Content delivery over TLS: a cryptographic analysis of keyless SSL

Karthikeyan Bhargavan, Ioana Boureanu, Cristina Onete, Pierre-Alain Fouque, Benjamin Richard

Content Delivery over TLS: A Cryptographic Analysis of Keyless SSL

Karthikeyan Bhargavan *, Ioana Boureanu †, Pierre-Alain Fouque ‡, Cristina Onete §, Benjamin Richard ¶
* *Inria de Paris*
karthikeyan.bhargavan@inria.fr
† *University of Surrey*
i.boureanu@surrey.ac.uk
‡ *University of Rennes 1/IRISA France*
pa.fouque@gmail.com
§ *INSA Rennes/IRISA France*
cristina.onete@gmail.com
¶ *Orange Labs Châtillon, France*
benjaminrichard913@gmail.com

***Abstract*—The Transport Layer Security (TLS) protocol is designed to allow two parties, a client and a server, to communicate securely over an insecure network. However, when TLS connections are *proxied* through an intermediate middlebox, like a Content Delivery Network (CDN), the standard end-to-end security guarantees of the protocol no longer apply. In this paper, we investigate the security guarantees provided by Keyless SSL, a CDN architecture currently deployed by CloudFlare that composes two TLS 1.2 handshakes to obtain a proxied TLS connection. We demonstrate new attacks that show that Keyless SSL does not meet its intended security goals. These attacks have been reported to CloudFlare and we are in the process of discussing fixes.**

We argue that proxied TLS handshakes require a new, stronger, 3-party security definition. We present 3(S)ACCE-security, a generalization of the 2-party ACCE security definition that has been used in several previous proofs for TLS. We modify Keyless SSL and prove that our modifications guarantee 3(S)ACCE-security, assuming ACCE-security for the individual TLS 1.2 connections. We also propose a new design for Keyless TLS 1.3 and prove that it achieves 3(S)ACCE-security, assuming that the TLS 1.3 handshake implements an authenticated 2-party key exchange. Notably, we show that secure proxying in Keyless TLS 1.3 is computationally lighter and requires simpler assumptions on the certificate infrastructure than our proposed fix for Keyless SSL. Our results indicate that proxied TLS architectures, as currently used by a number of CDNs, may be vulnerable to subtle attacks and deserve close attention.

1. Introduction

To protect sensitive data as it is communicated over an insecure network, such as the Internet, we rely on cryptographic protocols that implement secure channels. Traditionally, such protocols combine an *authenticated key-exchange* (AKE) phase that establishes *channel keys* with a subsequent *authenticated encryption* (AE) phase. This second step uses the computed keys to protect streams of messages between honest endpoints from powerful adversaries, who are assumed to control the network as well as any number of malicious endpoints.

Transport Layer Security. The Transport Layer Security (TLS) protocol is the most widely deployed secure-channel protocol on the Internet. For example, connections between modern web browsers and popular websites are secured using HTTP over TLS (HTTPS). With increased concerns over both mass surveillance and criminal activity on the Internet, the use of TLS is slowly becoming mandatory for many use cases, including the Web.[1]

TLS is designed to be used between a client and a server, typically authenticated using public-key certificates. In TLS, the initial AKE phase is called the *handshake* and the subsequent AE phase is called the *record*. The handshake protocol supports a number of modes, such as RSA key transport, Diffie-Hellman key exchange, or pre-shared keys. The record protocol also supports a variety of constructions, including stream ciphers, block ciphers, and modern AE with additional data (AEAD) schemes.

A series of papers have developed a precise cryptographic specification for TLS, called authenticated and confidential channel establishment (ACCE) [8], and proved that various combinations of handshake and record modes achieve this specification. A variant of ACCE, called SACCE, applies to the common case where only the server is authenticated, but the client remains anonymous [10].

On the other hand, a number of modes and constructions used in TLS have also been shown to be insecure, resulting in high-profile attacks on both the handshake and record layers [1], [3]. The next version of TLS (1.3) is currently being standardized, and it is designed to avoid many of the pitfalls in earlier versions of the protocol.[2] Early drafts of

1. See the latest HTTP/2 draft: http://http2.github.io/http2-spec/
2. See the latest TLS 1.3 draft at https://tlswg.github.io/tls13-spec/

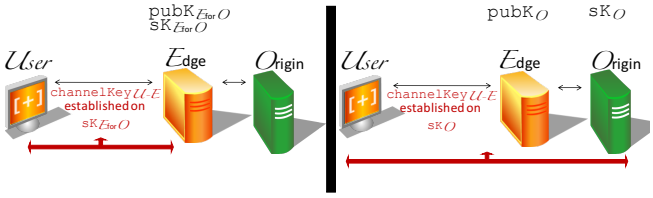

Figure 1: TLS in Classic CDNs (left-hand side); Cloudflare's Keyless SSL (right-hand side)

this new protocol have already been analyzed for security in models similar to ACCE [6], [9], [11].

By systematically deploying TLS on all websites, by eradicating obsolete cryptographic constructions, and by relying on strong security theorems for modern versions of the protocol, practitioners can significantly improve the security of the Web. However, the use of end-to-end encryption protocols like TLS raises new challenges for widely-used security mechanisms and performance enhancements that require third parties (proxies) between client and servers.

Content Delivery Networks. TLS connections may potentially be used to transfer large amounts of data (*e.g.*, movies in streaming applications). If the content owner (the *origin server*) and the receiver (the client) are situated geographically far away, data transfer will be slow, involving extensive routing. To speed up such connections, content owners can hire Content Delivery Networks (CDNs) that cache popular content at *edge servers* located around the world and deliver them to clients based on geographic proximity.

Deploying CDNs for public HTTP traffic is relatively straight-forward. The origin server (*e.g.*, example.com) decides what content will be cached by the CDN and puts it on some subdomain (*e.g.*, cdn.example.com). The origin server and the CDN use the DNS protocol to direct requests for this subdomain to the nearest edge server. The client sees a significant performance improvement, but is otherwise unaware that it is not directly connected to the origin server.

For HTTPS connections, however, this is more problematic. The client expects a direct *secure* channel to the origin server, and redirection to an unknown edge server will be seen as an attack. Consequently, the origin server and the CDN need to agree on a way by which edge servers can accept connections on behalf of the origin server.

In the classic case, the CDN generates a secret and public-key pair and, on these, provisions a certificate on behalf of certain domains of the origin server (*e.g.*, for cdn.example.com). This effectively allows the CDN's edge servers to impersonate the origin server, as depicted on the left-hand side of Figure 1. This architecture works well for simple origin servers who wish to delegate all TLS-related operations to the CDN. The main risk is that any attack on the edge server will expose the private key to an adversary who can then impersonate the origin server, at least until the corresponding certificate expires or is revoked.

To mitigate such risks, especially for "high-value" origin servers who may require some CDN-based performance-boost but do not want the CDN to handle long-term private keys on their behalf, CloudFlare implemented a version of proxied TLS called *Keyless SSL* [13]. A patent by Akamai on this matter existed already [7], yet a solution had not been deployed commercially before CloudFlare's.

In this setup, depicted on the right-hand side of Figure 1, edge servers do not get the certificate private key. Instead, the client, edge server, and origin server must now engage in a 3-party *proxied handshake*, where the client and edge server still compute channel keys for the TLS connection, but all private key operations are deferred to the origin server. The informal security goals of this novel design were outlined in [13] but they were not formally stated as a cryptographic definition. In this paper, we provide the first cryptographic analysis of Keyless SSL, in terms of a new 3-party security definition for proxied TLS connections.

3(S)ACCE-Security. We define 3(S)ACCE, a non-trivial adaptation of traditional, 2-party ACCE security when the handshake is run in the presence of middleboxes/middleware (MW) such as CDN edge servers. Notably, the middlebox is only authorised to access a subset of the server's contents, and it is a potentially malicious party. Our 3(S)ACCE model captures the characteristics of several types of proxied handshakes including classic CDNs and Keyless SSL.

Our notion of 3(S)ACCE encompasses four properties: (i) *entity authentication* for both middleboxes and servers (but not necessarily for clients); (ii) *channel security* for the client-to-middlebox and for the middlebox-to-server link; (iii) *accountability*, which says that if the client believes it is speaking to a server, then that server is able to compute the secret key for that session; (iv) *content soundness*, which says that the middlebox may not forward content that it has not been authorized to deliver by the server. All these propoerties need to hold in the presence of network attackers, malicious servers, and malicious middleboxes.

The first two properties are 3-party extensions of the corresponding preoperties in 2-ACCE, but take into account the possibility of malicious and compromised middleware. Content soundness captures the notion that middleware is authorized to deliver only specific sub-parts (*e.g.*, subdomains) of a server's content. Accountability requires that whenever a client identifies a server as its partner, that server should be able to decrypt and hence audit channels established between the client and middleware. This limits what malicious middleware can do without the server's knowledge. Note that classical CDNs do not guarantee accountability, since middleware may pose as the server independently of the server. However, we advocate it as a desirable security goal for proxied TLS architectures.

Our 3(S)ACCE-security definition is of independent interest for other proxied secure channel architectures. It relies on a novel notion of partnering in which one instance is partnered either with one peer instance (when the connection is not proxied), or with three other instances (for proxied connections). However, the definition still has many limitations. For example, it does not handle authenticated clients, and it does not require forward secrecy. We leave such

extensions of our model for future work.

Breaking and Fixing Keyless SSL. We present two attacks on Keyless SSL that break its intended security goals by relying on malicious middleware. The key weakness we expose and exploit is that Keyless SSL turns origin servers into signing and decryption oracles for the private key. These servers have no context for the values they are being asked to decrypt or sign, and so they are not able to know whether they correspond to a valid client session, or whether they are valid TLS values at all. As a consequence, we show that an attacker who can obtain an edge server's private key can decrypt all prior RSA-based connections to any edge server, hence breaking channel security. We also demonstrate a cross-protocol attack that enables an attacker who compromises an edge server's private key to set up a long-term QUIC server impersonating the origin server.

We propose a modification of the original Keyless SSL protocol that prevents these attacks and provably achieves 3(S)ACCE-security. Although close to the original architecture, our new protocol comes with several performance drawbacks. First, we increase the computational workload of the server whenever the latter aids the middlebox. Second, in order to achieve accountability, the middlebox may not use session resumption and will need the server to always be online. Third, our proofs of entity authentication and channel security require the use of an export key between the middlebox and the server. Fourth, to fully provide content soundness in the presence of malicious middleware, we need to provision one certificate per middlebox. However, some of these restrictions can be relaxed to obtain weaker guarantees. For example, we can allow some session resumption at the cost of losing some accountability, and we can reduce the number of certificates if we assume a weaker threat model.

We also present a new design for Keyless SSL applied to the current draft of TLS 1.3 and prove it to be 3(S)ACCE-secure. Interestingly, the complexity of the PKI, as well as the computational burden on the server, can be much reduced in the case of 3(S)ACCE-security over TLS 1.3. Various middleboxes can share the same public key and the same certificate. At the expense of some extra verifications and a nonce generation, the server can be reduced to acting as a signature oracle. Even our TLS 1.3 version, however, precludes session resumption. In the full version, we show how to attain 3(S)ACCE-security *and* allow session resumption, if we allow the client to be *aware* that the handshake is legitimately proxied.

Related Work. We discuss closely related work throughout the paper. While there is a wealth of literature on the cryptographic analysis of various versions and applications of TLS [5], [6], [8]–[11], three-party TLS scenarios have not received as much attention. We primarily refer to [13] for a description and informal analysis of Keyless SSL. A different proxying scenario is considered in mcTLS [12], which modifies the TLS handshake so that clients and servers can both explicitly grant read/write permissions to middleboxes. In contrast, we only consider solutions that use unmodified TLS handshakes between clients and middleboxes.

