its EDH group, so a fast-track client can cache the group parameters and use them to send a ClientKeyExchange message during a fast-track handshake. By contrast, a server employing temporary RSA keys for key exchange, in the RSA "export" cipher suites, will typically change its export RSA key quite often. The temporary RSA key, which a client would need for its fast-track key exchange, can be cached only briefly. Accordingly, fast-track explicitly does not support RSA export authentication. Since the RSA export mechanism is being phased out, this is not a serious constraint.

3.2 Design considerations

With significant deployment of legacy TLS clients, incompatible changes to the protocol are unlikely to be accepted. Accordingly, fast-track's design emphasizes interoperability and backwards-compatibility. Fast-track clients and servers must be able to interoperate with TLS servers and clients not capable of using fast-track; they must be able to discover which peers are capable of fast-track; and they must recover gracefully when configurations have changed, falling back on the ordinary TLS handshake protocol.

Through the use of TLS extensions [1], a client and server can, in an ordinary TLS handshake, negotiate the future use of fast-track. A subsequent fast-track connection uses another extension to allow the client and server to ascertain their both using the same unsent, client-cached parameters. Since a client must suggest, and a server must assent to the use of fast-track, the likelihood of a client's attempting to initiate a fast-track connection with a non-fast-track server is minimal.

If a client does attempt to initiate a fast-track connection with a non-fast-track server, it is important that it be alerted of its mistake quickly. A fast-track handshake is initiated through a message that TLS servers not implementing fast-track would reject as invalid. This minimizes confusion resulting from such a mismatch. For servers aware of fast-track, but not wishing to use it, we include a rollback mechanism to allow a server to revert gracefully to an ordinary TLS handshake if its configuration has changed.

3.3 The fast-track handshake

In this section, we describe the actual fast-track handshake protocol. There are two distinct phases. First, in the course of an ordinary TLS handshake, a client and server negotiate and agree on the future use of fast-track, and the client collects the parameters that will allow it to make that future

handshake. Next, the client initiates a fast-track handshake with the server, using the determining parameters from earlier.

Fast-track also defines a mechanism whereby the server can decline the use of fast-track; it would do so, for example, when its configuration has changed, rendering the client's cached determining parameters obsolete. This mechanism is also used for session resumes.

3.3.1 Negotiation of fast-track

A client wishing to engage in a fast-track handshake with a server must first determine whether that server is capable of (and willing to use) fast-track. This is not a problem, since the client must also have completed an ordinary handshake with the server to have obtained the information it needs for the new, fast-track handshake.

The TLS Extensions mechanism [1] provides the machinery for the negotiation. A client proposing the prospective use of fast-track includes the fasttrack-capable extension in its hello; a server assenting to the prospective use includes the same extension in its hello. Such a handshake is referred to as "enabling."

Servers might be reconfigured to disable fast-track, and clients should be alerted of the configuration change as soon as possible; preferably, before they undertake the computationally-heavy early steps of the fast-track handshake.

Accordingly, a client is expected to include in each of its handshakes the fasttrack-capable extension, and attempt a fast-track handshake with a server only if their most recent successful handshake was an enabling one. (Per the specification, the extensions governing a resumed session are those negotiated in the original handshake for that session; a successful resume is therefore not considered a handshake for this purpose.)

3.3.2 Fast-track

To engage in a fast-track handshake, the client and server must agree on certain determining parameters (see Section 3.1). The client obtains these from a previous, enabling handshake. But it and the server must make sure that they expect to use the same parameters. Fast-track ensures this as follows. As part of its fast-track hello message, a client must include, in the fasttrack-hash extension, the SHA-1 hash of the determining parameters. The server builds its own version of the parameters, and ensures that the hashes match.

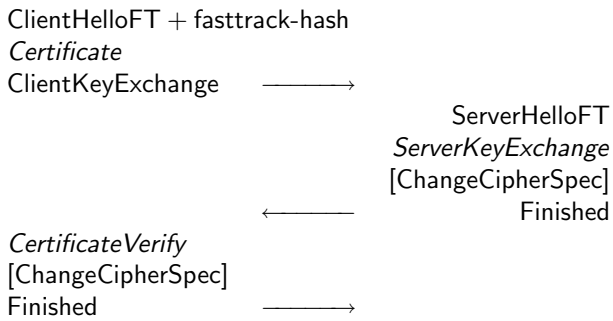

Figure 3: Message diagram for an accepted fast-track handshake

Suppose a client initiates a fast-track handshake, and includes in its hello message both the fasttrack-capable extension and the fasttrack-hash extension, accompanying the latter with a hash of what it thinks are the determining parameters for the handshake. If the server's configuration has changed, but it still wishes to engage in fast-track in the future (with the new, correct parameters), it ought to deny the fast-track, but include the fasttrack-capable extension in its (ordinary) hello message. If, instead, the server's configuration has changed, and it no longer wishes to engage in fast-track in the future, it ought to deny the fast-track, and ought not to include the fasttrack-capable extension in its hello.

