An Analysis of TLS Handshake Proxying (full version)

Nick Sullivan
CloudFlare Inc.
Email: nick@cloudflare.com

Douglas Stebila
Queensland University of Technology
Email: stebila@qut.edu.au

***Abstract*—Content delivery networks (CDNs) are an essential component of modern website infrastructures: edge servers located closer to users cache content, increasing robustness and capacity while decreasing latency. However, this situation becomes complicated for HTTPS content that is to be delivered using the Transport Layer Security (TLS) protocol: the edge server must be able to carry out TLS handshakes for the cached domain. Most commercial CDNs require that the domain owner give their certificate's private key to the CDN's edge server or abandon caching of HTTPS content entirely. We examine the security and performance of a recently commercialized delegation technique in which the domain owner retains possession of their private key and splits the TLS state machine geographically with the edge server using a private key proxy service. This allows the domain owner to limit the amount of trust given to the edge server while maintaining the benefits of CDN caching. On the performance front, we find that latency is slightly worse compared to the insecure approach, but still significantly better than the domain owner serving the content directly. On the security front, we enumerate the security goals for TLS handshake proxying and identify a subtle difference between the security of RSA key transport and signed-Diffie–Hellman in TLS handshake proxying; we also discuss timing side channel resistance of the key server and the effect of TLS session resumption.**

Keywords*—*cryptographic protocols; network topology; content distribution networks; secure outsourcing; TLS; proxy

I. Introduction

Large-scale websites demand significant infrastructure and services: they need to be able to handle a large volume of requests and they need to be secure. To handle large volumes, websites often distribute requests across many servers. To provide security, a key ingredient is the transmission of data using the Transport Layer Security (TLS) protocol [1], using an authenticated encryption tunnel between a server and a client.

With the high interactivity involved in Web 2.0 services, web sites also need to be fast and responsive. One of the major barriers to improving latency is physics itself. A bit of data cannot reach its destination faster than the speed of light, and the circumference of the globe means that the minimum time required to send information around the world is 80ms, a nontrivial amount of time in both computer terms and human terms, as human beings only perceive events that take less than 100ms as instantaneous [2]. To reduce latency, servers can be geographically distributed so that there is a server located physically close to the user.

A. Content delivery networks (CDNs)

A common approach to simultaneously handling a large volume of requests and reducing latency is the use of a *content delivery network (CDN)*. A CDN service typically runs a network of machines that are widely distributed geographically. These servers are often called *edge servers* since they are placed near the "edge" of the Internet, namely close (in terms of network topology) to retail commercial Internet users.

Website functionality is typically shared between the CDN edge server and the origin web server(s): CDNs are used to speed up delivery of static assets such as images and videos, while dynamic pages (login, account pages, etc.) are served from the origin server. The dynamic content may be served directly from the origin web server, or may be routed through a reverse proxy server run by the CDN. Either way, on average a request will travel a shorter distance through fewer networks, reducing the minimum transport latency needed.

Many modern CDNs use this idea of a reverse proxy to provide a more comprehensive service for websites. By using a third party reverse proxy service, a website can be run from a single location but gain all the advantages of a global reverse proxy with very little configuration. It can be enabled simply by setting the DNS records of the site to point to one of the reverse proxy service's IPs. The reverse proxy service can then use IP anycast routing [3] to allow each of the geographically dispersed edge servers to use the same IP address, providing global load balancing. Visitors to the site will be served data from the edge server obtained from the upstream origin site or a local cache. This architecture is flexible and provides additional benefits above and beyond caching such as DDoS protection and web application firewall services.

While the use of CDNs allows for handle a large volume of request and reducing latency, it has an impact security. Encryption on the web is typically achieved using a server-authenticated TLS channel between the web browser and web server. Server authentication is provided using a public key infrastructure (PKI): the server obtains a certificate linking its public key to its domain name. When the TLS channel is established, the client verifies that the server's certificate matches the domain name of the website.

Having a reverse proxy in the middle of this transaction requires rethinking the end-to-end TLS model. TLS requires that server which the serves content to the browser—in the case of a CDN, this means the edge server—must be able to complete TLS handshakes, which requires the private key from the certificate for this domain name. Previous generation

CDNs got around this by hosting static assets on a different domain served over unencrypted HTTP. However, modern browsers restrict display of unencrypted content on pages served over a secure channel: thus, if an HTML page is served over HTTPS, then all of its sub-resources—images, JavaScript, and stylesheets—must also be served over HTTPS.

An alternative approach employed by most reverse proxy CDNs is to have the private key for every cached domain present at the edge server. This allows the edge server to complete TLS handshakes on behalf of the domains it is caching, so it can serve all required content over HTTPS. Understandably, many website operators are reluctant to share their private key with a third party or store private keys in less secure edge locations, as this significantly increases the risk of key compromise. To ensure low latency to the maximum number of users, edge servers will often be physically located in a variety of different countries and political jurisdictions, which may be subject to distinct regulatory regimes, governmental controls, and levels of physical security. Protecting private keys on machines where the attacker has physical access has proven to be a near impossible task, even using dedicated hardware [4]; the lack of control of data on computers at some edge locations means that the confidentiality of these keys is at risk. Renewing TLS certificates can be expensive, and revocation of compromised certificates is not a solved problem. Consequently, placing private keys at risk in all locations of a CDN is inadvisable. Issuing edge servers with short-lived certificates for each hosted domain is also undesirable in the current web public key infrastructure due to the need for the domain owner to frequently obtain and deploy new certificates.

B. TLS handshake proxying

Very recently, commercial CDNs have explored various architectures for TLS proxying. Akamai filed a patent application in 2013 [5] for proxying TLS handshakes involving RSA key transport. In 2014, CloudFlare announced a product called *Keyless SSL* [6], [7] which provides proxying of TLS handshakes for both RSA key transport and signed-Diffie–Hellman ciphersuites.

The main idea of both of these approach is to split the TLS handshake so that the edge server does not have to store the keys at all. Part of the TLS handshake between the client and the edge server is *proxied* over a secure link to a third server, a *key server*, which maintains possession of the origin server's private key. For compatibility with clients, this split does not require any changes to the TLS protocol.

More specifically, in the TLS handshake there is a single operation which must be performed using the private key corresponding to the origin server's public key certificate. For TLS ciphersuites that use RSA encryption, the RSA private key must be used to decrypt the pre-master secret in the `ClientKeyExchange` message. For TLS ciphersuites that use digital signatures, the private key must be used to sign the server's ephemeral Diffie–Hellman public key in the `ServerKeyExchange` message. In this construction, the edge server relays these operations over a dedicated TLS channel to an off-site key server which performs the operations and returns the result, allowing the edge server to complete the TLS handshake for the origin server's domain name. The origin server's private key never leaves the key server. In practice, the key server could be embodied in several ways: (i) the key owner (the party behind the origin web site) could run its own key server, maintaining possession of the private key and ensuring it only uses the key on requests that come authenticated by edge servers in the CDN; (ii) the key owner could give the private key to the CDN, who runs a key server in a more controlled high-security environment; or (iii) a third party could run the key server, for example for compliance or liability reasons. In any of these cases, the private key could be kept in a *hardware security module (HSM)* if desired.