Our Contributions. In this paper, we investigate the security of proxied 3-party TLS scenarios, focusing on the case of Keyless SSL. We claim the following main contributions:

- we analyse the security of Keyless SSL against its intended goals, revealing several attacks (Section 2);
- we introduce the 3(S)ACCE model as a security specification for proxied TLS handshakes (Section 4);
- we modify Keyless SSL and prove it to be 3(S)ACCE-secure, at some cost to performance (Section 5);
- we design Keyless TLS 1.3, an efficient proxied handshake architecture for TLS 1.3, and prove that it attains 3(S)ACCE-security (Section 6);
- in the full version, we describe a modular 3(S)ACCE-secure protocol that can be instantiated with TLS 1.2 or TLS 1.3 and supports session resumption.

2. Proxied TLS Connections: Keyless SSL

The TLS protocol (formerly known as SSL) supports many versions, extensions, and ciphersuites. Each TLS connection begins with a *handshake* that negotiates the version and other protocol parameters and then executes an authenticated key exchange. In this section, we focus on the two most commonly-used handshake modes of TLS 1.2[3]

The RSA Handshake. In TLS-RSA, the client C sends a nonce N_C to the server S, which responds with its own nonce N_S and an RSA public-key certificate Cert_S. The client then generates and sends a *pre-master secret* pmk encrypted under the server's public key. The server decrypts pmk and the client and server both compute a *master secret* msk using pmk and the two nonces. To complete the handshake, both client and server use msk to MAC the full handshake transcript and send (in an encrypted form) these MACs to each other in *finished messages* ($\mathsf{Fin}_C, \mathsf{Fin}_S$). These messages provide key confirmation, and also help detect whether a network attacker has tampered with the handshake messages. At the end of the handshake, both client and server derive connection keys ck from msk and the two nonces, and these keys are subsequently used for authenticated encryption of application data in the record phase. (In fact, these keys have already been used to encrypt the finished messages.)

The TLS-RSA protocol is the oldest and most popular handshake mode in TLS, but it has recently fallen out of favour because it does not provide *forward secrecy*, which means that if an adversary records a TLS-RSA connection and much later compromises the server's private key, it can decrypt the pmk, derive the connection keys, and read all application data. This threat may seem unrealistic, but with new concerns about mass surveillance by powerful adversaries, TLS-based applications increasingly require forward secrecy by default. Moreover, TLS uses an RSA encryption mode called RSA-PKCS#1v1.5, which has proved vulnerable to a series of increasingly-effective padding-oracle attacks, first described by Bleichenbacher,

3. See https://tools.ietf.org/html/rfc5246 for full details.

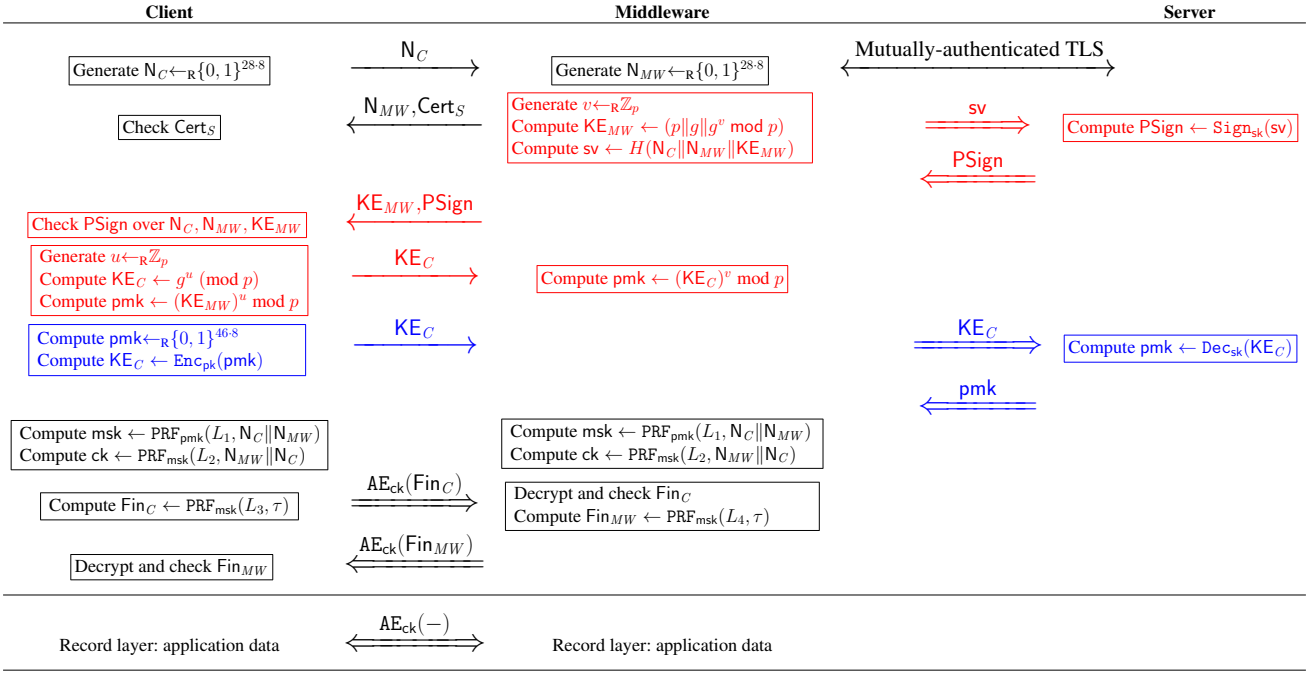

Figure 2: The Keyless SSL architecture as implemented by Cloudflare: (blue+black) TLS-RSA; (red+black): TLS-DH.

that have proved hard to fix. For both these reasons, the TLS working group is getting rid of RSA in TLS 1.3.

The DHE Handshake. In TLS-DHE, the client and server first exchange nonces and the server certificate just like in TLS-RSA. Then the server chooses a Diffie-Hellman group (p, q, g) (represented by an elliptic curve or by an explicit prime field) and generates a keypair $(v, g^v (\text{mod } p))$. It signs the nonces, the group, and its Diffie-Hellman public value with its certificate private key and sends them to the client, which then generates its own keypair $(u, g^u (\text{mod } p))$. Both client and server then compute the pre-master secret pmk as $g^{uv} \pmod p$. The rest of the protocol and computations $(\mathsf{msk}, \mathsf{ck}, \mathsf{Fin}_C, \mathsf{Fin}_S)$ proceed as in TLS-RSA.

The key advantage of TLS-DHE is that it does provide forward secrecy as long as both parties use a strong Diffie-Hellman group and generate fresh keypairs for each handshake. Furthermore, the elliptic curve variant of TLS-DHE is considered to be as fast (if not faster) than TLS-RSA.[4]

Proxied TLS in Keyless SSL. In proxied TLS, the client C wants to connect to a server S and is instead redirected to a geographically close CDN edge server (or middlebox) MW that serves cached content from S. The server S trusts MW enough to allow it to decrypt requests and encrypt responses for this content. However, unlike in classical CDNs, S does not want MW to impersonate it indefinitely; instead, S wants to keep full control over its long-term private keys.

In both TLS-RSA and TLS-DHE, the server authenticates by proving possession of its certificate private key, using it to either decrypt some secret (TLS-RSA) or to sign some value (TLS-DHE). The key observation of proxied TLS, as developed by CloudFlare in Keyless SSL, is that the middleware MW does not need to be given the server's private key as long as it can query the latter when it needs a signature or decryption capabilities.

This leads to the design in Figure 2. The client C and middleware MW execute a standard TLS-RSA or TLS-DHE handshake, where MW uses S's certificate. In TLS-RSA, C sends a client key-exchange message KE_C containing the encrypted pmk and MW forwards it to S's key server, which decrypts and returns pmk to MW. In TLS-DHE, MW generates the DHE keypair, composes the hashed value sv to be signed, and sends it to S's key server, which signs and returns the signature. All other computations are performed by MW with no assistance from S.

The queries from MW to the key server S are performed over a mutually-authenticated TLS channel. CloudFlare issues a client certificate (with a distinguished issuer) to each edge server MW, and a key server certificate to each S.

Keyless SSL is engineered for high performance. Most of the computation can be performed by edge servers; the key server can remain oblivious of the details of TLS. The additional cost of proxying is reduced to a single round-trip between the edge server and the key server. Furthermore, by using session resumption, the edge server can use the same master secret msk over many TLS connections with the same client, without needing to recontact S.

Security Goals of Keyless SSL. The security goals for Keyless SSL were informally described in [13], where the

4. TLS distinguishes between DH modes based on explicit primes from those based on elliptic curves, but for the results in this paper, the difference between the two is immaterial, and we refer to both as TLS-DHE.

authors observed that the addition of the third party MW necessitated a few new security goals.

In classic 2-party TLS with server-only authentication, as long as a server's private key is kept secret, we expect (i) *server-to-client authentication*, which says that the attacker cannot impersonate the server or otherwise interfere in handshakes between honest clients and the server, and (ii) *channel security*, which says that the attacker cannot read, alter, or insert application data on connections between an honest client and an honest server. These goals must hold in a threat model in which the attacker controls the network and any number of dishonest clients and malicious or compromised servers. Many variants of these goals have been previously formalized and proved for TLS. In this paper, we assume that classical TLS satisfies them.

In proxied TLS, the main new threat is that we need to consider malicious or compromised middleboxes MW. In "classic" CDNs, MW holds a long-term private key identifying S, and MW uses it on S's behalf on the C–MW side of the TLS-connections. So, any attack that leaks secrets stored at MW (*e.g.*, HeartBleed) could compromise this private key and allow the adversary to then impersonate S.

In Keyless SSL, the server keeps the private key and it may even store it securely in a hardware security module. So, the real threat is from attackers who learn the private key of some middlebox MW and use it to query the key-server and thereby impersonate the server. The key goal is that once the key-server learns of this compromise and de-authorizes the middlebox's certificate, the attacker should no longer be able to interfere with connections.

To reflect this intuition, [13] presents three goals for proxied TLS. The first two, *key-server-to-client* and *edge-server-to-client* authentication, generalize the authentication goals to three parties. The third generalizes *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.

That is, the confidentiality and integrity of a proxied TLS connection is guaranteed only if the server's private key is kept secret, and also only if *all middleboxes' private keys are secret until the end of the handshake*.

Are these new security definitions adequate? Does Keyless SSL satisfy them? [13] presents informal arguments about why these properties hold in Keyless SSL. In the rest of this paper, we seek to answer these questions through a detailed cryptographic analysis. But first, we show that even with the informal definitions above, Keyless SSL is vulnerable to important attacks.

A Middleware Attack on Keyless TLS-RSA. As discussed earlier, TLS-RSA does not provide forward secrecy, so if the server's private key is compromised, the adversary can decrypt previously recorded connections. In Keyless SSL, if we assume that the server's private key is kept secret, we should normally not have to worry about forward secrecy. However, we still need to consider compromised middleware.

Suppose an attacker records all the messages in a proxied TLS-RSA connection between C, MW, and S. Much later, suppose that the attacker compromises the private key for some middleware MW' that is authorized to query the key server S. The attacker can use this private key to establish a mutually-authenticated connection with S, it can ask S to decrypt the encrypted pmk for any previous connection and thereby decrypt its contents. This attack directly contradicts the channel security property of [13] presented above, since we only compromised the edge server *after* the connection was complete[5]. The attack can either be read as a forward secrecy attack (MW compromised after the connection is complete) or a cross-middlebox attack (MW' is compromised to break C's connection with MW).