The fast-track handshake is summarized in Figure 3. The notation is that employed in Figures 1 and 2, above. Note that the ClientHelloFT message must include the fasttrack-hash extension with a hash of the determining parameters; this requirement is indicated in the first line of the figure.

The exchange omits the server Certificate, CertificateRequest, and ServerHelloDone messages, and requires three flows rather than four. In an ordinary TLS handshake, the server has the last handshake flow; here, the client does. If the client sends the first application data — the typical situation — the savings in flows is magnified, since the client's first application-data flow can be coalesced with its last handshake flow.

The fast-track handshake calls for a nontrivial reordering of the TLS handshake messages. If a client were accidentally to attempt it with a server entirely unaware of fast-track, the client and server might befuddle one another. In keeping with the design goal that the client and server should expeditiously discover whether fast-track is appropriate, the fast-track client hello is made a different message type — ClientHelloFT rather than ClientHello — although the two message types have an identical format.

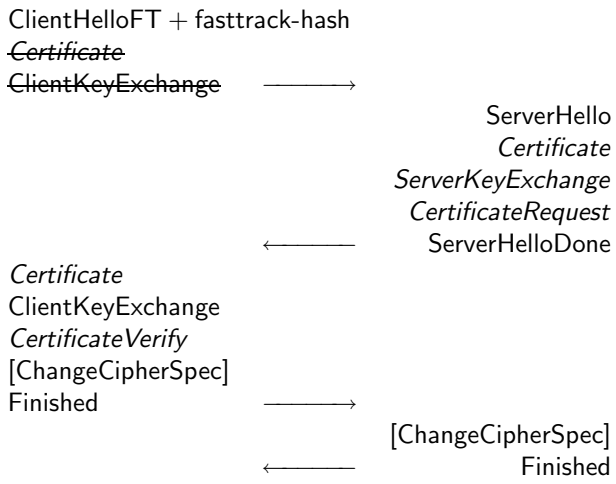

Figure 4: Message diagram for a denied fast-track handshake

A TLS server that is not aware of fast-track will alert the client immediately to the unexpected message type.

The client has enough information to create its key-exchange message without any additional server input, so this message can be sent in the first flow. Once the server has sent its server-random (in its hello) and potentially its key-exchange message, both sides have enough information to calculate the master secret and change cipher suites. The client must wait until it has seen a message from the server before sending its CertificateVerify message, to avoid replay attacks.

3.3.3 Denying fast-track

A server need not agree to engage in a fast-track handshake, even if it had previously assented to one through the fasttrack-capable extension. Fast-track includes a mechanism whereby the server denies an in-progress fast-track handshake, and the client and server revert to an ordinary handshake negotiation.

A server denies fast-track by responding to the client's first flow with a ServerHello message rather than a ServerHelloFT. Its response should be as though the client had initiated the connection through a ClientHello message with the same body as that of the ClientHelloFT message it actually had sent (except without the fasttrack-hash extension). From that point on, the parties carry on an ordinary TLS handshake, conforming to the rules given in the

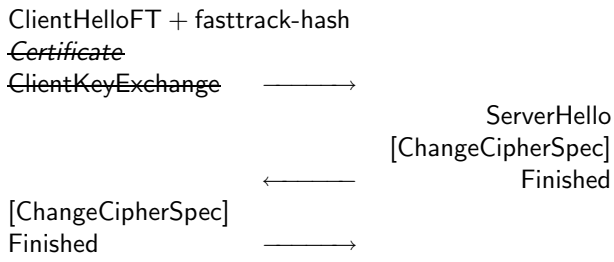

Figure 5: Message diagram for a session resume, with fast-track denied

TLS specification. The other messages sent by the client as part of its first flow are ignored by both parties, and are not included in any handshake message digests.

Figure 4 presents the messages exchanged when fast-track is denied. The notation is the same as employed in Figure 3, with the additional convention that messages printed with strike-through are not included in any handshake digests.

Finally, a server can deny fast-track but proceed with a session-resume if it wishes, and if the client sent a nonempty session-id in its fast-track hello message. Figure 5 gives the message flow in this case, using the same notational conventions as the previous figures. Session resumption provides less of a performance gain to fast-track clients, since they will have already engaged in the time-consuming ClientKeyExchange calculations when the server accepts the resume.