Figure 1 compares the round trips required without and with a proxy. Requesting a resource over HTTPS requires four round trips: one to establish the TCP socket, two for the TLS handshake, and one to request and receive the HTTPS resource. Without proxying (Fig. 1a), all four flows must take place between the client and original web server. With proxying (Fig. 1c), if the edge server has cached the content, the client communicates only with a nearby edge server, which then proxies just one TLS handshake message to the key server.

C. Contributions

This paper studies the performance and security aspects of TLS handshake proxying. Since these systems are already being used by a commercial CDN, it is essential to understand how the security goals change in the context of TLS handshake proxying, and assess whether the designs meet the security goals. It is also important to check that the latency improvements of TLS handshake proxying are indeed realized.

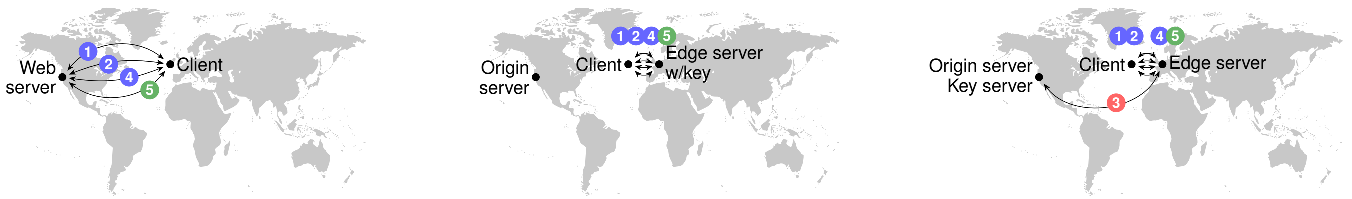
Our work focuses specifically on the TLS handshake proxying system implemented by CloudFlare in their Keyless SSL product [6], [7]; while Akamai has a patent application in this area, they have no deployed system as of the time of this writing.

Performance. Our experiments show that actual latency matches expected behaviour. For a client in Dublin and origin web server in San Francisco (Fig. 1a), the time for a direct handshake was 497ms. With the private key at an edge server in London (Fig. 1b), the handshake time is just 64ms. Using TLS proxying, where the key is kept safe in the key server in San Francisco but the rest of the handshake is handled by the edge server (Fig. 1c), the total handshake time is 395ms, a substantial reduction compared to a direct handshake.

Security. We enumerate the security goals of TLS handshake proxying: key-server-to-client authentication, edge-server-to-client authentication, and channel security. Signed-Diffie–Hellman meets all of these, but RSA key transport does not meet uniqueness of sessions in key-server-to-client authentication. We discuss security concerns for the key server (including the importance of resistance to timing side channel attacks in RSA key transport), and the effect of TLS session resumption using both session IDs and session tickets.

D. Related work

Cryptographic verification of delegated data. Lesniewski-Laas and Kaashoek [8] initiated a line of work aiming to provide authenticity of data delivered by CDNs and other proxies. In their SSL splitting technique, the proxy relays the entire



(a) Without proxying: 4 trans-Atlantic round trips

(b) With key and static content cached at edge server in London: 4 local round trips

(c) With TLS proxying to key server in San Francisco and static content cached at edge server in London: 4 local round trips and 1 trans-Atlantic round trip

1 TCP handshake 2 TLS handshake initial flow 3 Proxied private key operation 4 TLS handshake completion 5 TLS-encrypted application data

Fig. 1: Comparison of flows to establish a TLS channel and receive an HTTPS resource between a client in Dublin and a web server in San Francisco. (Sub-figures (b) and (c) show the client separated a bit to make the flows visible in the diagram.)

TLS handshake to the origin server, who completes the TLS handshake, then provides the proxy with the encryption session keys and with authentication tags for the requested resources; this allows the proxy to deliver the requested resources over an encrypted channel while authenticity of those resources still is shown demonstrable only by the origin server. This technique only works with MAC-then-encode-then-encrypt ciphersuites in TLS, not modern authenticated-encryption-with-associated-data (AEAD) in TLS 1.2. Several subsequent works, such as [9], [10], [11], describe alternatives to HTTPS/TLS that provide authenticity of cached resources. These are distinguished from proxied TLS handshakes, which does not aim to provide integrity or authenticity assurances of the content itself, only of the authority to deliver content on behalf of the original party. While this is admittedly a weaker assurance to the client, it is something that can be implemented on the web today with no client-side changes: cryptographic verification of delegated data would require changes in the TLS protocol and browser implementations. Other work does address verifying the delegation of TLS handshakes [12], this again requires protocol changes, and only works on TLS ciphersuites involving signatures.

Security of CDNs. Liang et al. [13] survey the practices of 20 major CDNs with respect to HTTPS certificates. The consider two scenarios for CDNs hosting HTTPS content: *shared certificates*, where the CDN obtains a certificate of its own for domains delegated to it by its clients, and *custom certificates*, which "requires web site owners to upload their certificates and private keys to CDN providers". This marks a major distinction with proxied TLS handshakes, where web site owners can retain their private keys at a key server. Liang et al. also discuss techniques for indicating SSL delegation from the web site owner to the CDN using certificate name constraints and DNS-based solutions. Delignat-Lavaud and Bhargavan [14] examine a variety of security issues in HTTPS proxying for commercial CDNs.