The attack is particularly worrisome for CDNs, implying that an attacker who compromises an edge server in country A will be able to decrypt all prior TLS-RSA conversations recorded in any country B. This emphasizes the new risks of proxied TLS: it strictly reduces the security of client-server connections by increasing the attack surface to include middleboxes distributed around the world. We are in discussions with CloudFlare to fix this attack; all fixes require the key server to do more than just act as a decryption oracle. For example, a minimal fix would be for the key server to generate the server nonce N_S and then, when the edge server queries it with the two nonces and the encrypted pmk, S directly derives msk and returns it. This ensures that a compromised MW cannot make decryption queries on old connections (since N_S will be different).

A Cross-Protocol Attack on Keyless TLS-DHE. Proxied TLS-DHE in Keyless SSL is vulnerable to a different attack, which also exploits the oracle-access to the key server. The key server is willing to sign any hashed value that the middlebox MW provides. Note that the key server cannot check, even if it were willing to do so, that the value it is signing is the hash of a valid TLS-DHE server key exchange message (KE_{MW}). This leads to a cross-protocol attack between Keyless SSL and QUIC.

The QUIC protocol[6] was proposed by Google as a faster alternative to TLS 1.2 and it is transparently used (instead of TLS) on most HTTPS connections from the Chrome web browser to servers that support it. QUIC servers reuse the same X.509 server certificates as TLS to execute a handshake similar to TLS-DHE. However, instead of per-connection signatures, QUIC requires each server to sign a long-term server configuration message SCFG that contains a semi-static Diffie-Hellman key. Once a client has obtained and cached a signed SCFG, it can use it for many

5. The authors of [13] acknowledged this attack and are seeking to fix their definition.

6. See https://www.chromium.org/quic.

connections to the server, without needing any new server signatures, until the configuration expires.

The format of the signed value in QUIC is distinct from that of TLS. For example, the signed value begins with the ASCII string "QUIC server config signature". This value is first hashed using SHA-256, then signed *e.g.*, using ECDSA. The key observation that leads to our attack is that once the signed values in QUIC or TLS are hashed, the key server cannot tell the difference between them.

Suppose the attacker has compromised the private key of some middlebox MW. It then composes a QUIC SCFG message containing its own Diffie-Hellman public value and a large expiry time ($2^{64}-1$ seconds). It sends this message to the key server, which will sign it thinking it is for a TLS-DHE handshake. The attacker can now pretend to be a QUIC server for S until the configuration expires, *even though S never intended to support QUIC*. De-authorizing the middlebox does not stop this attack.

The flaw in Keyless SSL that enables this attack is that the key server blindly signs messages without checking them. For example, the attack is prevented if MW provides the client nonce and key exchange value to the key server S, and S generates the server nonce and compiles the value to-be-signed before hashing and signing it.

The Problem with Session Resumption. Once a client has established a TLS session with a server, it does not have to redo the full handshake on new connections. Instead, it can rely on an abbreviated handshake, commonly called session resumption, that relies on the client and server storing the master secret msk and other parameters of previously established sessions. A variant called session tickets allows servers to offload session storage to clients.

The extensive use of session resumption in modern web browsers is crucial to the efficiency of Keyless SSL, implying that for a majority of connections, the edge server need not contact the key server. However, resumption also allows an adversary who has compromised an edge server to create a session ticket with a long expiry time, and then impersonate the server on resumed connections until the session expires, even if the edge server is de-authorized immediately. This attack is hard to prevent without changing the way web browsers work; thus, for strong security against malicious middleboxes, we forbid session resumption in proxied TLS, except in special cases.

Towards a Stronger Security Definition. We have described two concrete attacks that break the intended security goals of Keyless SSL. These attacks have been acknowledged by CloudFlare, and we are working on fixes. Before fixing Keyless SSL, however, it is worth asking if the original goals were the right ones in the first place, or whether they are too weak and need to be strengthened.

The channel security definition from [13] quoted above only applies if none of the middlebox private keys is compromised. Suppose an honest client managed to connect to an uncompromised middlebox MW in country A. The definition says that this connection is not guaranteed to be secure if the attacker can compromise some edge server MW' in any country B. However, it seems valuable to strengthen the goals to require security for connections to honest middleboxes even if other middleboxes were compromised.

The authentication goals in [13] are also quite weak. The client authenticates the key server, the edge server authenticates the key server, but there is no guarantee that the client and edge server agree on the identity of the key server. That is, the definitions allow the case where the client thinks it is connected to S via MW but MW thinks it is connected to S'. In the CDN context, MW would then be serving content from S' (instead of S) to C, and this becomes a serious attack which is not forbidden by the authentication goals.

More generally, extending two-party secure channel definitions to three-party scenarios like proxied TLS requires close attention or it may leave gaps that miss new attacks. Over the next two sections, we explore and present a formal definition for proxied TLS that attempts to close these gaps.

3. Background: ACCE for 2-party TLS

An authenticated key-exchange (AKE) protocol is said to be secure if a MiM (man-in-the-middle) adversary cannot distinguish the established session-keys from random keys [2]. However, TLS 1.2 handshakes do not meet this security definition since the channel keys are used to encrypt the finished messages, prior to the end of the key exchange. This provides a real-from-random distinguishing oracle for the MiM attacker. After years of struggling to find a definition that captures the channel-security of TLS, researchers developed the notion of ACCE-security [8], which requires that the keys generated by the TLS handshake can safely be used for authenticated encryption in the record layer.

In this section, we briefly describe the ACCE security model (which we call *2-ACCE*), using the notations of Brzuska et al. [5] for more details, see our full version.

Parties and instances. The 2-ACCE model considers a set $\mathcal{P}$ of parties, which can be either *clients* $C \in \mathcal{C}$ or *servers* $S \in \mathcal{S}$. Parties are associated with private keys sk and their corresponding, certified public keys pk. The adversary can interact with parties in concurrent or sequential executions, called sessions, associated with single party *instances*. We denote by π_i^m the m-th instance (execution) of party P_i. Each instance is associated with the following attributes:

- the instance's **secret**, resp. **public keys** $\pi_i^m.\mathsf{sk} := \mathsf{sk}_i$ and $\pi_i^m.\mathsf{pk} := \mathsf{pk}_i$ of P_i. In unilaterally-authenticated handshakes, clients have no such parameters, thus we set $\pi_i^m.\mathsf{sk} = \pi_i^m.\mathsf{pk} := \perp$.
- the **role** of P_i as either the *initiator* or *responder* of the protocol, $\pi_i^m.\rho \in \{init, resp\}$.
- the **session identifier**, $\pi_i^m.\mathsf{sid}$ of an instance, set to $\perp$ for non-existent sessions.
- the **partner identifier**, $\pi_i^m.\mathsf{pid}$ set to $\perp$ for non-existent sessions. This attribute stores either a party identifier P_j, indicating the party that P_i believes it is running

the protocol with (in unilateral authentication, clients are associated with a label "Client").

- the **acceptance-flag** $\pi_i^m.\alpha$, originally set to $\perp$ while the session is ongoing, but which turns to 1 or 0 as the party accepts or rejects the partner's authentication.
- the **channel-key**, $\pi_i^m.\mathsf{ck}$, which is set to $\perp$ at the beginning of the session, and becomes a non-null bitstring once π_i^m ends in an accepting state.
- the **left-or-right** bit $\pi_i^m.\mathsf{b}$, sampled at random when the instance is generated. This bit is used in the key-indistinguishability and channel-security games.
- the **transcript** $\pi_i^m.\tau$ of the instance, containing the suite of messages received and sent by this instance, as well as all public information known to all parties.

The definition of 2-ACCE security heavily relies on the notion of *partnering*. Two instances π_i^m and π_j^n are said to be **partnered** if $\pi_i^m.\mathsf{sid} = \pi_j^n.\mathsf{sid} \neq \perp$.

Games and adversarial queries. In the 2-ACCE game, the adversary interacts with parties by calling *oracles* and making queries. It can generate new instances of P_i by calling the $\mathsf{NewSession}(P_i, \rho, pid)$ oracle. It can send messages by calling the $\mathsf{Send}(\pi_i^m, M)$ oracle. It can learn the party's secret keys via $\mathsf{Corrupt}(P_i)$ queries, and it can learn channel keys (for accepting instances) by querying $\mathsf{Reveal}(\pi_i^m)$. A $\mathsf{Test}(\pi_i^m)$ query outputs either the real channel keys $\pi_i^m.\mathsf{ck}$ computed by the accepting instance π_i^m or random keys of the same size. As opposed to standard AKE security, in the 2-ACCE game, the adversary is also given access to two oracles, $\mathsf{Encrypt}(\pi_i^m, l, M_0, M_1, H)$ and $\mathsf{Decrypt}(\pi_i^m, C, H)$, which allow some access to the secure channel established by two instances. The output of both these oracles depends on the hidden bit $\pi_i^m.\mathsf{b}$ for any instance π_i^m.

The adversary's *advantage* to win is defined in terms of its success in two security games, namely *entity authentication* and *channel security*, the latter of which is subject to the following freshness definition.

Session freshness. A session π_i^m is *fresh* with intended partner P_j, if, upon the last query of the adversary $\mathcal{A}$, the uncorrupted instance π_i^m has finished its session in an accepting state, with $\pi_i^m.\mathsf{pid} = P_j$, for an uncorrupted P_j, such that no Reveal query was made on π_i^m, π_j^n.

2-ACCE Entity Authentication (EA). In the EA game, the adversary queries the first four oracles above and its goal is to make one instance, π_i^m of an uncorrupted P_i *accept maliciously*. That is, π_i^m must end in an accepting state, with partner ID P_j, also uncorrupted, such that no other unique instance of P_j partnering π_i^m exists. The adversary's advantage in this game is its winning probability.

2-ACCE Security of the Channel (SC). In this game, the adversary $\mathcal{A}$ can use all the oracles except Test and must output, for a fresh instance π_i^m, the bit $\pi_i^m.\mathsf{b}$ of that instance. The adversary's advantage is the absolute difference between its winning probability and $\frac{1}{2}$.

An additional assumption. Ideally, since we construct the proxied handshake from a single, unilaterally authenticated TLS negotiation (between the client and the middleware), and a mutually-authenticated one (between the middleware and the server), we would want to reduce the security of our schemes only to the basic 2-SACCE and 2-ACCE games, respectively. The two differ only in their Entity Authentication property, with 2-SACCE restricting the adversary's *winning condition* to client instances π_i^m only.

However, for technical reasons, we need to rely on a slightly different security notion, in which clients are also issued certificates. We associate party instances with a bit denoted flag, called a *mode-flag*, which is set to 0 by default. If the client receives a known, constant message denoted $\mathsf{prompt.flag}$ from the server, then his flag bit is set to 1, and the client is expected to authenticate. If the flag is 0, the protocol is run with server-only authentication. In practice, the $\mathsf{prompt.flag}$ message for TLS is the Certificate Request message sent by the server to the client[7]. The standard 2-(S)ACCE matching-conversation definitions (which we did not include here) can be adapted trivially to capture that partnering instances must have matching mode-flags' values. In addition, the EA game incorporates these mode-flags as follows:

Mixed-2-ACCE Entity Authentication (mEA). In the mEA game, the adversary queries the first four oracles above and its goal is to make one instance, π_i^m of an uncorrupted P_i *accept maliciously*. That is, π_i^m must end in an accepting state, with partner ID P_j also uncorrupted, such that no other unique instance of P_j partnering π_i^m exists. Furthermore, let flag_i^m denote the mode-flag for the instance π_i^m. Furthermore, if $\mathsf{flag}_i^m = 0$, then P_i *must* be a client only. The adversary's advantage in this game is its winning probability.