3.4 Performance considerations

The fast-track handshake mechanism reduces the protocol's communication requirements and round trips but has little effect on CPU load. We briefly discuss fast-track's effect on CPU load for both servers and clients A more extensive analysis of the performance of standard TLS in the Web environment is available [4].

The performance of servers employing fast-track is comparable to that of ordinary servers. Fast-track servers avoid sending as many as three messages (Certificate, CertificateRequest, and ServerHelloDone), but none of these involves any computationally-intensive operation; contrariwise, fast-track servers must verify the SHA-1 hash of the determining parameters.

Performance of fast-track clients is slightly improved, with a proper implementation. For example, once a client has validated a server's certificate

chain, it need not revalidate it in the course of a fast-track handshake. Indeed, once it has computed the determining parameters hash which will later be sent to the server, the client may choose to discard the chain, maintaining only the server's public key. Thus, in a fast-track handshake, a client avoids the signature verifications of an ordinary handshake, with a long-term space overhead of only a few hundred bytes for the server key. Note that the client would need to store one server public key (about 128 bytes) for every server it talks to.

In truly limited clients, even this small memory overhead may be excessive. One solution to this problem is to replace RSA with an identity-based encryption (IBE) scheme, such as that proposed by Boneh and Franklin [3]. When using IBE, a client can derive a server's public key from its identity and the current time (e.g., from the string "urlwww.example.com:443") as needed. Therefore, there is no need to cache the server's public key. Naturally, this would require defining IBE-based cipher suites for TLS, but doing so would be straightforward.

3.5 Implementation

We have modified OpenSSL 0.9.6a to negotiate and perform fast-track handshakes. Since OpenSSL does not currently support TLS extensions, our implementation instead used TLS' version negotiation scheme: fast-track-capable clients and servers speak the fictitious TLS "Version 1.1."

We summarize our observed savings in bandwidth below. Aside from the bytes-sent measurements, our implementation also maintains the savings in flows that fast-track provides over ordinary TLS handshakes: three flows, rather than four.

Table 1 presents, for each of two cipher suites, the number of bytes written across the wire by the client and by the server in both a standard (RFC 2246) TLS handshake [5], and a fast-track handshake. The first cipher suite, called "`TLS_RSA_WITH_3DES_EDE_CBC_SHA`" in RFC 2246 (and called "`DES-CBC3-SHA`" in OpenSSL), uses RSA for key exchange. It does not require a ServerKeyExchange message to be sent. The second cipher suite, "`TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA`" (and "`EDH-RSA-DES-CBC3-SHA`" in OpenSSL), employs Ephemeral Diffie-Hellman (EDH) for key exchange, with RSA authentication. A handshake using this cipher suite requires the server to send a ServerKeyExchange message. At present, EDH-based key exchange is not widely deployed in TLS environments, though support for it has been added in some recent browsers; accordingly, the first of the two settings in Table 1 is by far the more common.

	RFC 2246	Fast-Track	Savings
TLS_RSA_WITH_3DES_EDE_CBC_SHA			
Client	322	291	10%
Server	1187	130	89%
Total	1509	421	72%
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA			
Client	285	245	14%
Server	1461	404	72%
Total	1746	649	63%

Table 1: Handshake bytes sent for TLS key exchange methods; no client authentication

	RFC 2246	Fast-Track	Savings
TLS_RSA_WITH_3DES_EDE_CBC_SHA, client auth			
Client	2519	2488	1%
Server	1196	130	89%
Total	3715	2618	30%
TLS_DHE_RSA_WITH_3DES_SHA, client auth			
Client	2482	2442	2%
Server	1472	404	73%
Total	3954	2846	28%

Table 2: Handshake bytes sent for TLS key exchange methods; client authentication required

The data in Table 1 show quite clearly that, in typical situations, the bandwidth cost of a TLS handshake is dominated by the server certificate chain. The server's key exchange message, when sent, is also a significant component. Note that the server here sends only its own certificate. Since the client must already have a copy of the self-signed CA certificate to assess the server's credentials, the CA certificate need not be transmitted. (This is permitted by the TLS specification [5, 7.4.2].)

Although the savings in bandwidth generated by the server is substantial, the savings in client bandwidth is quite modest. In fact, our implemented client does not (yet) send the determining-parameters hash to the server. These additional 22 bytes of extension to the client hello (required in a fully-conforming fast-track implementation) would largely negate the savings in client bytes-sent evident in Tables 1 and 2. The savings in server bytes-sent is unaffected. This underscores that, since fast-track does not assume a server-side cache, it can do little to reduce the information that a client must supply during a handshake. (The client bytes-sent savings are largely at the TLS transport layer, where the reduced number of flows allows greater consolidation of messages.)