II. TLS AND HANDSHAKE PROXYING

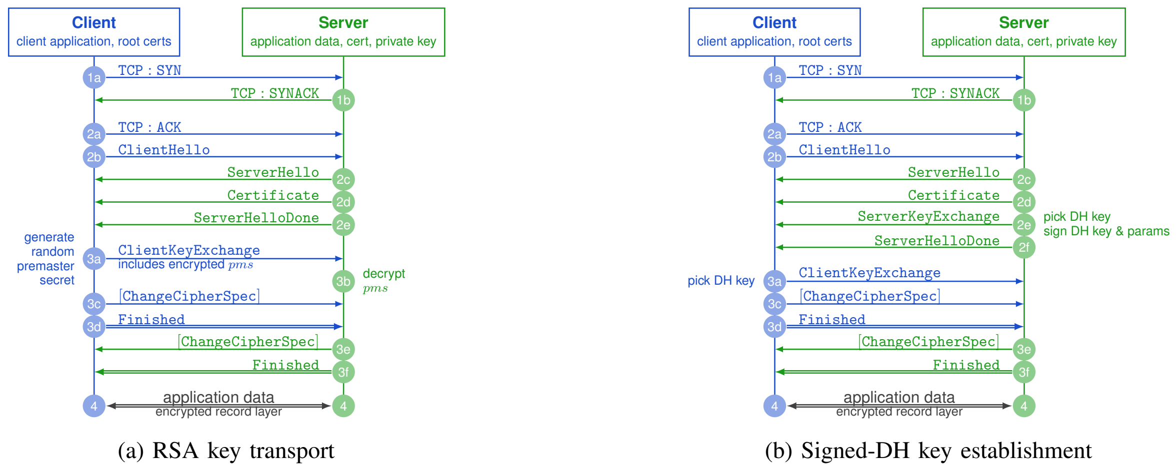
The Transport Layer Security (TLS) protocol [1] is the successor of the Secure Sockets Layer (SSL) protocol, and is used to provide confidentiality, integrity, and authentication for a variety of Internet protocols, most prominently the Hypertext Transport Protocol (HTTP) [15] in the form of HTTPS (HTTP over SSL) [16]. As most widely used on the web, HTTPS provides server-to-client authentication using X.509 certificates issued by a commercial certificate authority binding a particular RSA public key to a particular domain name.

The TLS protocol consists primarily of two sub-protocols: the *handshake protocol* and the *record layer*. In the handshake protocol, the client and server negotiate which combination of cryptographic algorithms to use (called a ciphersuite), as well as other parameters, then perform server-to-client authentication and establish a shared secret session key. The two most common authentication and key establishment mechanisms are *RSA key transport*, where the client picks a random session key and encrypts it under the server's RSA public key, and *signed Diffie–Hellman*, where the client and server perform an ephemeral Diffie–Hellman (DH) key exchange and the server signs its ephemeral DH public key to demonstrate authenticity; these two methods are described in more detail in Section II-A. The session key established by the handshake protocol is then used in the record layer to encrypt and authenticate application data using the cipher negotiated during the handshake protocol.

A. Traditional TLS handshake

In TLS handshakes involving RSA key transport, as shown in Fig. 2a, the server sends its RSA public key to the client in the `Certificate` message. The client generates a premaster secret (a uniformly random string of fixed length), then encrypts the premaster secret under the server's RSA public key and sends it to the server in the `ClientKeyExchange` message. The server decrypts using its private key and the decrypted premaster secret is used to establish the master secret and session key. Only the true server holding the private key can decrypt the message successfully, so the client obtains an assurance of server-to-client authentication if the server demonstrates it knows the session key in the `Finished` message.

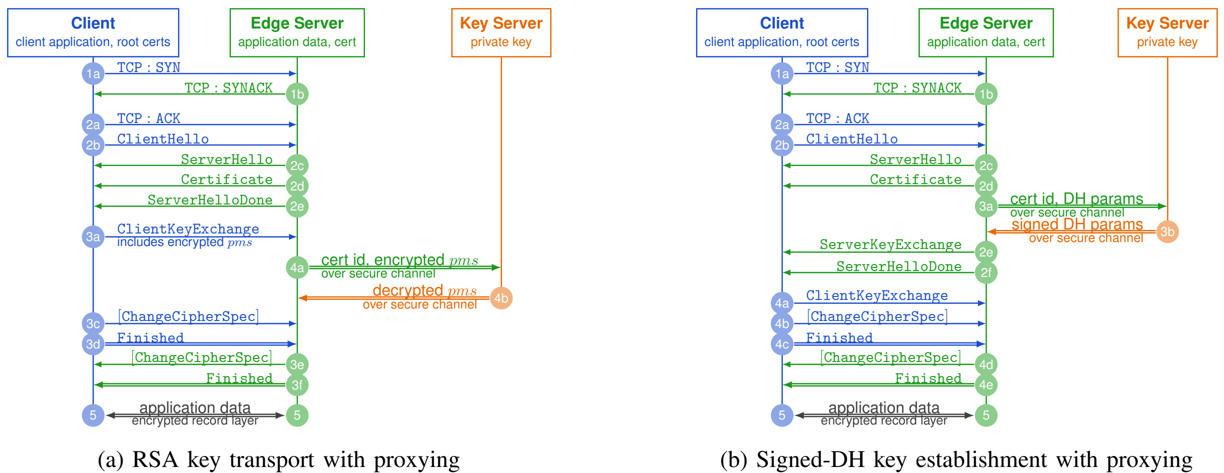
In TLS handshakes involving signed-Diffie–Hellman key establishment, as shown in Fig. 2b, the server sends its public key to the client again in a `Certificate` message; notably this can be any type of public key, including RSA, DSA, or elliptic curve DSA (ECDSA). The server picks a random DH key, and sends the DH public key and parameters, along with a signature of those values, to the client in the `ServerKeyExchange` message. The client verifies the signature, then sends its own



(a) RSA key transport

(b) Signed-DH key establishment

Fig. 2: Traditional TLS handshake (without proxying)



(a) RSA key transport with proxying

(b) Signed-DH key establishment with proxying

Fig. 3: TLS handshake with proxying

DH public key in the `ClientKeyExchange` message; both parties can compute the DH shared secret.

Notably, in both RSA and signed-DH ciphersuites, there is only one step in the handshake that requires the server's long-term private key. For RSA key transport, this is step 3b in Fig. 2a, the decryption of the premaster secret in the `ClientKeyExchange`. For signed-DH, this is step 2e in Fig. 2b, the signing of the server's ephemeral DH public key and parameters for the `ServerKeyExchange` message.

Additionally, there is an abbreviated form of the TLS handshake which allows the client and server to resume a previously established session using fewer roundtrips. There are two forms for session resumption: either the server stores state and the client sends a session ID [1, §7, F.1.4], or the server encrypts the state under a master secret key and offloads that state to the client in the form of a session ticket [17].

B. TLS handshake proxying

The server's long-term private key is a valuable resource that it wishes to protect. In TLS handshake proxying, the private key operation is performed remotely for both RSA key transport and signed-Diffie–Hellman ciphersuites. During the TLS handshake between the client and the edge server of a CDN, the single operation involving private key is proxied to a separate *key server*, which may for example be a separate server in a high-security area of the CDN's network, or may be hosted by the original domain owner. The connection between the edge server and the key server is a pre-established secure connection using TLS with mutual authentication between the edge server and the key server (using credentials from a separate PKI).

For RSA key transport, as shown in Fig. 3a, the edge server relays the encrypted premaster secret from the `ClientKeyExchange` message, as well as an identifier of the certificate used in the connection (in case there is more than one) to the key server, who responds over the pre-established secure connection with the decrypted premaster secret. The edge server then computes the master secret key and continues processing the TLS handshake.

For signed-Diffie–Hellman, as shown in Fig. 3b, the edge server generates an ephemeral DH public key and relays that, along with the DH parameters, a certificate identifier and the client and edge server random nonces, to the key server, who responds over the pre-established secure connection with the

signature. The edge server assembles the ServerKeyExchange message and continues processing the TLS handshake.