Proving 2-ACCE for TLS. The first ACCE-definition for TLS came with a 2-ACCE-security proof for TLS-DHE in [8]. This was followed by the systematic analysis of TLS 1.2 by [10], but strictly in a two-party setting. In fact, [4] describes an attack which involves using the TLS handshake between three parties, one of which is malicious. Although it is not known whether TLS 1.2 is mEA-secure as adapted above, this assumption seems quite reasonable, given its unilateral and respectively mutual security proofs. In this paper, we also use the recent result of Brzuska et al. [5], who proved that for mutually-authenticated *TLS-like* protocols (including TLS), deriving an export key from the master secret via a pseudorandom function yields a session key that is indistinguishable from random. In addition, a recent result also proved the AKE security for TLS 1.3 keys (thus implying 2-ACCE security); though this result holds for an earlier draft of TLS 1.3, we still assume TLS 1.3 to have this property [6].

4. ACCE with 3 parties: 3(S)ACCE

The attacks presented in Section 2 show that proxied TLS is difficult to get right, even if the individual channels

7. The fact that this message is encrypted in TLS 1.3 does not matter in our analysis, since the reduction is such that the right flag is used within.

are 2-ACCE secure. Furthermore, we saw that generalizing 2-party security to 3 parties requires care. We now present 3(S)ACCE, our security definition for proxied handshakes.

An overview. Our framework captures several types of architectures in a generic interaction model. We consider a PKI in which middleware and servers have registered credentials, but clients do not. Any secure channel the client establishes is only *unilaterally* authenticated. Middleware receive credentials for content-delivery upon *registering* with the content owner. This models both architectures in which the middleware impersonates the server for content-delivery (*e.g.*, CDNs) and those in which it uses its own credentials. Our notions of *partnering* and *freshness* capture both the designs in which the middleware can independently compute the session keys (as in CDNs) and those in which the middleware needs the server's help for some computations (like in Keyless SSL).

Apart from extending classical 2-ACCE entity authentication and channel security, we require two additional properties: accountability and content soundness.

Accountability. In CDN-ing over TLS, the middleware impersonates servers to clients that are oblivious of this fact. To protect against malicious or compromised middleboxes, the server must have the power to detect and de-authorize such middleboxes. Accountability requires that the server should be able to compute the channel keys used by the client and the middlebox. With this key, the server can audit the behaviour of the middlebox and take action against it, thus becoming *accountable* for any damage to the client.

Content soundness. Servers may authorize CDN middleboxes to deliver some of their contents. Content soundness requires that no malicious middlebox may serve content that it is *not* authorized to deliver. Typically, servers can achieve this by distributing content within separate subdomains with their own certificates, and authorizing middleboxes to serve content for specific subdomains. This demands a more extensive certificate infrastructure at the server. We explicitly note that we do not take into consideration websites with "public" content, *i.e.*, in our model, a middleware can only obtain content from an issuing server. In that sense, we also rule out (most) phishing attacks.

4.1. Attributes and Partnering

We extend the 2-ACCE setting to three parties, by adding a set $\mathcal{MW}$ of *middleware* entities to the existing sets of clients and servers. We instantiate parties as before, but extend the model to include new attributes as follows.

Names and certificates. In 2-ACCE infrastructures, if one party authenticates by means of, *e.g.*, a certificate, then that certificate will point to the right partner (unless an impersonation has occurred). In CDNs and Keyless SSL, however, the infrastructure *allows* entity impersonation to some extent. To capture the fact that one party may effectively compute keys based on another party's certificate, we no longer associate pid values with parties, but with *names* (for instance the common name –CN– entry in X.509 certificates). We assume that each certificate has a unique name associated to it. Since we still consider only unilaterally-authenticated handshakes for clients, we continue to use the generic label "Client" to indicate partnering clients.

We thus define, for each 3(S)ACCE party P_i, two new attributes: a *name* $P_i.\mathsf{name}$ and a set of certificates the party may use, denoted as $P_i.\mathsf{CertSet}$.

Pre-channel keys. In some proxying architectures, the middleware can only compute channel keys for middleware-client sessions with a server's assistance. In Keyless SSL, the server must compute any values requiring the secret key associated with the server's certified public key. We model this as follows. We assume that server instances, and middleware instances *partnered with server instances* may compute a local value, which allows the middlebox to compute the channel keys ck for a session it runs *with the client*. We call this value a *pre-channel key* and denote it as a new attribute $\pi_j^n.\mathsf{pck}$. For example, for Keyless SSL running in RSA mode, $\pi.\mathsf{pck} = \mathsf{pmk}$, whereas for the DHE mode, $\pi.\mathsf{pck} = (p, g, \mathsf{KE}_S, \mathsf{Cert}_S, \mathsf{PSign})$ (see Figure 2).

More formally, let PCK be the set of all pre-channel keys, Π the set of all possible transcripts τ, and CK the set of all channel keys. We are interested in protocols for which a function $\varphi : \ PCK \times \Pi \rightarrow CK$, taking as input a pre-channel key pck and a (handshake) transcript τ and outputting a channel key ck, can be defined. We require that $\forall pck, pck' \in PCK$, $\forall \pi, \pi' \in \Pi$ we have that if $\varphi(pck, \pi)$=$\varphi(pck', \pi')$, then π=π' (*i.e.*, that the projection of φ on Π be injective).

The notion of pre-channel keys is crucial in identifying *all* partnering instances of a given instance. Whenever the middleware can compute all session keys independently of the server, the classical 2-ACCE instance partnering holds. However, when the server needs to aid the middlebox and the pre-channel key is required, four instances are partnered instead of two.

Owned and proxied content. Another important aspect of proxying TLS connections is related to *content*. We associate to each server $P_j \in \mathcal{S}$ the content it owns, denoted Ω^j (which the adversary could choose). We assume that a content consists of unitary elements $\omega_i \in \Omega^j$ (*e.g.*, subdomains or single web-pages). Recall that we do not consider public contents, *i.e.*, the attacker may only retrieve a content by demanding it from a server which owns it. We assume that each handshake is meant to deliver a single content unit.

Middleware can receive (and then deliver) server-content in two steps. First the middlebox *registers* with a server for a given content, receiving credentials that identify it as a legitimate distributer of that content. Secondly, the middleware must execute a handshake with the correct server, then – upon succcessful mutual authentication and verification of registration for the content – the latter is delivered. We restrict the delivery to one content per handshake, sent as a response to a "$\mathsf{Contentrequest}, \omega$" message from the middleware.

We assume that clients can verify whether a content they receive in the record layer matches the certificate they received during the handshake[8].

Party Attributes. We require the following additional attributes for our formalization, for each *party* P_i:

- the **party's name** $P_i.\mathsf{name}$, linked to (possibly multiple) certificates. This value is unique per middleware or server, and is set to the label "Client" for all clients.
- an **indexed set** $P_i.\mathsf{CertSet}$ **of certificates** held locally by P_i, for which P_i stores the associated public keys that are certified, and possibly also the corresponding secret keys. This is only true for middleware and servers; for clients, the certificate set is empty.
- an **indexed set** $P_i.\mathsf{PKSet}$ **of public keys**, containing certified public keys held locally by P_i (either a middleware or a server), such that the k-th public key of that party, $P_i.\mathsf{Cert}_k$, is certified in the k-th certificate. This set is empty for all clients.
- an **indexed set** $P_i.\mathsf{SKSet}$ **of secret keys** held locally by P_i, defined analogously as $P_i.\mathsf{PKSet}$. For any secret keys that P_i does not have, P_i stores $\perp$ instead.
- a **set** $P_i.\mathsf{Contents}$ **of contents** to which P_i has access. For a server P_j, we denote its content by Ω^j. For middlewares P_k, we denote its entire registered content with server P_j by Ω^k_j. For clients, this set is empty.
- a **hashtable** $P_j.\mathsf{Contracts}$ held locally by servers P_j, for which the entries are of the form (P_k, Ω^k_j) with $P_k \in \mathcal{MW}$ and Ω^k_j is the subcontent P_k registered with P_j. This table is used by the server so that it forwards *only* registered contents to appropriate middleware.
- a **list of instances** $P_i.\mathsf{Instances}$, to keep track of all existing instances of this party.

Party-Instance Attributes. We proceed to extend the attribute-set of each *party-instance* π^m_i as follows:

- the **instance parameters** $\pi^m_i.\mathsf{par}$ consisting of: a certificate, the corresponding public key, and the matching private key from the sets $P_i.\mathsf{CertSet}, P_i.\mathsf{PKSet}$, $P_i.\mathsf{SKSet}$. Clients have no such values; servers always have non-$\perp$ values; and middleware may have the private key set to $\perp$.
- the **partner identifier** $\pi^m_i.\mathsf{pid}$, which will be set to a party *name* (rather than a party).
- a **pre-channel key** $\pi^m_i.\mathsf{pck}$ such that $\pi^m_i.\mathsf{pck} \in PCK$, set to $\perp$ for clients and for middleware instances whose partner identifier is set to "Client". Only *accepting* instances of servers or middleware partnering a server have a non-$\perp$ pre-channel key.
- a **record-layer transcript** $\pi^m_i.\mathsf{rec}\tau$, which is non-$\perp$ only for accepting instances that have already computed a channel key $\pi^m_i.\mathsf{ck}$. The record-layer transcript stores plaintext messages that a party encrypts and sends to its partner, or that the party decrypted from its partner.

8. This is modelled by means of an additional function with a Boolean output, which we describe in more detail in our full version.

4.2. Partnering and Freshness

Beyond simply tracing partners based on the session identifier `sid`, as in 2-ACCE, the design of Keyless SSL induces a dependency between a client-middleware session and an associated middleware-server session. In the latter, the server aids the middlebox to compute its client-middlebox channel key. The two sessions (and four instances) contain information about each other, and thus should be partnered.

We define 3(S)ACCE partnering for a given instance π^m_i of a party P_i in terms of two sets. The first of these, denoted $\pi^m_i.\mathsf{PSet}$, stores the *parties* that are partnered with π^m_i (*e.g.*, in Keyless SSL, a client, a middleware, and a server). The second set, denoted $\pi^m_i.\mathsf{InstSet}$, stores *instances* partnered wtih π^m_i. For ease of notation, both sets include the input party P_i and instance π^m_i. The *partner-instances* $\pi^m_i.\mathsf{InstSet}$ and *partner-parties* $\pi^m_i.\mathsf{PSet}$ *of an instance* π^m_i are defined constructively by Algorithm 1.

Algorithm 1 covers all the possible configurations of proxied handshakes: client–server, client–middleware, client–middleware–server, middleware–server. If middleboxes impersonate servers, as in CDNs, we denote this by $MW(S)$. If the middleware uses its own credentials, we use $MW(MW)$. We assume that the middleware always does the latter when interacting with the server, and do not specify this explicitly. To give the reader an intuition, we summarize all the possibilities in Figure 3. The rows are designed such that, for each entry, the partnering is defined from the viewpoint of one fixed party-instance involved in the link (this party is underlined).

Say we want to determine the partnering for a specific *client* instance π^m_i. This instance can be in one of the following communication scenarios: (1) speaking to a server in a direct "C–S" link; (2) speaking to a middleware that authenticates as itself, *i.e.*, a "C – MW(MW)" link; (3) speaking to a middleware that impersonates a server, but computes keys independently: "C – MW(S)" ; (4). speaking to a middleware impersonating a server, needing the latter's assistance, as in Keyless SSL:"C – MW – S". These are the first four rows in Figure 3.

To show how Algorithm 1 operates, take as an example row 3. As discussed, P_i, is a client, and its partner identifier will be the name of some party $P_j \in \mathcal{S}$. Thus, the algorithm enters the `IF` on line 2, but not the `IF` on line 3 (essentially demanding that there be "matching conversation" between π^m_i and a server instance). We enter the `ELSE` on line 14. Via line 15, we find the middleware that is *actually* partnering P_i, which is a middleware P_k, for an instance π^ℓ_k. Since the middleware does not need the server's aid to authenticate to the client, the pre-channel keys will not be used; this makes us skip line 17, and we output the partnering sets in line 37.