Table 2 presents data in the same format as in Table 1, but in which the server requires that the client authenticate itself. Here, the dominant component is the client's certificate chain. Unlike the server, the client does send the CA certificate along with its own.

The limited gains in bytes-sent seen in Table 2 again reflect fast-track's inability to do away with the sending of client information to the server. The specific problem of client certificates can be alleviated via a different mechanism, complementary to fast-track: the TLS Extensions document defines a client-certificate-url extension [1, 3.3]. With this extension, a client sends the URL where its certificate may be found, along with a hash of the certificate, rather than the certificate itself.

The number of bytes which a server writes depends on its certificate chain; similarly for a client when client authentication is required. Since certificates vary in length, a limit is placed on the accuracy of bytes-sent measurements. This limit is made more severe by the presence at several points in the TLS handshake of arbitrary-length lists: the client's supported cipher suites; the client's supported compression methods; and the server's acceptable certificate types and acceptable CA names (for client authentication).

3.6 Security analysis

In this section we argue that fast-track is no less secure than the ordinary TLS handshake protocol. Unfortunately, a formal argument about the security of fast-track as a handshake protocol is extremely difficult, especially in the absence of a comprehensive formal analysis of TLS [7]. Nor is a rigorous reduction of fast-track security to TLS security feasible – the message order is changed between the two protocols, so an attacker on one would not necessarily be able to create messages for the other without breaking the hash functions used in the finished-message digests. In light of these limitations, we present common arguments about the security of fast-track.

Fast-track is negotiated in the course of an ordinary TLS handshake, using the fasttrack-capable extension (Section 3.3.1). The extension itself contains no sensitive data, and the negotiation is protected by the same mechanisms that protect other negotiated extensions.

A client should store determining parameters for use in a future fast-track handshake only after verifying that the server has a valid certificate, and the parameters come from an ordinary handshake, so these parameters should not be open to tampering. Furthermore, if the client and server determining parameters differ, the mismatch will be detected in the course of the handshake, since some messages will be incomprehensible. Thus, determining parameter mismatch is not a security problem, and the SHA-1 hash should be sufficient to provide collision-resistance for robustness. (The exception is if the client has obtained an adversary's certificate for the server's distinguished name, a situation that could allow for a man-in-the-middle attack. But this would require a compromise of the public key infrastructure.)

All the same information exchanged in a standard handshake is exchanged in a fast-track handshake, except for the determining parameters, for which a cryptographic hash is exchanged. The handshake digest hashes in the Finished messages should thus provide the same security as in ordinary TLS.

The ordering of the server and client Finished messages is opposite of that in ordinary TLS handshakes, but TLS session resumes also use this reversed ordering.

The server response message (ServerHello or ServerHelloFT) is included in the final hashes regardless of whether fast-track is denied, so rollback attacks should be impossible.

The only message not verified by both the client and server finished-message hashes is the client CertificateVerify message. It is included in the

client finished-message hash, so the server should be able to detect its having been modified and abort the connection.

In any case, the client certificate itself is included in both finished-message hashes, and is presumably no more open to tampering than in an ordinary TLS handshake. The client CertificateVerify message contains only a signature with the certificate's key, so opportunities for mischief through its modification are limited.

4 Client side session caching

In most cases, TLS session resumption dramatically improves handshake performance, since it allows the peers to skip the time-consuming key agreement computation. However, maintaining the session cache imposes a substantial memory burden on the server. In addition, when multiple SSL servers are used together for load balancing, session-cache coordination between the servers becomes problematic. Client side session caching (CSSC) substantially alleviates these problems.

4.1 Memory consumption

The amount of memory consumed by the session cache scales roughly with the number of sessions cached on the server. The exact amount of memory consumed by each session varies, but is at minimum 48 bytes for the master secret. Since session IDs are themselves 32 bytes, 100 bytes is a reasonable approximation. Assuming a server operating at 1000 handshakes/second, which is easily achievable with modern hardware, the session cache will grow at a rate of 3 MB/minute.

This effect makes a 24 hour timeout, as suggested by the TLS RFC, quite impractical for any server with reasonable transaction volume. Typical settings are on the order of a small number of minutes. For example, mod_ssl's default setting is 5 minutes. Experiments with sites like Etrade show that sessions are evicted from the session cache after approximately 5 minutes.