This architecture ensures that the private key does not need to be present on the edge server. It does not remove the edge server entirely from the security equation. If an ephemeral key is compromised, an attacker could decrypt the entire transaction, but not any subsequent or previous transactions. If the edge server's credentials for authenticating itself to the key server are compromised, an attacker could get the key server to help it carry out handshakes. If the edge server is itself malicious, it could deliver content other than that intended by the origin server. We discuss these and other security issues in Section IV.

TLS handshake proxying can work with all ciphersuites that use RSA key transport or signed-DH, regardless of the bulk cipher: older ciphers such as RC4 and block ciphers in MAC-then-encode-then-encrypt mode, as well as modern designs such as authenticated encryption with auxiliary data schemes.

C. Implementation consideration: Mapping requests to keys

Logically, the content delivery network is a collection of HTTPS multiplexers as defined by Delignat-Lavaud and Bhargavan [14]. In particular, an HTTPS multiplexer receives TCP connections identified by an IP address and a port number. It has a collection of TLS certificates, and a TLS session cache. It picks a certificate based on the IP address, port number, and optionally the server name indicator (SNI) TLS extension, and attempts to complete the TLS handshake. If successful, it passes the HTTP request off to a virtual host for the request, based either on the IP address and port combination, or based on the Host HTTP header. The system works as follows:

- The content delivery network maintains a table of triples consisting of:
  1) one or more IP address and port combinations;
  2) a certificate valid for one or more domains;
  3) one or more virtual hosts.

  No IP address and port combination will appear in more than one triple.
- Client: To access a web resource on `https://example.com:443/path`, the client sends a DNS request for `example.com`, which resolves to the IP address `1.2.3.4` of the edge server to be used; the DNS resolution may be geographically specific. The client then initiates a TLS handshake with that IP address and port.
- Edge server on `1.2.3.4:443`: Upon receiving a TLS handshake request on a given IP address and port, the edge server looks up the cerificate and virtual hosts associated with the IP address and port. The edge server executes the TLS handshake with the client. When the edge server requires private key operations to be performed for the certificate in question, the edge server transmits to the key server as in Section II-B.
- Key server: The key server verifies that the connection with the edge server is secure (with the edge server authenticating using a client certificate). The key server verifies that the edge server is allowed to request private key operations for the requested host, and, if so, does the operation and returns the result.
- Edge server on `1.2.3.4:443`: Upon completion of the TLS handshake, the edge server checks the `Host` header of the HTTP request. If the domain name in the `Host` header is not part of the triple for this IP address/port and this certificate, the edge server aborts.

III. Performance

In this section, we examine the performance in terms of latency that can be expected from TLS handshake proxying. We consider the case of a traditional reverse proxy network with the key server being located in a more secure location, in particular under the control of the original domain owner and located in the same place as the origin server, as in Fig. 1c.

Let A denote the location of the client, B denote the location of the edge server, and C denote the location of the origin server. As noted above, we assume the key server is located in the same place as the origin server, at C. We denote the *roundtrip time* between locations A and B as AB, and so on.

First consider the traditional TLS handshakeas shown in Fig. 2. Including a standard TCP handshake, there will be 6 flows or 3 roundtrips between the client A and the origin C before application data can be exchanged. (We assume that the server's IP address is known so no DNS lookup is needed.)

Now consider the proxied TLS handshake as shown in Fig. 3. As noted above, we assume the key server is in the same location as the origin server C. We also assume that the tunnel from B to C is pre-established, and that the cost of encryption on this tunnel is negligible (due to modern processors and technology like Intel's AES instructions). Including a standard TCP handshake, there will be 6 flows or 3 roundtrips between the client A and the edge server B and 2 flows or 1 roundtrip between the edge server B and the key server C before application data can be exchanged. (If the application data is cached at the edge server, no additional flows between locations B and C are required to deliver the content.)

Let $\mathrm{HS}_{\mathrm{full}}$ denote the time for the original, unproxied handshake and $\mathrm{HS}_{\mathrm{prox}}$ the time for the proxied handshake. The amount of computation in both is equal so we ignore that factor; as noted above we assume negligible the cost of cryptographic computations for the pre-established tunnel. Then

$$\mathrm{HS}_{\mathrm{full}} = 3\mathrm{AC} \tag{1}$$

$$\mathrm{HS}_{\mathrm{prox}} = 3\mathrm{AB} + 1\mathrm{BC} \tag{2}$$

In practice, edge server locations are chosen to be as close as possible to visitors in order to reduce latency. If we can assume that AB is significantly lower than AC, and that AB + BC is on the same order as AC, then

$$\begin{aligned}\mathrm{HS}_{\mathrm{full}} &= 3\mathrm{AC} \approx 3\mathrm{AB} + 3\mathrm{BC} \\ &= 3\mathrm{AB} + 1\mathrm{BC} + 2\mathrm{BC} = \mathrm{HS}_{\mathrm{prox}} + 2\mathrm{BC}\end{aligned} \tag{3}$$

With these assumptions, a proxied TLS handshake should save nearly as much time as two roundtrips between the edge server and the origin server.

A. Methodology

We tested TLS key proxying using CloudFlare's implementation, which was implemented with the following three parts:

1) Changes to OpenSSL to allow the TLS state machine to use a callback to perform the private key operation.
2) Changes to the nginx web server to establish and maintain connections to the key server and create callbacks that bundle the private key operation request and send it to key server.
3) Creation of a simple key server for the private key operation, implemented in C using OpenSSL.

We then tested the performance of this mechanism. We positioned a client (A) in Dublin, and an edge server (B) in London. The origin server and key server (C) were positioned in San Francisco. We collected the following performance data:

- network roundtrip time between the various locations (average of 32 iterations using `ping -c 32`); and
- handshake times (average over 60 seconds of repeated connections using `cfssl scan`[1]) for: (i) a direct handshake between the client A and the origin server C; (ii) a handshake where the key is held by the edge server B; and (iii) a proxied handshake; these three scenarios correspond to those in Fig. 1.

The different scenarios were set up through CloudFlare's control panel. In the direct handshake handshake scenario, the site is set up with no reverse proxy. In the scenario where the key is held by the edge server, the same certificate that was used on the origin is uploaded to CloudFlare and the reverse proxy is enabled for the site. To test the proxied handshake scenario, Keyless SSL is enabled in the control panel and the key server is configured.

Some additional precautionary steps were made to ensure that the edge server is in the proper state to test the assumptions. First, we used testing tool that has disabled all forms of session resumption (both sesion id based, and session ticket based). Second, performed a test run before the experiment to make sure that the persistent connection between the edge server and the key server was fully established. The existence of this persistent connection was validated on the key server before proceeding with the experiment.