Now analyse the situation in row 4. The algorithm behaves as above, but now we do enter the `IF` on line 17. At that point, we find the matching middleware-server instances which will compute the pre-channel key for the session between π^m_i, π^ℓ_k. These are: an instance of that same

middleware P_k and an instance of the server who is the purported partner of π_i^m. In line 18 we add that party to the partnering instance-set of π_i^m, and these values are returned in line 37.

The other cases follow similarly.

	Actual Link	Partnering by Alg.1	Partner-sets forming Relevant Alg.1 lines
1.	$\underline{C} - S$	$\pi_i^m.\mathsf{PSet} = \{P_i \in C, P_j \in S\}$ $\pi_i^m.\mathsf{InstSet} = \{\pi_i^m, \pi_j^n\}$	4; 13
2.	$\underline{C} - MW(MW)$	$\pi_i^m.\mathsf{PSet} = \{P_i \in C, P_j \in MW\}$ $\pi_i^m.\mathsf{InstSet} = \{\pi_i^m, \pi_j^n\}$	4; 13
3.	$\underline{C} - MW(S)$	$\pi_i^m.\mathsf{PSet} = \{P_i \in C, P_k \in MW, P_j \in S\}$ $\pi_i^m.\mathsf{InstSet} = \{\pi_i^m, \pi_k^l\}$	16; 37
4.	$\underline{C} - MW - S$	$\pi_i^m.\mathsf{PSet} = \{P_i \in C, P_k \in MW, P_j \in S\}$ $\pi_i^m.\mathsf{InstSet} = \{\pi_i^m, \pi_k^l, \pi_j^n\}$	16, 18; 37
5.	$\underline{MW} - S$	$\pi_i^m.\mathsf{PSet} = \{P_i \in MW, P_j \in S\}$ $\pi_i^m.\mathsf{InstSet} = \{\pi_i^m, \pi_j^n\}$	4; 13
6.	$\underline{MW(MW)} - C$	$\pi_i^m.\mathsf{PSet} = \{P_j \in C, P_i \in MW\}$ $\pi_i^m.\mathsf{InstSet} = \{\pi_i^m, \pi_j^n\}$	23; 37
7.	$\underline{MW(S)} - C$	$\pi_i^m.\mathsf{PSet} = \{P_j \in C, P_i \in MW, P_k \in S\}$ $\pi_i^m.\mathsf{InstSet} = \{\pi_i^m, \pi_j^n\}$	23, 28; 37
8.	$S - \underline{MW} - C$	$\pi_i^m.\mathsf{PSet} = \{P_k \in C, P_i \in MW, P_j \in S\}$ $\pi_i^m.\mathsf{InstSet} = \{\pi_i^m, \pi_i^p, \pi_k^l\}$	7; 13
9.	$C - \underline{MW} - S$	$\pi_i^m.\mathsf{PSet} = \{P_j \in C, P_i \in MW, P_k \in S\}$ $\pi_i^m.\mathsf{InstSet} = \{\pi_i^m, \pi_j^n, \pi_i^p, \pi_k^l\}$	23, 28, 31; 32
10.	$\underline{S} - C$	$\pi_i^m.\mathsf{PSet} = \{P_j \in C, P_i \in S\}$ $\pi_i^m.\mathsf{InstSet} = \{\pi_i^m, \pi_j^n\}$	23; 25
11.	$\underline{S} - MW$	$\pi_i^m.\mathsf{PSet} = \{P_j \in MW, P_i \in S\}$ $\pi_i^m.\mathsf{InstSet} = \{\pi_i^m, \pi_j^n\}$	4; 13
12.	$\underline{S} - MW - C$	$\pi_i^m.\mathsf{PSet} = \{P_k \in C, P_j \in MW, P_i \in S\}$ $\pi_i^m.\mathsf{InstSet} = \{\pi_i^m, \pi_j^n, \pi_j^p, \pi_k^l\}$	4, 11; 13

Figure 3: Algorithm 1 Coverage of 3-Party Communication Settings

Session freshness in 3(S)ACCE**:** We adapt the 2-ACCE notion of freshness to 3(S)ACCE as follows.

3(S)ACCE **Freshness.** An instance π_i^m of P_i is *fresh* with intended partner P_j if the following conditions hold:

- $\pi_i^m.\alpha = 1$ with $\pi_i^m.\mathsf{pid} = P_j.\mathsf{name}$.
- All parties in $\pi_i^m.\mathsf{PSet}$ are uncorrupted. Note that this includes P_i itself.
- No Reveal query was made to any instance in $\pi_i^m.\mathsf{InstSet}$, which includes π_i^m itself.

Correctness in 3(S)ACCE**.** We extend the definition of 2-ACCE correctness in terms of the partnering algorithm above. We demand that, for any instance π_i^m ending in an accepting state with partnering instance-set $\pi_i^m.\mathsf{InstSet}$ and partnering party-set $\pi_i^m.\mathsf{PSet}$, the following conditions hold.

1) If $|\pi_i^m.\mathsf{InstSet}| = 2$, then both instances in $\pi_i^m.\mathsf{InstSet}$ compute the same channel key ck.
2) Consider the case of $|\pi_i^m.\mathsf{InstSet}| = 4$. Let $\pi_j^n, \pi_k^\ell \in \pi_i^m.\mathsf{InstSet}$ for $P_i, P_j \in \pi_i^m.\mathsf{PSet}$, such that π_i^m, π_j^n share a session identifier. Then, the following holds:
   a). Any instances in $\pi_i^m.\mathsf{InstSet}$ sharing session ID compute the same channel key:
   - π_i^m and π_j^n compute the same channel key x, *i.e.*, $\pi_j^n.\mathsf{ck} = \pi_i^m.\mathsf{ck}$ and $x{:=}\pi_i^m.\mathsf{ck}$;
   - the other two instances in $\pi_i^m.\mathsf{InstSet}$ (*i.e.*, π_k^ℓ and $\pi_i^m.\mathsf{InstSet} \setminus \{\pi_i^m, \pi_j^n, \pi_k^\ell\}$) compute the same channel key x';

   b). Moreover, the pre-channel key computed in the middleware-server session must be consistent with that of the client-middleware channel key. Thus,
   - If P_i or P_j is a client, then $x = \varphi(\pi_k^\ell.\mathsf{pck}, \pi_i^m.\tau)$.
   - If neither P_i nor P_j is a client, it holds that $x' = \varphi(\pi_i^m.\mathsf{pck}, \pi_k^\ell.\tau)$.

4.3. Adversarial Interactions

In Section 4.4 we define four security notions, which together constitute the definition of 3(S)ACCE-security. As in 2-ACCE security, each notion is defined in terms of a security game. The adversary has access to oracles that are adapted from standard 2-ACCE. We also need an additional party-registration oracle. Due to lack of space, we simplify the descriptions of the oracle somewhat, leaving a complete description to the full paper.

2-ACCE-like Oracles. In the 3(S)ACCE games, most of the 2-ACCE oracles remain unchanged (we review them below, underlining any different input they might take), and we modify the NewSession oracle as follows:

- $\mathsf{NewSession}(P_i, \rho, \underline{\mathsf{real.pid}}, \underline{\mathsf{int.pid}}, \underline{\mathsf{content}})$: This query creates a new session π_i^m executed by party P_i with the role ρ. The input values real.pid and int.pid indicate the real partner of π_i^m, and resp. its intended partner. The value content indicates a content for which the session will be run. The oracle creates π_i^m, sets it in running state, then sets $\pi_i^m.\alpha := \perp$ and $\pi_i^m.\rho := \rho$. We continue as follows:
  - If $P_i \in \mathcal{C}$, then real.pid is the true partner of π_i^m and int.pid is the intended partner, the owner of content content. The oracle checks that real.pid, int.pid contain at most one middleware and one server, and that if real.pid is a server, then real.pid= int.pid. If all this verifies, π_i^m is instantiated with $\pi_i^m.\mathsf{par} := (\perp, \perp, \perp)$ (it will have no certificates, no public, or secret keys). Furthermore if P_i is the initiator, the oracle outputs the protocol's first message.
  - For $P_i \in \mathcal{MW}$, int.pid indicates whether P_i acts as itself (with its own certificate) or it legitimately impersonates a server it registered with. If $\mathsf{int.pid} = P_i$, then we set $\pi_i^m.\mathsf{par} := (\mathsf{Cert}_i, \mathsf{pk}_i, \mathsf{sk}_i)$ (its own credentials); else, if $\mathsf{int.pid} = P_j \in \mathcal{S}$, the owner of content, then $\mathsf{real.pid} \in \mathcal{C}$ and $\pi_i^m.\mathsf{par}$ contains the parameters output by the RegParty query for P_i and P_j. If the middleware executes the protocol with a server, then $\mathsf{content} = \perp$. If P_i is the initiator, then the first protocol message is also output.
  - If $P_i \in \mathcal{S}$, then the $\mathsf{real.pid} = \mathsf{int.pid} \in \{\mathcal{C}, MW\}$. Then $\pi_i^m.\mathsf{par}$ consists of the server's own credentials. If the server is the initiator, then the first message is generated.
- $\mathsf{Send}(\pi_i^m, M)$: This is the classical 2-ACCE query, through which the adversary can send a message M to π_i^m, receiving a response M' or a fixed error message $\perp$ (if the instance does not exist).
- $\mathsf{Corrupt}(P_i)$: This query returns all the secret keys stored by P_i's.

Algorithm 1 Partnering in 3-party ACCE

Require: The input is a party instance π_i^m (ending in accepting state).
Ensure: The output are two sets $\pi_i^m.\mathsf{PSet}$ and $\pi_i^m.\mathsf{InstSet}$.