4.2 Distributed implementations

When a web site is served by a cluster of SSL servers behind a load balancer, the problem of sharing the session cache becomes a distributed systems problem. In general, server implementors choose to ignore this problem. Instead, each server has its own session cache and the load balancer is expected to

direct returning clients to their original server. This increases load balancer complexity.

Moreover, the need to maintain connection locality to make use of the session cache can interfere with the load balancer's ability to distribute load evenly.

4.3 Client side session caching theory

One way to reduce the overhead of session caching is to force the client to store the session cache data for the server and provide the server with it when attempting to resume. For obvious reasons, it's not safe to provide the client with the cache data in the clear, but it's easy to encrypt it using a symmetric cipher under a fixed server key called *enc-key*. The simplest such token is:

$$Token = E_{enc-key}[Cache\ Data]$$

Note that the symmetric cipher must be semantically secure (for example, by using CBC mode with a new random IV for every token) since otherwise an attacker might deduce relations between the cache data in different tokens.

When using the simple token above the server may not be able to tell whether the token has been modified. The problem is that encryption by itself does not guarantee integrity. To verify integrity the server should also use a MAC with a fixed server key called *mac-key*. There are several ways for combining encryption and a MAC [6]. For consistency with TLS we construct tokens using the following method called mac-then-encrypt:

$$\begin{aligned} Token &= E_{enc-key}[Cache\ Data \parallel mac] \\ &\quad \text{where} \quad mac = MAC_{mac-key}[Cache\ Data] \end{aligned}$$

Both *enc-key* and *mac-key* can be derived from a single fixed server master key as done in TLS. This approach allows us to do away completely with the server-side session cache. Any data that the server wishes to retain across sessions can be placed in the authentication token. Since the token is authenticated and encrypted, even sensitive information can be carried around by the client. Only the master key need to be shared across server processors or cluster members, and these can be statically configured.

4.4 Adapting TLS for client side caching

For CSSC to work, the server must ensure that the client returns the authentication token when requesting resumption. The only piece of information

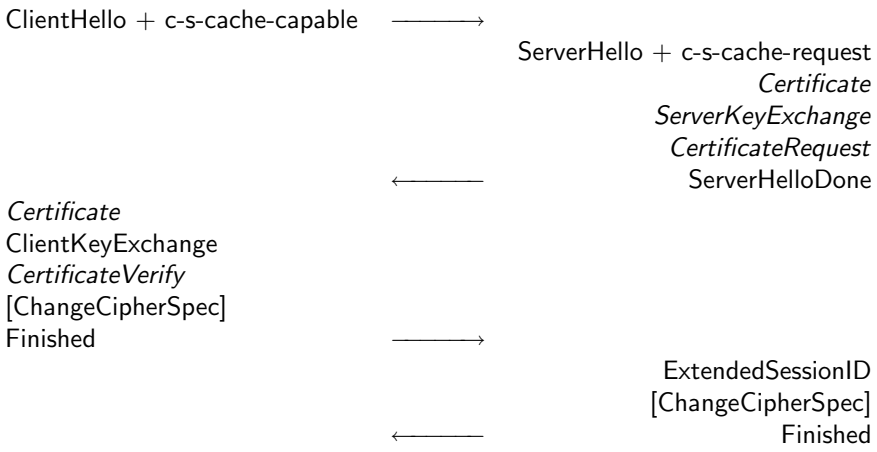

Figure 6: TLS handshake using extended session ID

that the client is guaranteed to provide during a resumption attempt is the session ID, so this suggests that the authentication token must be placed in the session ID. Unfortunately, the session ID is currently unsuitable for two reasons. First, it's too short. The TLS session ID is limited to 32 bytes. Since the Master Secret is 48 bytes long, an encrypted Master Secret cannot fit in the session ID. Second, it's delivered at the wrong time. The server provides the session ID in the ServerHello, which is transmitted before it knows the Master Secret.

We must therefore modify the TLS handshake message flow to make client-side caching possible. The necessary changes are relatively simple. First, we relocate the session ID. The server delivers a zero-length session ID in the ServerHello message, and then sends a new ExtendedSessionID message containing the authentication token immediately before the ChangeCipherSpec message. The sequence of events is shown in Figure 6.

Like fast-track, CSSC is negotiated using a TLS Extension. The client signals to the server that it can do client-side session caching using the client-side-cache-capable extension in the ClientHello. If the server wants to request CSSC, it responds with the client-side-cache-request extension in its ServerHello.

When the client attempts to resume a client-side cached session, it needs to provide the server with the authentication token. If we could guarantee that this token would be less than 256 bytes long, it could be placed in the session ID field of the ClientHello. However, if the authentication token includes a client certificate, it will not fit in the ClientHello. Instead, we use

another TLS extension to carry the extended session ID. Session resumption with CSSC is otherwise identical to ordinary session resumption.