Note that this experiement was performed on a site with extremely low traffic and for which the key server was not under load. For higher traffic sites, the edge server can be configured to point to multiple key servers, via DNS load balancing or through a round-robin selection process.

B. Measured Performance

Fig. 4a shows the observed network roundtrip times. Fig. 4b shows the predicted and actual handshake times for the three scenarios correspond to those in Fig. 1. The predicted times in Fig. 4b are based on the network roundtrip times observed in Fig. 4a and assume zero time for the cryptographic operations, which of course is a simplification, but suffices since the amount of cryptographic computation in all three cases is similar.

[1]https://github.com/cloudflare/cfssl/blob/jacob/scan-pki/scan/tls_handshake.go#L154

Endpoints	Roundtrip time
AB: Dublin (client)–London (edge)	11ms
BC: London (edge)–San Francisco	159ms
AC: Dublin (client)–San Francisco	163ms

(a) Observed network roundtrip times

Operation	Predicted	Actual
(i) Direct handshake (3AC) (Fig. 1a)	489ms	496ms
(ii) Handshake w/key at edge (3AB) (Fig. 1b)	33ms	64ms
(iii) Proxied handshake (3AB + BC) (Fig. 1c)	329ms	395ms

(b) Predicted and observed handshake times; predictions based on network roundtrip times from Fig. 4a and assume zero time for cryptographic operations

Fig. 4: Predicted and observed performance of TLS handshakes without and with proxying

Observe that the predicted performance benefits of a proxied handshake roughly match the actual results. The handshake time of a proxied handshake is more than 2 full BC roundtrips shorter than a direct handshake, as expected from equation (3).

IV. Security

Since the origin server—the owner of the certificate—is not the direct point of contact for connections from clients, the security properties expected by the various parties differ somewhat from the normal TLS setting. The client still wants to be sure that it is communicating with the correct server, but now this may be an edge server that has been authorized to serve data by an origin server. The origin server wants to ensure that edge servers cannot continue to answer requests without the origin server's ongoing authorization.

In this section, we recap the security goals of traditional client-server TLS connections, then discuss the security goals for proxied handshakes and provide a justification that the design meets those goals. Special consideration will be given to the treatment of side channels and session resumption.

A. Security goals of client-server TLS

The security goals for TLS in a client-server setting are:

- *Server-to-client authentication*. If a client *accepts* in a session—meaning that it completes the handshake and believes it is securely talking with a peer—and the long-term private key of the server that the client thinks it is talking to was not compromised, then the server did participate in a unique session with the server, and the transcripts of the client and server match. (Requiring that the server and client having matching transcripts captures the notion that the adversary was effectively passive in this session. Requiring that the corresponding session be unique captures replay attacks.)
- *Channel security*. The adversary cannot read, alter, or insert messages on the authenticated encryption channel between the client and server, provided that the client

and server's session keys were not compromised and that the server's long-term key was not compromised. If the property holds even if server's long-term key was compromised after the client accepted, then the channel is said to have *forward secrecy*.

Formal security definitions typically go into significant technical detail on how to identify matching client and server sessions for the purposes of defining security, but we omit that detail and focus on high level goals. Several recent works have shown that TLS 1.2 ciphersuites using RSA key transport and signed-DH satisfy a reasonable security definition under the assumption that the cryptographic primitives used in the ciphersuite are secure, with the analysis shown using either provable security [18], [19], [20] or a combination of formal verification and provable security [21].

B. Threat model for TLS handshake proxying

In addition to the standard concerns about a network adversary breaking the confidentiality or integrity of communication, the primary threat about which we are concerned is that of a rogue edge server being able to impersonate an origin server after being "deauthorized" by the key server. In this work we are not concerned with an edge server who, while still being trusted by the key server, decides to collect or store user data, nor who decides to delivery alternative (possibly malicious) content than the content intended by the origin server: the edge server may misbehave, but if it is detected misbehaving (for example through routine monitoring), it can be immediately cut off and should no longer be able to act on behalf of the origin server.

C. Security goals of TLS handshake proxying

At a very high level, the security goals of TLS with handshake proxying are like in the normal client-server setting: the client wants to be assured that it is talking to the right server, and that no one can read or manipulate their communications. There are now four parties involved in the connection: the client, the edge server, the key server, and the origin server. The client communicates with the edge server, and the edge server presents the (cached) content of the origin server to the client with the assistance of the key server. However, there is the potential that the edge server may be compromised, so the origin server wants assurance that the edge server cannot impersonate it once the key server has stopped helping the edge server. This leads to three high-level security goals:

- *Key-server-to-client authentication*. If a client accepts in a session, and the long-term private key of the origin server that the client thinks it is talking to was not compromised, then the key server did participate in a (unique) session with the values used by the client.
- *Edge-server-to-client authentication*. If a client accepts in a session, and the long-term private key of the origin server that the client thinks it is talking to was not compromised, and no edge server's private key was compromised between the time when the client sent its first message and received the response, then a legitimately authorized edge server for that key server did participate in a (unique) session with the client.
- *Channel security*. The adversary cannot read or insert messages on the authenticated encryption channel between the client and edge server, provided that the client and edge server's session keys were not compromised, and the long-term private key of the origin server that the client thinks it is talking to was not compromised, and no edge server's private key was compromised between the time when the client sent its first message and accepts.

This channel security goal can be further strengthened to have *forward secrecy*, by requiring that the origin server's private key was uncompromised only before the client accepts, but not necessarily after. We always assume forward secrecy with respect to the long-term keys that the edge servers and key servers use to authenticate each other.

Signed-DH meets all of the required security goals for TLS handshake proxying, under the assumption that the building blocks are secure. RSA key transport meets all goals except for uniqueness of the key server session in key-server-to-client authentication, which is not possible since the key server only receives the `ClientKeyExchange` message containing the encryption of the premaster secret, which an attacker could replay. However, we will still have uniqueness in edge-server-to-client authentication, so an attacker cannot replay the actual content in a session between an edge server and a client. A more detailed justification is given in Appendices B and C.

D. Security of the key server

Because this new architecture for proxied TLS handshakes is designed to put the private key operation on its own server that can be physically separated, we will assume that the key server itself is physically secure. However, it is still network-connected and it acts as a private key oracle, so there are additional threat vectors to consider.

Access to the key server needs to be restricted to trusted parties only. For the private key operations, access needs to be restricted to only trusted edge servers. We employ a dedicated TLS connection between the key server and the edge server, using a separate authentication infrastructure (e.g., dedicated pre-shared keys or certificates from a private PKI). See Section IV-E for discussion of the security of these keys.

The communication between the edge server and the key server is over the public Internet, and this communication is of values that are normally computed internally in a TLS implementation. As noted above, this communication is encrypted using a dedicated TLS connection between the edge server and the key server, but there may still be side channels associated with this communication. In particular, there are two types of side channels to consider.