Set $\pi_i^m.\mathsf{PSet} := \{P_i\}$ and $\pi_i^m.\mathsf{InstSet} := \{\pi_i^m\}$.
if $(\pi_i^m.\mathsf{pid} = P_j.\mathsf{name} \mid \text{for some } P_j \in \{\mathcal{MW}, \mathcal{S}\})$ **then**
  if ($\exists$ unique π_j^n s. that $\pi_i^m.\mathsf{sid} = \pi_j^n.\mathsf{sid}$ and $\pi_j^n.\mathsf{pid} = P_i.\mathsf{name}$) **then**
    Do $\pi_i^m.\mathsf{PSet} \leftarrow \pi_i^m.\mathsf{PSet} \cup \{P_j\}$ and $\pi_i^m.\mathsf{InstSet} \leftarrow \pi_i^m.\mathsf{InstSet} \cup \{\pi_j^n\}$
    if ($P_i \in \mathcal{MW}$ and $\pi_i^m.\mathsf{pck} \neq \perp$) **then**
      Find $P_k \in \mathcal{C}$ s.that $\exists \pi_i^p, \pi_k^\ell$ with $\pi_k^\ell.\mathsf{sid} = \pi_i^p.\mathsf{sid}$ and $\pi_k^\ell.\mathsf{ck} = \pi_i^p.\mathsf{ck} = \varphi(\pi_i^m.\mathsf{pck}, \pi_k^\ell.\tau)$
      Do $\pi_i^m.\mathsf{PSet} \leftarrow \pi_i^m.\mathsf{PSet} \cup \{P_k\}$ and $\pi_i^m.\mathsf{InstSet} \leftarrow \pi_i^m.\mathsf{InstSet} \cup \{\pi_i^p, \pi_k^\ell\}$
    end if
    if ($P_j \in \mathcal{MW}$ and $\pi_j^n.\mathsf{pck} \neq \perp$) **then**
      Find $P_k \in \mathcal{C}$ s.t. $\exists \pi_j^p, \pi_k^\ell$ with $\pi_k^\ell.\mathsf{sid} = \pi_j^p.\mathsf{sid}$ and $\pi_k^\ell.\mathsf{ck} = \pi_j^p.\mathsf{ck} = \varphi(\pi_j^n.\mathsf{pck}, \pi_k^\ell.\tau)$
      Do $\pi_i^m.\mathsf{PSet} \leftarrow \pi_i^m.\mathsf{PSet} \cup \{P_k\}$ and $\pi_i^m.\mathsf{InstSet} \leftarrow \pi_i^m.\mathsf{InstSet} \cup \{\pi_j^p, \pi_k^\ell\}$
    end if
    Return $\pi_i^m.\mathsf{PSet}$ and $\pi_i^m.\mathsf{InstSet}$.
  else
    Find unique P_k, π_k^ℓ s.that $\pi_k^\ell.\mathsf{sid} = \pi_i^m.\mathsf{sid}$
    Do $\pi_i^m.\mathsf{PSet} \leftarrow \pi_i^m.\mathsf{PSet} \cup \{P_j, P_k\}$ and $\pi_i^m.\mathsf{InstSet} \leftarrow \pi_i^m.\mathsf{InstSet} \cup \{\pi_k^\ell\}$
    if $\exists \pi_k^p, \pi_j^n$ s.t. $\pi_k^p.\mathsf{sid} = \pi_j^n.\mathsf{sid}$ and $\pi_i^m.\mathsf{ck} = \pi_k^\ell.\mathsf{ck} = \varphi(\pi_j^n.\mathsf{pck}, \pi_i^m.\tau)$ **then**
      Do $\pi_i^m.\mathsf{InstSet} \leftarrow \pi_i^m.\mathsf{InstSet} \cup \{\pi_k^p, \pi_j^n\}$
    end if
  end if
else
  Find unique P_j, π_j^n s.that $\pi_i^m.\mathsf{sid} = \pi_j^n.\mathsf{sid}$
  Do $\pi_i^m.\mathsf{PSet} \leftarrow \pi_i^m.\mathsf{PSet} \cup \{P_j\}$ and $\pi_i^m.\mathsf{InstSet} \leftarrow \pi_i^m.\mathsf{InstSet} \cup \{\pi_j^n\}$
  if $P_i \in \mathcal{S}$ **then**
    Return $\pi_i^m.\mathsf{PSet}, \pi_i^m.\mathsf{InstSet}$
  else
    if $\pi_j^n.\mathsf{pid} = P_k.\mathsf{name}$ for $P_k \in \mathcal{S}$ **then**
      Do $\pi_i^m.\mathsf{PSet} \leftarrow \pi_i^m.\mathsf{PSet} \cup \{P_k\}$
      if $\exists$ π_k^ℓ of party P_k s.t. $\pi_j^n.\mathsf{ck} = \varphi(\pi_k^\ell.\mathsf{pck}, \pi_j^n.\tau)$ **then**
        Find unique π_i^p such that $\pi_i^p.\mathsf{sid} = \pi_k^\ell.\mathsf{sid}$
        Do $\pi_i^m.\mathsf{InstSet} \leftarrow \pi_i^m.\mathsf{InstSet} \cup \{\pi_i^p, \pi_k^\ell\}$
        Return $\pi_i^m.\mathsf{PSet}, \pi_i^m.\mathsf{InstSet}$
      end if
    end if
  end if
end if
Return $\pi_i^m.\mathsf{PSet}, \pi_i^m.\mathsf{InstSet}$

- $\mathsf{Reveal}(\pi_i^m)$: This query returns the channel key $\pi_i^m.\mathsf{ck}$ of an instance π_i^m ending in an accepting state. The revealed bit of the session is set to 1.
- $\mathsf{Enc}(\pi_i^m, l, M_0, M_1, H)$: This is a left-or-right oracle, encrypting a bit M_b where $b = \pi_i^m.\mathsf{b}$ with header H, for a ciphertext length l, with the channel keys of an accepting instance π_i^m.
- $\mathsf{Dec}(\pi_i^m, C, H)$: This query only decrypts the input ciphertext C for a given header H on behalf of an accepting instance $\pi_i^m.\mathsf{ck}$ if, and only if, the ciphertext was generated by the adversary and if $\pi_i^m.\mathsf{b}$.

New Oracles. We also require a new party-registration oracle, as described below.

- $\mathsf{RegParty}(P_i, \Omega, \mathsf{dest})$: taking as input a party P_i, a content Ω (possibly set to a special symbol $\perp$), and a so-called *destination* $\mathsf{dest} \in \{\text{“}\texttt{MW}''\text{, “}\texttt{Server}''\} \cup \mathcal{S}$. This query works in three modes.
  - **The server mode**, with input $(P_i, \Omega, \text{“}\texttt{Server}''\text{)}$. This query registers P_i as a server (unless this has been done before), outputting, for each subcontent $\omega \in \Omega$, a tuple consisting of a secret key, a public key, and a certificate for that public key on the server's name. The content Ω is added to the server's already-registered content. The adversary receives the certificates and public keys.
  - **The middleware mode**, with input $(P_i, \perp, \text{“}\texttt{MW}''\text{)}$. The query registers P_i as middleware, outputting a single secret key, public key, and certificate for that public key associated with P_i's name (the latter two are also output to the adversary).
  - **The contract mode**, with input (P_i, Ω, P_j) and $P_i \in \mathcal{MW}$, $P_j \in \mathcal{S}$, both registered parties such that $\Omega \subseteq \Omega^j$ (the requested content is a subset of the server's content). This oracle registers locally in the server's hashtable $P_j.\mathsf{Contracts}$ the entry P_j, Ω, and it adds Ω to the middleware's content Ω_j^i. Then a set of secret keys, public keys, and certificates are given to P_i (one tuple of keys with the certificate per subcontent). In some cases, the secret key may be set to $\perp$. All the public information is also given to the adversary.

4.4. 3(S)ACCE

3(S)ACCE-security consists of four properties: entity authentication, channel security, accountability, and content soundness.

For each game, the PPT adversary $\mathcal{A}$ plays against a challenger, interacting with various party instances sequentially or concurrently by the oracles above. We quantify attacks in terms of the number N_P of parties in the system (*i.e.*, the number of clients N_C, the number of servers N_S, and the number of middleboxes N_{MW}), the number q_n of NewSession queries that the adversary makes, and the number t of the total of queries made by the $\mathcal{A}$. The time-complexity of the adversary is determined by t and q_n.

Entity Authentication (EA). In the *entity authentication game*, the adversary $\mathcal{A}$ can query the new oracle RegParty and traditional 2-ACCE oracles. Finally, $\mathcal{A}$ ends the game by outputting a special string "Finished" to its challenger. The adversary *wins* the EA game if there exists a party instance π_i^m *maliciously accepting* a partner $P_j \in \{\mathcal{S}, \mathcal{MW}\}$, according to the following definition.

Winning condition – EA game. An instance π_i^m of some party P_i is said to *maliciously accept* with partner $P_j \in \{\mathcal{S}, \mathcal{MW}\}$ if the following holds:

- $\pi_i^m.\alpha = 1$ with $\pi_i^m.\mathsf{pid} = P_j.\mathsf{name} \neq$ "Client";
- No party in $\pi_i^m.\mathsf{PSet}$ is corrupted, no party in $\pi_i^m.\mathsf{InstSet}$ was queried in Reveal queries;
- There exists no unique $\pi_j^n \in P_j.\mathsf{Instances}$ such that $\pi_j^n.\mathsf{sid} = \pi_i^m.\mathsf{sid}$;
- If $P_i \in \mathcal{C}$, there exists no party $P_k \in \mathcal{MW}$ such that: $\mathsf{RegParty}(P_k, \cdot, P_j)$ has been queried, and there exists an instance $\pi_k^\ell \in \pi_i^m.\mathsf{InstSet}$.

The adversary's advantage, denoted $\mathsf{Adv}_\Pi^{\mathsf{EA}}(\mathcal{A})$, is defined as its winning probability *i.e.*:

$$\mathsf{Adv}_\Pi^{\mathsf{EA}}(\mathcal{A}) := \mathbb{P}[\mathcal{A} \text{ wins the EA game}],$$

where the probability is taken over the random coins of all the N_P parties in the system.

Channel Security (CS). In the *channel security game*, the adversary $\mathcal{A}$ can use all the oracles (including RegParty) adaptively, and finally outputs a tuple consisting of a fresh party instance π_i^j (in the sense of Definition 4.2) and a bit b'. The winning condition is defined below:

Winning Conditions – CS Game. An adversary $\mathcal{A}$ *breaks the channel security* of a 3(S)ACCE protocol, if it terminates the channel security game with a tuple (π_i^j, b') such that:

- π_i^m is fresh with partner P_j;
- $\pi_i^m.\mathsf{b} = b'$.

The advantage of the adversary $\mathcal{A}$ is defined as follows:

$$\mathsf{Adv}_\Pi^{\mathsf{SC}}(\mathcal{A}) := \left|\mathbb{P}[\mathcal{A} \text{ wins the SC game}] - \frac{1}{2}\right|,$$

where the probability is taken over the random coins of all the N_P parties in the system.

Accountability (Acc). In the *accountability* game the adversary may arbitrarily use all the oracles in the previous section, finally halting by outputting a "Finished" string to its challenger. We say $\mathcal{A}$ *wins* if there exists an instance π_i^m of a client P_i such that the following condition applies.

Winning Conditions – Acc. An adversary $\mathcal{A}$ *breaks the accountability* for instance π_i^m of $P_i \in \mathcal{C}$, if the following holds simultaneously:

(a) $\pi_i^m.\alpha = 1$ such that $\pi_i^m.\mathsf{pid} = P_j.\mathsf{name}$ for an *uncorrupted* $P_j \in \mathcal{S}$;
(b) There exists no instance $\pi_j^n \in P_j.\mathsf{Instances}$ such that $\pi_j^n.\mathsf{ck} = \pi_i^m.\mathsf{ck}$;
(c) There exists no probabilistic algorithm Sim (polynomial in the security parameter) which given the view of P_j (namely all instances $\pi_j^n \in P_j.\mathsf{Instances}$ with all their attributes), outputs $\pi_i^m.\mathsf{ck}$.

The adversary's advantage is defined as its winning probability, *i.e.*:

$$\mathsf{Adv}_\Pi^{\mathsf{Acc}}(\mathcal{A}) := \mathbb{P}[\mathcal{A} \text{ wins the Acc game}],$$

where the probability is taken over the random coins of all the N_P parties in the system.

Content Soundness (CSound). In the *content soundness* game, the adversary $\mathcal{A}$ queries the oracles arbitrarily, and ends the game with a "Finished" message. The adversary *wins* if there exist: a client instance π_i^m and a content ω such that $\mathcal{A}$ breaks soundness in the definition below.