4.5 Cache Invalidation

TLS sessions can become invalid in a number of ways, including expiration, improper closure, and error. Expiration is easily dealt with by including a timestamp in the token, provided that the servers in a load-balancing configuration are roughly synchronized. Moreover, when CSSC is in use, expiration is less important, since it's not required for garbage collection.

Unscheduled invalidation due to errors or improper closure is a more serious problem. In the stateless regime described in Section 4.4, tokens are completely self-authenticating and there is no way to invalidate sessions before they expire. One can proceed along two paths. The first path is simply to fail to invalidate sessions under such circumstances. This violates the TLS specification, but, as we argue in Section 4.8, is still secure. The second is to maintain a "black list" consisting of those sessions that have been invalidated but have not yet expired.

The size of the black list depends on the exact data structure chosen. In general, a black list consumes far less memory than a session cache. Suppose the server is processing R new handshakes per second where each session has a lifetime of T seconds. Suppose the invalidation rate is $E \in [0, 1]$, i.e. we have ER invalidations per second. Then an ordinary session cache has size $80TR(1-E)$ bytes assuming 80 bytes per entry (32 bytes for the session ID and 48 bytes for the premaster secret). A naïve hash table implementation that just keeps track of invalidated session IDs has size $32TRE$ bytes, which will be less than the size of the session cache unless $E > 0.7$.

In practice, the size of the black list can be substantially reduced by clever implementation. In particular one can store a message digest of the session ID rather than the session ID itself, thus reducing the size of the black list to $8TRE$ bytes when using a 64-bit digest. (Note that this trick can be used to reduce the size of the session cache as well, but since the bulk of the session cache data is the Master Secret the effect is less.)

It's worth mentioning two more-sophisticated data structures. In environments where E is relatively high, it is more efficient to assign tokens in sequence. The black list can then be stored as a low-water mark and a bitmask (the bitmask is 0 at positions that correspond to invalid sessions and 1 elsewhere). The size of the bitmask vector is $(1/8)TR$ bytes. When the bitmask becomes excessively large, the low-water mark can simply be rolled forward, expiring all sessions created before that time. In environ-

Invalid Rate	Session Cache	Hash Table	Bitmask	Wide Bitmask	Bloom Filter
.01	171,072,000	172,800	270,000	8,640,000	25,879
0.1	155,520,000	1,728,000	270,000	8,640,000	258,793
0.2	138,240,000	3,456,000	270,000	8,640,000	517,587
0.5	86,400,000	8,640,000	270,000	8,640,000	1,293,969
1	0	17,280,000	270,000	8,640,000	2,587,938

Table 3: Memory consumption, in bytes, for various session cache data structures. Here, TR, the number of sessions that must be kept track of, is 2,160,000. Column (1) refers to the standard valid session cache. Column (2) refers to a simple hash table storing a 64-bit hashes of invalid session IDs. Column (3) refers to the bitmask method and column (4) refers to a bitmask stored as one 32-bit word per bit. Column (5) refers to storing the invalid sessions using a Bloom filter tuned for a 1% false positive rate.

ments where E is low, a more efficient approach is to use a Bloom filter [2]. Although a Bloom filter has false positives and thus identifies as invalid some valid sessions, this merely causes the occasional full handshake. The size of the Bloom filter can be tuned for the appropriate balance between memory consumption and false positive frequency.

Table 3 shows the memory consumption, in bytes, for various session-cache data structures. As this table shows, CSSC consumes less memory than ordinary session caching unless the invalidation rate is very high.

Aside from their small size, both Bloom filters and bitmasks have the advantage that they require only minimal synchronization and can be stored in shared memory. If we assume that bit-wise ORs can be performed atomically, both writers and readers can safely write the blacklist without first synchronizing. (In the worst case we can allocate one word per bit). The only operation that requires synchronization is rollover, in order to ensure that two writers don't independently try to move the high water mark.

4.6 Performance Analysis

When a session is resumed, the CSSC token must be decrypted and MAC verified. (Tokens must also be encrypted when issued but we assume that session resumption is more common than full handshakes.) This requires symmetric decryption and the computation of a MAC, which, though not free, are relatively efficient. Table 4 shows the time (in seconds) to perform

a million token encryptions using HMAC-MD5 and various encryption algorithms. A size of 100 bytes is realistic for sessions which do not require client authentication, 500 bytes is realistic for small certificates and 1000 bytes for large certificates. Note that the high estimate is somewhat pessimistic, since the certificate, if MACed, can be left unencrypted.