Timing side channels. If the processing time of the operation performed by the key server is dependent on secret key data, a timing side channel may exist.

- For RSA key transport, it is essential that the key server's implementation of the decryption of the `ClientKeyExchange` message containing the encrypted premaster secret take into account all normal timing side channel protections, such as in [1, §7.4.7.1], to prevent Bleichenbacher's attack [22]. It

should be noted that standalone implementations of RSA PKCS#1v1.5 do not have sufficient side-channel protection, and that the TLS code must add its own side-channel protection, for example by always constructing a random premaster secret first to use in case PKCS#1 padding is incorrect.[2]
- For signed-Diffie–Hellman, standard timing countermeasures for the signing operation should suffice.

Length side channels. The length of the ciphertext response from the key server can be observed by an attacker. If the key server is designed to return constant-length ciphertexts for all requests, the length side channel is eliminated. Note that it is not necessary that ciphertexts for different ciphersuites (RSA key transport versus signed-Diffie–Hellman) be the same length, as which ciphersuite is used is public knowledge after observing `ClientHello` and `ServerHello` messages. There are no additional side channel vectors that are opened up by having several load-balanced key servers.

E. Security of the edge server

When establishing the tunnel, the key server needs to validate the identity of the edge server. This can be achieved with mutually authenticated TLS, but this moves the root of trust to the credential (such as a client certificate or pre-shared key) on the edge server that it uses to authenticate to the key server. The risk to this credential can be limited in a number of ways: by reducing the validity period of the credential and rotating quickly; by using trusted computing mechanisms like remote attestation to ensure that the server is not compromised; by limiting access to the key server through IP filtering; and by revoking access from certain machines based on monitoring.

F. Session resumption

Session resumption allows a TLS client and server to resume a previously established session with an abbreviated handshake, using the previous master secret key to derive new session keys, without performing the expensive public key operations and additional roundtrip required to derive a new premaster secret. There are two mechanisms for session resumption.

Session IDs. As specified in the main TLS standard [1, §7, F.1.4], during the initial handshake the server can store the required state information, then provide the client with a session ID; upon future handshakes, the client includes the session ID in its `ClientHello` message and, if the session ID is valid (has not expired), the server retrieves the state and both parties compute new session keys and `Finished` messages.

- If individual edge servers allow session resumption using session IDs, and keep local session state (rather than sharing it with other edge servers), then no additional security considerations apply compared with session ID-based resumption in normal SSL.
- We do not consider the case of edge servers sharing local session state as it generally goes against the goal of geographic dispersion of edge servers.

[2]See for example OpenSSL's `ssl3_get_client_key_exchange` method in `ssl/s3_srver.c`.

Session tickets. TLS session tickets [17] allow a server to off-load storage of session state to the client by asking the client store the server's state, encrypted under a medium-term session ticket key used by the server. In the initial handshake, the server provides the encrypted session state to the client in a `NewSessionTicket` message before its `Finished` message. In the next handshake, the client returns that encrypted session state in a `SessionTicket` extension in the `ClientHello` message, which the server attempts to decrypt and use.

If individual edge servers use their own local medium-term session ticket key, then no additional security considerations apply compared with use of session tickets in normal TLS.

If edge servers for the same origin server share medium-term session ticket keys, then the client may resume one session for the same origin server at a different edge server, but only when that edge server is acting for the same origin server.

Finally, suppose all edge servers—regardless of the origin server—share medium-term session ticket keys. As in the previous case, the client may resume one session for an origin server at a different edge server. However, since session ticket keys are shared among edge servers regardless of origin server, it is possible that a session resumption request, redirected to an edge server for a different origin server than the original one, would accept. Correct mapping of sessions to hosts now relies on the edge server respecting the HTTP `Host` header.

Further discussion on the challenges in handling session resumption in an HTTPS multiplexing situation is given by Delignat-Lavaud and Bhargavan [14]. Our recommendation for session resumption using session tickets is the second option above, providing a mechanism for load balancing without the risk of virtual host confusion in the third option.

The use of session resumption with renegotiation should take into account the impact of the triple handshake attack [23].

V. Conclusion

By proxying the private key operations in the TLS handshake to a separate key server, content delivery networks can continue to serve many clients with low latency via edge servers while allowing private keys to be maintained in a higher security key server. This architecture can reduce the risk of website operators having their private keys stolen from any edge server in a global network of content delivery servers. Given that commercial CDNs are deploying infrastructures like this, it is important to understand their properties.

We observed this improved latency performance in a small scale proof-of-concept experiment. An interesting line of future work would be a detailed analysis of latency characteristics for TLS proxying on a global content delivery network. We have described the security goals for the TLS proxying architecture and discussed the extent to which RSA key transport and signed-DH ciphersuites in TLS meet these security goals.

While we have phrased our study in terms of content delivery networks, an alternative application of TLS handshake proxying is to the use of one or more SSL termination proxies within a single organization, where the private key operation is outsourced from the SSL termination proxy on the network boundary to a key server in a safer location inside the network.

ACKNOWLEDGEMENTS

D.S. was supported by Australian Research Council (ARC) Discovery Project DP130104304.

REFERENCES

[1] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," RFC 5246 (Proposed Standard), Internet Engineering Task Force, Aug. 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5246.txt

[2] S. K. Card, G. G. Robertson, and J. D. Mackinlay, "The information visualizer, an information workspace," in *CHI'91*. ACM, 1991, pp. 181–188.

[3] C. Partridge, T. Mendez, and W. Milliken, "Host Anycasting Service," RFC 1546 (Informational), Internet Engineering Task Force, Nov. 1993. [Online]. Available: http://www.ietf.org/rfc/rfc1546.txt

[4] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: Cold boot attacks on encryption keys," in *USENIX Security Symposium*. USENIX Association, 2008, pp. 45–60.

[5] C. E. Gero, J. N. Shapiro, and D. J. Burd, "Terminating SSL connections without locally-accessible private keys," Jun. 20 2013, WO Patent App. PCT/US2012/070,075.

[6] CloudFlare Inc., "CloudFlare Keyless SSL," Sep. 2014, https://www.cloudflare.com/keyless-ssl.

[7] N. Sullivan, "Keyless SSL: The nitty gritty technical details," Sep. 2014, https://blog.cloudflare.com/keyless-ssl-the-nitty-gritty-technical-details/.

[8] C. Lesniewski-Laas and M. F. Kaashoek, "SSL splitting: Securely serving data from untrusted caches," *Computer Networks*, vol. 48, no. 5, pp. 763–779, August 2005.

[9] C. Gaspard, S. Goldberg, W. Itani, E. Bertino, and C. Nita-Rotaru, "SINE: Cache-friendly integrity for the web," in *5th IEEE Workshop on Secure Network Protocols (NPSec) 2009*, Oct 2009, pp. 7–12.