Winning Conditions – CSound. An adversary $\mathcal{A}$ breaks soundness of an arbitrarily fixed context ω an 3(S)ACCE protocol if there exists a client-instance $\pi_i^m \in P_i.\mathsf{Instances}$ for $P_i \in \mathcal{C}$ and $\pi_i^m.\alpha = 1$, and there exists no server-instance π_u^ℓ of a party $P_u \in \mathcal{S}$ such that: $\pi_i^m.\mathsf{pid} = P_u.\mathsf{name}$, $\omega \in \Omega^u$, and $\pi_i^m.\mathsf{sid} = \pi_u^\ell.\mathsf{sid}$, and the following conditions simultaneously hold:

- $\omega \in \pi_i^m.\mathsf{rec}\tau$ and ω received by π_i^m;
- there exists no instance π_k^p of a party P_k such that $\pi_i^m.\mathsf{sid} = \pi_k^p.\mathsf{sid}$ and, by defining $\Omega_k := \{\hat{\omega} : \mathsf{RegParty}(P_k, \Omega, \cdot)$ queried and $\hat{\omega} \in \Omega\}$, it holds that $\omega \in \Omega_k$;
- any party P_x such that $\mathsf{RegParty}(P_x, \Omega, \cdot)$ was queried, with $\omega \in \Omega$, P_x is uncorrupted.
- furthermore, no party P_y such that $\mathsf{RegParty}(P_y, \Omega,$ "Server") was queried, with $\omega \in \Omega$, is corrupted.

The adversary's advantage is defined as its winning probability, *i.e.*:

$$\mathsf{Adv}_\Pi^{\mathsf{CSound}}(\mathcal{A}) := \mathbb{P}[\mathcal{A} \text{ wins the CSound game}],$$

where the probability is taken over the random coins of all the N_P parties in the system.

5. Fixing Keyless SSL

Keyless SSL is not 3(S)ACCE-secure. The attack on TLS-RSA in Section 2 breaks our channel security definition, since a malicious middlebox MW' can decrypt connections to an honest middlebox MW. Similarly, a malicious middlebox can confuse a honest middlebox about the identity of the server, breaking entity authentication. Session resumption breaks accountability.

5.1. Achieving 3(S)ACCE-security for Keyless SSL

We introduce an 3(S)ACCE-secure variant of Keyless SSL called 3(S)ACCE-K-SSL, depicted in Figure 4. In the following description, we focus mainly on the differences between this variant and the original Keyless SSL protocol.

A more extensive PKI. In the original Keyless SSL architecture, whenever a middlebox registers with the server, for

any content, the server forwards the same certificate, which we denote as $\mathsf{Cert}_{MW,S}$. However, in order to ensure content soundness and entity authentication, we need servers to issue one public key (with a corresponding certificate) for each subcontent ω *and* for each middlebox authorized to deliver that subcontent. We denote a certificate associated with a server P_j which is used by a middlebox P_k for some content ω as $\mathsf{Cert}^{\omega}_{P_k,P_j}$.

The protocol description. The key new elements of our protocol (see Figure 4) are as follows.

An export key ek. We begin by describing the handshake between the middlebox and the server. In Keyless SSL this is a generic 2-ACCE-secure protocol; in our design we use a TLS handshake, and then compute an export key ek, which is computed as the result of a pseudorandom function G which is independent of the one used in the TLS handshake itself. We key this function with the master secret used in the TLS handshake between the middlebox and the server, and use it on input the nonces used in that session. For this type of protocol, Brzuska et al. [5] proved that ek is indistinguishable from random from any party other than the two protocol participants.

The key share KE_{MW}. In the traditional Keyless TLS-DHE architecture, the middlebox always generates its own key share KE_{MW} and sends a hash of that and other values to the appropriate server to certify. In our new design, it is the server that generates the key share. This is essential in order to achieve accountability: if the middleware were able to generate msk, it could use session resumption in a future session, thus winning the accountability game. Note that the server has already run a handshake with the middlebox at this point, and the server has verified the identity of its partner. The middlebox forwards the content ω requested by the client and the two session nonces used in the client-middlebox session; then the server generates the signature of the public key certified in $\mathsf{Cert}^{\omega}_{MW,S}$.

Computing the channel key. In DHE mode, the server now holds the private DH exponent to the key share KE_{MW}. After the middlebox receives (in its session with the client) the client's key share KE_C and the Finished message Fin_C (encrypted as in $\mathtt{AE}_{\mathsf{ck}}(\mathsf{Fin}_C)$), it transmits the entire session transcript τ to the server. The latter verifies: that the nonces in the transcript are those received earlier; that the certificate used by the middlebox in that transcript is the correct one, and finally, that the client's Finished message once decrypted at this server's end is as expected (note that the server can compute pmk, msk, and the channel key ck locally). Finally, the server also computes the Finished message Fin_S on behalf of the middlebox, which it encrypts and authenticates.

In TLS-RSA mode, the client chooses pmk and sends it to the middlebox, encrypted with the received certified public key. The middlebox forwards the entire session transcript up to, and including, the client's encrypted Finished message. The server checks that the certificate is compatible with the querying middlebox' authentication information (certificate). Then it obtains pmk by decrypting under the secret key corresponding to the certificate the middlebox forwarded. After computing msk and ck, the server verifies the validity of the client's Finished message. Upon successful verification, the server computes and sends the encrypted Finished message the client expects from the middlebox.

Blinding the key. In both modes, the server will first blind the computed channel key ck with the export key ek, thus sending the encrypted Finished message and $ek \oplus \mathsf{ck}$ to the middlebox, who will recover the value ck and use the Finished message. In particular, the blinding is necessary in the security proof in order to reduce the security of our variant to the 2-(S)ACCE security of TLS (we will need to simulate the encrypted messages so that they are consistent with what the adversary expects).

The security statement. We now state an informal security statement, referring the interested reader to the full version for the exact security bounds.We begin by noting that, following the results of Brzuska et al. [5], if G is a PRF that is independent from the key-derivating PRF used in TLS, then the keys ek computed by the middlebox and the server are indistinguishable from random. Furthermore, the TLS 1.2 protocol with unilateral authentication was proved to attain 2-SACCE security by Krawczyk et al. [10]; they also proved the same protocol was 2-ACCE-secure. For technical reasons, we also require a less-than-ideal assumption, namely, that P and P' are, essentially, the same protocol, one in the SACCE setting, the other, in the ACCE setting, as also requested in Section 3.

Theorem 1. Let Π be the 3(S)ACCE-K-SSL variant. We denote by P be the unilaterally-authenticated TLS 1.2 handshake, by P', the mutually-authenticated TLS 1.2 handhsake, and by Ψ, the transformation of P' to an AKE protocol by the computation of the export key ek. If the following conditions hold:

- If P is a 2-SACCE-secure protocol, P' is a 2-ACCE-protocol such that P and P' are together mEA-secure w.r.t. mEA-game (i.e., P and P' have entity authentication w.r.t. the adversary in the mEA-game, and they can be obtained from one another via the mode-flag flag defined in the mEA-game)[9], and $\Psi(P')$ yields pseudorandom keys;
- For TLS-DHE: if the hash function H is collision-resistant and the signature scheme used to generate PSign is unforgeable;
- For TLS-RSA: if P guarantees channel security;

Then Π guarantees 3(S)ACCE-security[10].

5.2. Efficiency vs. Security

Our 3(S)ACCE-K-SSL proposal has several disadvantages in terms of efficiency with respect to Keyless SSL.

9. Using mEA-security instead of the standard EA-security implies the following restriction. Instead of being free to choose two different 2-(S)ACCE protocols, we work in a setting where by picking one protocol P we also fix P' and vice versa.

10. In fact, we need fewer assumptions to prove accountability and content soundness, since these are defined from the point of view of the client only.

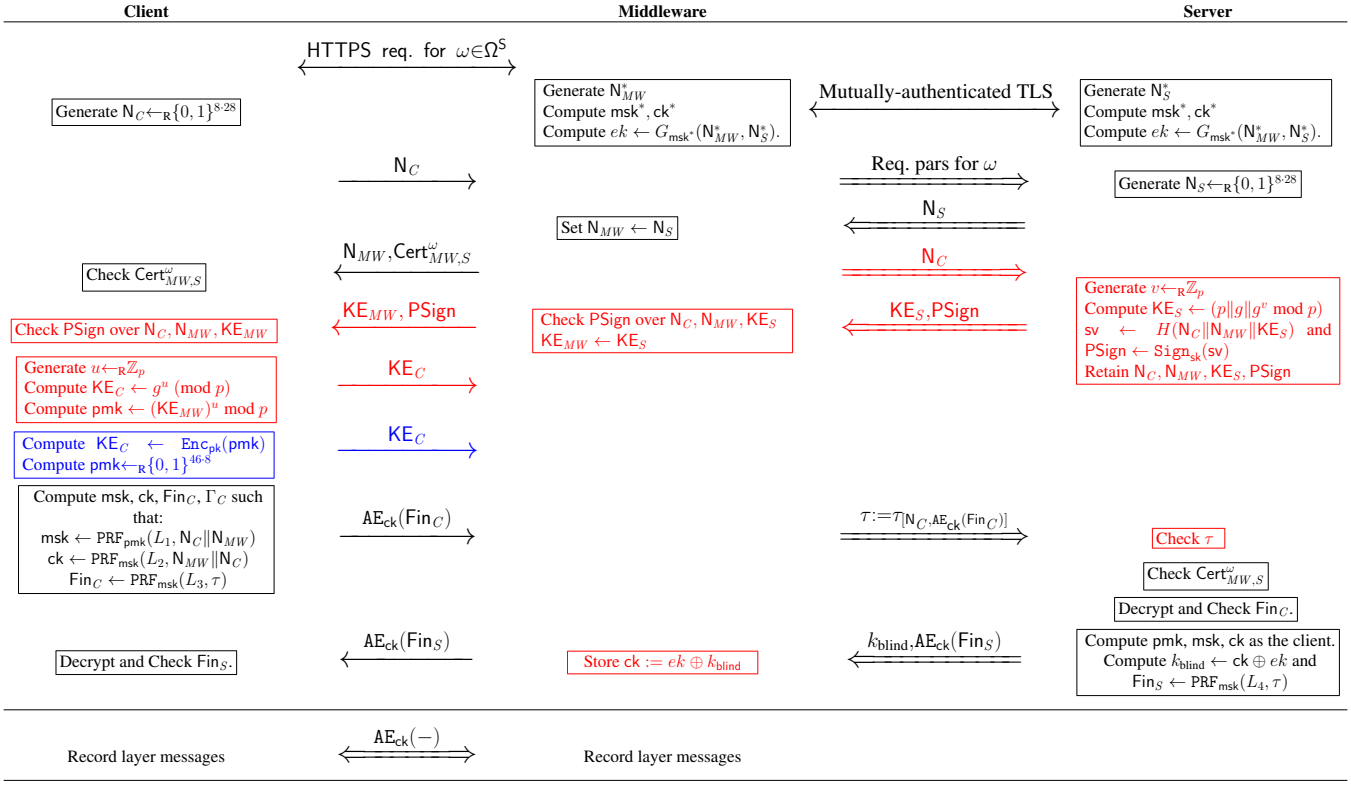

Figure 4: An 3(S)ACCE-secure variant of Keyless SSL. (blue+black) TLS-RSA; (red+black): TLS-DHE.

First, the server is involved more heavily in client-middleware handshakes. In fact, in 3(S)ACCE-K-SSL the middlebox is practically reduced to relaying the handshake between the client and the server. Most of our modifications are meant to achieve accountability. In particular, we must both *allow* the server to compute the master secret and channel keys, and at the same time *prevent* the middleware from calculating those values itself (otherwise the msk can be used for session resumption). In addition to computing msk, ck, the server must also verify the client Finished message, and generate and encrypt Fin_S on behalf of the middlebox.

Second, we disallow session resumption. An alternative is to relax the accountability definition requiring servers to only be able to compute the msk for each client-middleware session (and not directly the channel key ck). Using TLS 1.2 session resumption is compatible with this relaxation; however, we *cannot recommend* using it, since it weakens security against malicious middleboxes. We also discourage the use of TLS-RSA in Keyless SSL, as this mode is inherently not forward secure; although TLS-DHE involves more exchanges with the server, it offers a better protection against malicious middleware and corruptions.