Since CSSC does not require session cache locking, it is natural to compare the additional cost incurred for encryption and MACing to that for locking the session cache. Table 5 shows the performance of locking and unlocking on a typical server, an Ultra60 running Solaris 8. The benchmark task, from Stevens [10], is incrementing a single counter from one to one million. The column labeled Semaphores uses System V semaphores and the column labeled Mutexes is POSIX Mutexes. Note that performance declines dramatically as the number of processes attempting to acquire the lock increases. To some extent, this is not representative of SSL servers since the server processes, unlike our benchmark, do other things than attempt to acquire locks. However, on an active server, it is reasonable to expect that there will be significant contention for the session cache.

As is apparent, the cost for CSSC is commensurate with that incurred by locking the shared session cache. When client authentication is not required, CSSC overhead is comparable to that of an average of 10 processes' accessing the session cache (using mutexes) or merely 2 processes' (using semaphores). Even with client authentication, we expect the CPU cost from CSSC to be better than for semaphores though somewhat worse than that for mutexes. (How much worse depends on how much contention there is for the session cache.) Note that CSSC has the advantage here in that its CPU overhead per connection is constant rather than increasing with load on the server as does cache locking.

Bytes	**AES-128**	**DES**	**3DES**
100	8	9	18
200	13	16	33
300	18	22	48
500	27	34	78
1000	51	65	214

Table 4: Token decryption and MAC performance: seconds per million

Processes	Semaphores	Mutexes
1	2.2	0.2
2	10.6	0.8
3	25.4	1.8
5	51.9	7.7
10	100.2	10.1
20	192.3	20.5
40	388.5	44.1
60	586.0	79.2

Table 5: Locking performance: seconds per million

4.7 Backward Compatibility

CSSC is designed to be backward compatible with ordinary TLS so that it can be deployed in stages. If both peers are CSSC-capable, they will use CSSC and omit ordinary session caching. Otherwise they will fall back to ordinary TLS with ordinary session resumption. This means that CSSC-capable agents need simultaneously to do old-style and CSSC-style session caching.

This does not introduce any security problems. Since the CSSC token and session ID are delivered in separate fields, they can't be confused; and, since they are different lengths, they will not collide. (The probability of random 32-byte fields colliding is negligible in any case.) Note that regular TLS session IDs are separately generated from the CSSC tokens. Hence, no information about open sessions in one method can be gained from open sessions in the other method.

4.8 Security Analysis

CSSC introduces three new security concerns: (1) Can CSSC be safely negotiated? (2) Can attackers tamper with or forge CSSC tokens in any useful way? (3) What is the impact of failing to invalidate sessions?

The use of CSSC is negotiated via the usual TLS extension mechanisms. Thus, any attempt to force peers to use CSSC by tampering with the handshake would be detected by the Finished check. No sensitive information is contained in any of the new CSSC extensions, so having them appear in the clear is not a problem. Similarly the ExtendedSessionID message is covered by the Finished message and therefore cannot be forged.

Since the integrity of CSSC tokens is protected using a key known only to the server, it is not possible for attackers to tamper with valid tokens or forge their own. It is of course possible for an attacker to attempt to use a passively-captured CSSC token to resume his own connection, but since the attacker will not know the corresponding keying material, he will not be able to complete the handshake. Similarly, since the CSSC token is encrypted an attacker cannot learn anything about the keying material hidden in the token. (When encrypted using a semantically secure cipher, the token provides no more information than the random session ID currently used in TLS.) It's worth noting that if a stream cipher is used to encrypt the token, care must be taken to ensure that a fresh section of keystream is used each time, perhaps by providing an offset at the beginning of the token.

As we noted earlier, although the TLS specification requires that a session be invalidated when errors occur, this procedure adds substantial complexity to CSSC. With respect to security, rather than standards compliance, invalidation is unnecessary. To see this, consider that errors can occur under three circumstances: (1) local-side error; (2) error by the peer; or (3) active attack. Only the third case represents a potential threat.

However, the structure of TLS key derivation renders this threat minimal. In particular, because the individual session keys are derived from the master secret via a message-digest-based PRF, it is not possible to obtain useful information about the master secret or about other sessions by observing server behavior with respect to one session. To do so would require reversing the message digests. If such a reversal were possible it would seriously threaten the security of TLS. Thus, failing to invalidate sessions — even under an active attack — does not pose a security risk.