[10] T. Moyer, K. Butler, J. Schiffman, P. McDaniel, and T. Jaeger, "Scalable web content attestation," *IEEE Transactions on Computers*, vol. 61, no. 5, pp. 686–699, May 2012.

[11] M. Backes, R. W. Gerling, S. Gerling, S. Nürnberger, D. Schröder, and M. Simkin, "WebTrust - A comprehensive authenticity and integrity framework for HTTP," in *ACNS 14*, ser. LNCS, I. Boureanu, P. Owesarski, and S. Vaudenay, Eds., vol. 8479. Springer, Jun. 2014, pp. 401–418.

[12] G. Medvinsky, N. Nice, T. Shiran, A. Teplitsky, P. Leach, and J. Neystadt, "Authentication delegation based on re-verification of cryptographic evidence," Jun. 5 2008, US Patent App. 11/607,720.

[13] J. Liang, J. Jiang, H.-X. Duan, K. Li, T. Wan, and J. Wu, "When HTTPS meets CDN: A case of authentication in delegated service," in *2014 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2014, pp. 67–82.

[14] A. Delignat-Lavaud and K. Bhargavan, "Virtual host confusion: Weaknesses and exploits," in *Black Hat 2014*, 2014, http://bh.ht.vc/vhost_confusion.pdf.

[15] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1," RFC 2616 (Draft Standard), Internet Engineering Task Force, Jun. 1999. [Online]. Available: http://www.ietf.org/rfc/rfc2616.txt

[16] E. Rescorla, "HTTP Over TLS," RFC 2818 (Informational), Internet Engineering Task Force, May 2000. [Online]. Available: http://www.ietf.org/rfc/rfc2818.txt

[17] J. Salowey, H. Zhou, P. Eronen, and H. Tschofenig, "Transport Layer Security (TLS) Session Resumption without Server-Side State," RFC 5077 (Proposed Standard), Internet Engineering Task Force, Jan. 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5077.txt

[18] T. Jager, F. Kohlar, S. Schäge, and J. Schwenk, "On the security of TLS-DHE in the standard model," in *CRYPTO 2012*, ser. LNCS, R. Safavi-Naini and R. Canetti, Eds., vol. 7417. Springer, Aug. 2012, pp. 273–293.

[19] F. Kohlar, S. Schäge, and J. Schwenk, "On the security of TLS-DH and TLS-RSA in the standard model," Cryptology ePrint Archive, Report 2013/367, 2013, http://eprint.iacr.org/2013/367.

[20] H. Krawczyk, K. G. Paterson, and H. Wee, "On the security of the TLS protocol: A systematic analysis," in *CRYPTO 2013, Part I*, ser. LNCS, R. Canetti and J. A. Garay, Eds., vol. 8042. Springer, Aug. 2013, pp. 429–448.

[21] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P.-Y. Strub, "Implementing TLS with verified cryptographic security," in *2013 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2013, pp. 445–459.

[22] D. Bleichenbacher, "Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1," in *CRYPTO'98*, ser. LNCS, H. Krawczyk, Ed., vol. 1462. Springer, Aug. 1998, pp. 1–12.

[23] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Pironti, and P.-Y. Strub, "Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS," in *2014 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2014, pp. 98–113.

[24] F. Giesen, F. Kohlar, and D. Stebila, "On the security of TLS renegotiation," in *ACM CCS 13*, A.-R. Sadeghi, V. D. Gligor, and M. Yung, Eds. ACM Press, Nov. 2013, pp. 387–398.

[25] N. Mavrogiannopoulos, F. Vercauteren, V. Velichkov, and B. Preneel, "A cross-protocol attack on the TLS protocol," in *ACM CCS 12*, T. Yu, G. Danezis, and V. D. Gligor, Eds. ACM Press, Oct. 2012, pp. 62–72.

[26] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and S. Zanella Béguelin, "Proving the TLS handshake secure (as it is)," in *CRYPTO 2014, Part II*, ser. LNCS, J. A. Garay and R. Gennaro, Eds., vol. 8617. Springer, Aug. 2014, pp. 235–255.

[27] F. Bergsma, B. Dowling, F. Kohlar, J. Schwenk, and D. Stebila, "Multi-ciphersuite security of the Secure Shell (SSH) protocol," in *ACM CCS 14*, G.-J. Ahn, M. Yung, and N. Li, Eds. ACM Press, Nov. 2014, pp. 369–381.

APPENDIX

A. Security experiment

The following formulation of security for TLS handshake proxying follows that of provable security for TLS—the *authenticated and confidential channel establishment (ACCE)* protocol model of Jager et al. [18]—but we omit the technical formalization and instead focus on a more intuitive presentation.

Parties and long-term key generation. The execution environment consists of many parties; the parties are divided into three categories: *clients*, *edge servers*, and *key servers*.

Each key server $\mathcal{K}_i$ and every edge server $\mathcal{E}_j$ has a long-term public key / private key pair. Each key server also has an *origin server*'s long-term public key / private key pair. Clients will be considered to be unauthenticated and have no long-term public key / private key pairs.

We assume that all clients have authentic copies of every origin server long-term public key, e.g. via the web PKI. We assume that each key server $\mathcal{K}_i$ which is willing to delegate sessions to edge server $\mathcal{E}_j$ has an authentic copy of $\mathcal{E}_j$'s long-term public key, and vice versa that $\mathcal{E}_j$ has an authentic copy of $\mathcal{K}_i$'s long-term internal public key, for example set up during edge server configuration/registration or via a public key infrastructure (possibly a different public key infrastructure than that used for distribute host public keys to clients).

Sessions. Each party can execute multiple sessions of the protocol, either concurrently or subsequently. The party maintains per-session variables associated with each session and updates them based on incoming messages and the protocol specification. Among others, the per-session variables include:

- A *status* variable, either in-progress, accept, or reject.
- A *transcript* of the messages sent and received.

- An identifier of the purported *peer(s)*. For three-party TLS handshake proxying: a client records the identity of its purported origin server; an edge server records the identity of its purported peer key server; a key server records the identity of its purported peer edge server and the origin server public key it used.

A complete execution of a session involves interaction between a client and an edge server (called the *client-edge subsession*, interaction between an edge server and a key server (called the *edge-key subsession*), and an operation by the key server using the origin server key.

During execution of a session, eventually a party will set the per-session status variable to either accept or reject.

Adversary interaction. The adversary controls all communication between parties: it directs parties to start sessions, delivers messages to parties, and can reorder, alter, delete, and create messages. The adversary can also compromise certain long-term and per-session variables as indicated below.

- *Session keys*. The adversary can learn the session key (in TLS, the master secret) in any client-edge subsession or any edge-key subsesssion. Such a subsession is then considered *compromised*.
- *Long-term private keys*. The adversary can learn the long-term private key of any edge server or key server, or the origin server private key of any key server. The corresponding key is then considered *compromised*.