If session invalidation is performed, then one possible concern here is the predictability of CSSC tokens. If the sequence-number-based blacklisting method of Section 4.5 is used, then the plaintext of the tokens (at least the part containing the sequence number) will be predictable. If an attacker could generate tokens with known sequence numbers he could invalidate the corresponding sessions. Even though the tokens are encrypted, it is possible that, given a token for one session, the attacker could generate a token for another session if, for instance, the token were encrypted in CTR mode (by using the malleability of ciphertexts generated by this mode). However, the MAC used to test token integrity prevents a user from generating a valid token for another user's session. Therefore, servers must test token integrity before using any data in the token. Otherwise, an attacker might be able to take advantage of bad logic by generating an invalid token with a predictable sequence number, which would then cause session invalidation

when the handshake failed. Note that it is still possible to invalidate sessions that the attacker has observed, but this attack is also possible with ordinary TLS.

5 Comparison of fast-track and CSSC

TLS servers deal many clients, whereas clients typically connect to only a few servers in a given time period. Nevertheless, standard TLS requires more extensive state maintenance on the server. This is understandable, since servers cannot trust clients' honesty, but not optimal. Information that clients could remember must be retransmitted to them with each connection; state which thus must remain on the server is expired for lack of space.

Both fast-track and CSSC make use of client-side caching. Both use cryptography to ensure that malicious clients cannot subvert the handshake protocol and reduce the security of TLS.

Although both fast-track and CSSC make use of client-side caches to improve TLS efficiency, the environments in which their use is appropriate are quite different. Fast-track is intended to save bandwidth and round-trip-induced latency over a full handshake, and is therefore most useful where the connection between the client and server is slow. Also, fasttrack requires no state on the server. By contrast, since CSSC increases the size of the TLS handshake messages, it is most appropriate in situations where the connection is fast, but a large number of clients are connecting to a given server.

That fast-track and CSSC are applicable in such widely different scenarios is evidence that client-side caching is a versatile tool for reducing or rebalancing costs (such as bandwidth and memory overhead) associated with the TLS handshake protocol, and likely in other security protocols as well.

6 Conclusions

We have described two extensions to TLS that use client-side caches to improve efficiency. The fast-track extension caches the server's long-lived parameters, reducing network bandwidth consumption in a handshake by up to 72% and the number of flows from four to three. The CSSC extension relocates the session cache from the client to the server, allowing the server to maintain a much larger cache, so that connections that would otherwise have required a full handshake can be resumed.

We have implemented our proposals in a backwards-compatible fashion. If either the client or the server does not understand or does not wish to use these extensions, it can revert to the standard TLS handshake. Our prototype implementations are available for download. An Internet-Draft describing fast-track is also available [8].

Acknowledgments

We thank Dan Simon and Dan Wallach for helpful conversations about this paper. This work was partially supported by an NSF CAREER grant.

References

[1] S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen, and T. Wright. TLS Extensions. Internet-Draft: `draft-ietf-tls-extensions-05.txt`, July 2002. Work in progress.

[2] B. Bloom. Space/time Trade-offs in Hash Coding with Allowable Errors. Comm. ACM, 13(7):422–6, 1970.

[3] D. Boneh and M. Franklin. Identity-Based Encryption from the Weil Pairing. In Joe Killian, editor, Proceedings of Crypto 2001, volume 2139 of LNCS, pages 213–29. Springer Verlag, 2001.

[4] C. Coarfa, P. Druschel, and D. Wallach. Performance Analysis of TLS Web Servers. In Mahesh Tripunitara, editor, Proceedings of NDSS '02, pages 183–94. Internet Society, feb 2002.

[5] T. Dierks and C. Allen. RFC 2246: The TLS Protocol, Version 1, January 1999.

[6] H. Krawczyk. The order of encryption and authentication for protecting communications. In Joe Killian, editor, Proceedings Crypto 2001, volume 2139 of LNCS, pages 310–31. Springer Verlag, 2001.

[7] J. Mitchell, V. Shmatikov, and U. Stern. Finite-State Analysis of SSL 3.0. In Proceedings of USENIX Security 1998, pages 201–16, 1998.

[8] H. Shacham and D. Boneh. TLS Fast-Track Session Establishment. Internet Draft: `draft-shacham-tls-fasttrack-00.txt`, August 2001. Work in progress.

[9] H. Shacham and D. Boneh. Fast-Track Session Establishment for TLS. In Proceedings of Internet Society's 2002 Symposium on Network and Distributed System Security (NDSS), pages 195–202, 2002.

[10] W. Richard Stevens. UNIX Network Programming, Volume 2: Interprocess Communications. Prentice-Hall, 1999.

[11] Wireless Application Forum. Wireless Transport Layer Security Specification. `http://www.wapforum.org`, 2000.