Security goals. The security goals for TLS handshake proxying are as listed in Section IV-C: *key-server-to-client authentication* (existence and uniqueness of key server session when client accepts in an uncompromised session); *edge-server-to-client authentication* (existence and uniqueness of edge server session when client accepts in an uncompromised session); and *client-edge-server channel security*.

B. Security of TLS handshake proxying—RSA key transport

In the RSA key transport setting, during the TLS handshake the client sends a randomly chosen premaster secret to the edge server, encrypted under the host public key; this is the `ClientKeyExchange` message. The edge server then establishes an edge-key subsession with the key server holding the desired origin server key and sends the received `ClientKeyExchange` as application data. The key server decrypts and sends the result back to the edge server. Note that it is imperative that the key server implements protections against side channels such as Bleichenbacher's attack on PKCS#1v1.5 decryption [22], for example by randomizing the response in constant time as described in the TLS specification.

Here, the matching condition for authentication between the client and the key server is based solely on the `ClientKeyExchange` / premaster secret: the only message the key server receives is the `ClientKeyExchange`, and the only message it sends is the decrypted result—the premaster secret—so this is the only value on which matching with the key server is possible.

The matching condition for authentication between the client and the edge server is based on the entire TLS transcript.

Existence of key server session. Suppose the client accepts in a session, and the purported peer key server public key is uncompromised. Since the client accepted, it successfully verified the `Finished` message, computing using the master secret (derived from the client-selected premaster secret). Under the assumption that the MAC is unforgeable, the master secret was indeed used to compute the `Finished` message. Under the assumption that the PRF used to derive the master secret is secure, the premaster secret was indeed used to compute the master secret. Under the assumption that TLS RSA key transport is a secure TLS-KEM (see Krawczyk et al. [20]), and that the peer key server public key is uncompromised, it must be that the peer key server at some point in time decapsulated the `ClientKeyExchange` ciphertext and returned the given premaster secret. Thus a key server session exists.

Existence of edge server session. Suppose the client accepts in a session, and the purported peer key server public key is uncompromised. Suppose further that no edge server's public key is compromised between when the client sends its first message and receives the response. As previously identified, there exists a key server session, and thus the key server really did decrypt the `ClientKeyExchange` message using the key server's private key. Except with negligible probability, the client's premaster secret in the `ClientKeyExchange` is unique, and thus was not decapsulated by the key server prior to the client sending its first message in this session. Since no edge server's public key is uncompromised between when the client sends its first message and receives the response, under the assumption that edge-key TLS connection provides secure authentication, only honest edge servers obtain decapsulations from the key server during this time. Thus an honest edge server must have existed that requested a decapsulation from the key server on that `ClientKeyExchange` message, and so the edge server has a matching client-edge subsession.

(Non)-Uniqueness of key server session. Uniqueness of key server sessions *cannot* be guaranteed. An adversary observing the messages exchanged between a client and an edge server can replay the observed `ClientKeyExchange` message in a new session to the same (or a different) edge server, who will then request of the key server that same `ClientKeyExchange` be decrypted again using the key server key, resulting in a second identical session at the key server.

Uniqueness of edge server session. By the same argument in the existence of an edge server session, during the time between when the client sends its first messages and receives the response, only honest edge servers obtain decapsulations from the key server. Except with negligible probability, honest edge servers generate unique `server_random` nonces. Thus there exists a unique edge server session matching the client's session when the client verifies the server's `Finished` message.

Channel security. Confidentiality and integrity of application data on the client-edge channel follows immediately from edge-server-to-client authentication above and the existing channel security of TLS RSA-key transport ciphersuites [19], [20].

C. Security of TLS handshake proxying—signed-DH

In the signed-Diffie–Hellman setting, the key server signs the `ServerKeyExchange` message, which consists of the `client_random` and `server_random` values and the DH

parameters, either ServerDHParams or ServerECDHParams (for finite-field or elliptic curve DH, respectively).

Existence of key server session. Suppose the client accepts in a session, and the purported peer key server public key is uncompromised. Since the client accepted, it successfully verified a signature under the key server's public key of the ServerKeyExchange message the client received. Under the assumption that the signature scheme is unforgeable, the key server's private key was indeed used to sign the ServerKeyExchange message, thus a key server session exists.

Existence of edge server session. Suppose the client accepts in a session, and the purported peer key server public key is uncompromised. Suppose further that no edge server's public key is compromised between when the client sends its first message and receives the response. As previously identified, there exists a key server session, and thus the key server really did sign the ServerKeyExchange message using the key server's private key. Except with negligible probability, the client's nonce client_random in the ServerKeyExchange is unique, and thus was not signed by the key server prior to the client sending its first message in this session. Since no edge server's public key is uncompromised between when the client sends its first message and receives the response, under the assumption that the edge-key TLS connection provides secure authentication, only honest edge servers obtain signatures from the key server during this time. Thus an honest edge server must have existed that requested a signature from the key server on that ServerKeyExchange message, and so the edge server has a matching client-edge subsession.

Uniqueness of key server session. By the same argument in the existence of an edge server session, during the time between when the client sends its first messages and receives the response, only honest edge servers obtain signatures from the key server. Except with negligible probability, honest edge servers generate unique server_random random nonces. Thus there exists a unique key server session matching the client's session at the time the client verifies the signature.

Uniqueness of edge server session. By the same argument in the existence of an edge server session, during the time between when the client sends its first messages and receives the response, only honest edge servers obtain signatures from the key server during this time. Except with negligible probability, honest edge servers generate unique server_random random nonces. Thus there exists a unique edge server session matching the client's session at the time the client verifies the signature.

Channel security. Confidentiality and integrity of application data on the client-edge channel follows immediately from edge-server-to-client authentication above and the existing channel security of TLS signed-DH ciphersuites [18], [20].

D. Limitations of security analysis

The analysis in this paper inherits many of the limitations of previous work on analyzing TLS using the authenticated and confidential channel establishment framework [18].

It does not cover the use of TLS's renegotiation feature in either the client-edge or edge-key subsessions. Renegotiation has been separately analyzed using an extension of ACCE [24], which could conceivably be applied to this setting, but would add a layer of complexity.

It does not cover the use of TLS's session resumption feature in either the client-edge or edge-key subsessions. Session resumption currently does not seem to have the properties it would need to be proven secure based on the recent triple handshake attack [23].

It does not cover the use of the same long-term key in different ciphersuites, for example using the same signing key with both finite field and elliptic curve Diffie–Hellman, or using the same RSA key with both signed-DH ciphersuites and RSA key transport ciphersuites, both of which are common in practice. Some combinations are known to be insecure as demonstrated by the signed-DH/signed-ECDH cross-protocol attack by Mavrogiannopoulos et al. [25]. Some progress has been made on specific combinations [26] and on developing a more generic approach [27], but these cannot in general overcome the aforementioned cross-protocol attack without changes to the TLS protocol.