Third, we need to compute the export key ek to blind the channel-key transmission. In the proofs, this allows us to simulate the record-layer transcript to an adversary, even when the reduction does not know ck. Although not computationally-heavy, this step does deviate from the standard TLS 1.2 handshake.

Fourth, we also require a very large certificate infrastructure, with one public key/certificate per middlebox, per content. A way to reduce that number is to ensure, on the server side, that no content can be delivered by more than one middlebox. This could, however, significantly decrease the efficiency of the CDN.

Our design represents one set of trade-offs that favors security over efficiency. Other designs may choose a different balance. For example, if we want to prevent just the attacks in Section 2 without attaining full 3(S)ACCE-security, smaller changes are probably adequate.

6. Keyless TLS 1.3

Surprisingly, using TLS 1.3 instead of 1.2 allows us to significantly reduce the PKI and the computational burden on server-side computation. Indeed, in this section we present our take on Keyless TLS 1.3 and outline its advantages and security guarantees.

A lighter PKI. In Keyless TLS 1.3 the middlebox is back to using servers as signature oracles, but with additional verifications. Since the server is generating the *nonce* and since the nonces and certificates are signed at each handshake, we reduce the infrastructure to one certificate (and public key) per content; this key can, however, be shared by several middleboxes. Moreover, if a server misbehaves and tries to

re-use the discrete logarithm of some value KE_{MW}, as long as the honest middleware generates a fresh key exchange value at every session, this is sufficient to guarantee security for new instances.

Keyless TLS 1.3. We present our protocol in Figure 5. Due to space restrictions, we do not detail all the computation required for the key-derivation of TLS 1.3, but mention that, in the absence of a pre-shared key PSK, the Diffie Hellman value g^{uv} mod p, the session nonces, and the intermediate transcript, including the auxiliary information aux which is provided by the client and the middlebox, are sufficient to allow both the server and the middlebox to go through the key scheduling, thus outputting all the necessary keys. An interesting development is that in TLS 1.3, the option to resume or not a session is reflected in the signature provided by the server; consequently, as long as the server never signs a message allowing for resumption, we can allow the middleware to compute all the session keys. This is not possible for TLS 1.2, in which the signature does not contain this data.

We list only the more significant of the session keys, listed in our protocol depiction. The handshake traffic keys HTK are used to authenticate and encrypt handshake-information. The client, and respectively server finished keys $\mathsf{CFK}, \mathsf{SFK}$ are used to compute the Finished messages. Finally, the traffic keys TK secure the record-layer messages. We note that all these keys are independent of each other, and they all depend on the full transcript.

Note that TLS 1.3 guarantees AKE security for the traffic keys [6], [9]. Moreover, if the AEAD primitive attains real-from-random indistinguishability for the plaintexts, then the composition of the two also yields the same security; this allows us to dispense with the export key ek. We let the middlebox generate its key share, but request that it sends the discrete logarithm of that value along, which enables the server to compute $g^{uv}(\text{mod } p)$, which allows this party to eventually compute all the session keys. After verifying that the discrete logarithm is correct, and ensuring that the middlebox will accept neither resumption, nor pre-shared keys (this is reflected in the variable aux), the server will generate a new nonce and compute a signature over a hash of the two nonces, the two key-exchange shares, and the auxiliary material. This signature enters the calculation of all the key material (as part of the transcript). Given the server's nonce and signature, the middlebox can compute the relevant keys, encrypt its certificate, the received signature, and the Finished message with the *handshake* traffic keys. The client verifies the material it has received from the middlebox and, if all verifications pass, it generates and sends its own encrypted Finished message.

The two parties will then use the *traffic* keys to secure record-layer transmissions.

Key Updates and Resumption. Whilst our definition of accountability only stretches as far as the *traffic keys*, TLS 1.3 introduces the concept of *updating* record-layer keys. Thus, the two partners may agree to update their old record-layer keys TK_m to TK_{m+1} by a transformation requiring only knowledge gained during the *handshake*. Therefore, the server not only has accountability for the first traffic key TK_0, but also for all future updates of that key.

Whereas updating keys poses no problems, session resumption still does. In the case of two-party TLS 1.3, session resumption seems so far to provide acceptable guarantees [6]; however, for the three-party case, it could prove more problematic and should not be adopted before being extensively studied.

Security statement. We give only an informal security statement here, and leave the fully formalized theorem, with the exact bounds, for the full version of this paper. The standard, unilaterally-authenticated TLS 1.3 handshake was proved to provide 2-S-AKE-security, and, for uncompromised session keys, the composition of TLS 1.3 with its chosen AEAD cipher guarantees that the MiM adversary cannot distinguish between ciphertexts generated on a real message, and those generated for a random message [6], [9]. The TLS 1.3 handshake with mutual authentication guarantees 2-AKE-security and the same authenticated encryption guarantee.

Theorem 2. Let Π be the Keyless TLS 1.3 protocol presented in this section. We denote by P the unilaterally-authenticated TLS 1.3 handshake, and by P', the mutually-authenticated one. If the following conditions hold:

- P guarantees 2-S-AKE-security and the security of the constructed channel (as described above);
  P' guarantees 2-AKE-security and channel security;
  P and P' can be obtained from one another via the mode-flag flag in the mEA-game;
  P and P' guarantee mEA-security;
- H is a collision-resistant hash function;
- the signature-scheme used to generate PSign is unforgeable;

then Π guarantees 3(S)ACCE-security.

7. In perspective: proxying over TLS

Content delivery networks, such as CloudFlare, now serve a significant proportion of HTTPS traffic on behalf of millions of high-profile websites. However, the performance gains offered by CDNs must be balanced against new security threats. CDNs offer a tempting target for powerful adversaries, especially in the context of mass surveillance: by stealing the private keys from a single edge server, an attacker can impersonate millions of websites, and decrypt any previously recorded TLS-RSA connections.

Proxied TLS architectures such as Keyless SSL seek to mitigate the impact of edge server compromise, but such designs need to be cryptographically analyzed against strong threat models. In this paper, we introduced the notion of 3(S)ACCE-security that tries to formally capture the guarantees that a proxied handshake *should provide*. Our definition aims to protect clients even though they are oblivious that any proxying is taking place. Instead, we make the origin server *accountable* for auditing the behavior of the edge

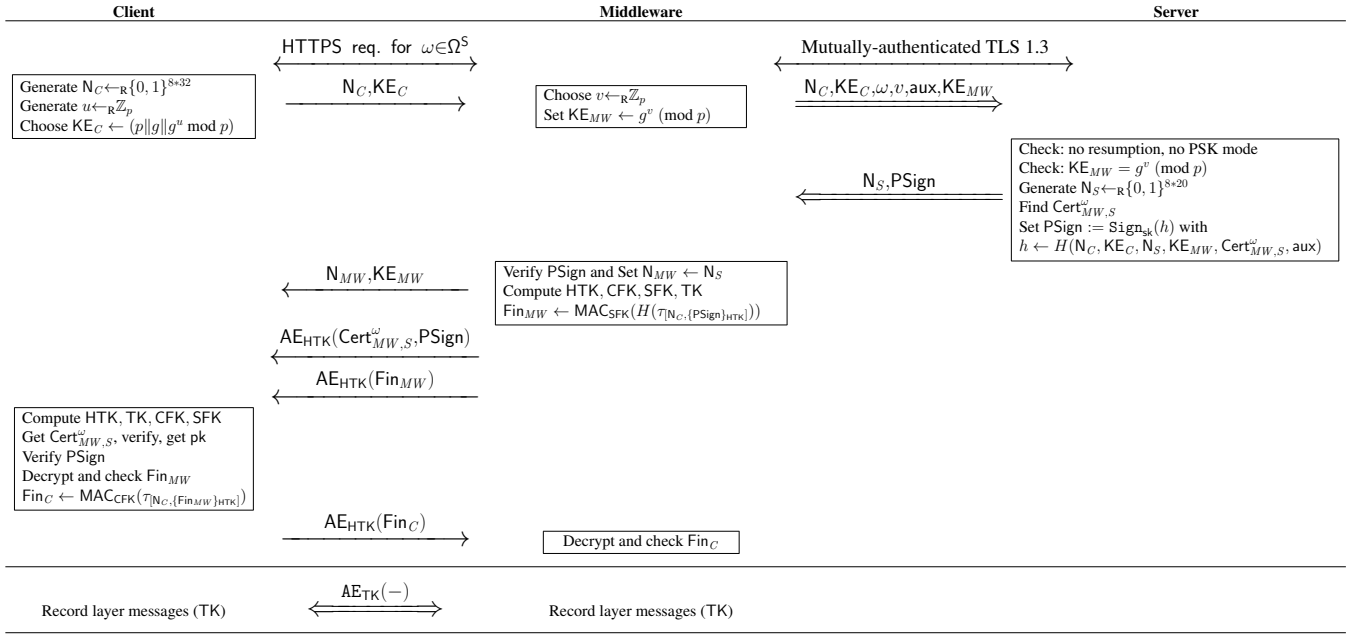

Figure 5: A 3(S)ACCE-secure variant with TLS 1.3. We denote by aux the non-explicit auxiliary information we consider, such as ciphersuites, TLS version number, and extensions.

server, and for detecting and de-authorizing compromised edge servers. Admittedly, this stronger security comes at the cost of more computation and more computation at the origin server. We show that new protocol designs, such a TLS 1.3 can enable both efficient and secure designs.

To fully protect clients' privacy, we believe that clients should be made aware of proxying, so they can decide whether they want a faster but less secure connection to a CDN or a slower but more secure connection directly to the origin server. Designing and analyzing new CDN architectures where both clients and servers collaborate to protect against malicious middleboxes (*e.g.*, see [12]) is a promising direction for future research.

Acknowledgement

The authors thank the anonymous reviewers of Euro S&P for their helpful comments. Pierre-Alain Fouque and Cristina Onete also thank the Agence Nationale de Recherche (ANR), which partially funded this research by means of the SafeTLS project. Ioana Boureanu acknowledges the support of the Marie Skłodowska-Curie grant No 661362 by the European Union under the Horizon 2020 research and innovation programme.

References

[1] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *IEEE Symposium on Security and Privacy (SP'13)*, 2013.

[2] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In *CRYPTO*, pages 232–249, 1993.

[3] Benjamin Berdouche, Kartikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre Yves Strub, and Jean Karim Zinzindohoue. A messy state of the union: Taming the composite state machines of TLS. In *Proceedings of IEEE S&P 2015*, pages 535–552. IEEE, 2015.

[4] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Alfredo Pironti, and Pierre-Yves Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *Proceedings of IEEE S&P 2014*, pages 98–113. IEEE, 2014.

[5] Christina Brzuska, Håkon Jacobsen, and Douglas Stebila. Safely exporting keys from secure channels: on the security of EAP-TLS and TLS key exporters. In *EuroCrypt*, 2016.

[6] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol candidates. In *ACM CCS*, pages 1197–1210, 2015.

[7] C.E. GERO, J.N. SHAPIRO, and D.J. BURD. Terminating ssl connections without locally-accessible private keys, June 20 2013. WO Patent App. PCT/US2012/070,075.

[8] Tibor Jager, Florian Kohlar, Sven Schäge, and Jörg Schwenk. On the security of TLS-DHE in the standard model. In *Proceedings of CRYPTO 2012*, volume 7417 of *LNCS*, pages 273–293, 2012.

[9] Markulf Kohlweiss, Ueli Maurer, Cristina Onete, Björn Tackmann, and Daniele Venturi. (De-)constructing TLS 1.3. In *Proceedings of Indocrypt*, volume 9462 of *LNCS*, pages 85–102. Springer, 2015.

[10] Hugo Krawczyk, Kenneth Paterson, and Hoeteck Wee. On the security of the TLS protocol: A systematic analysis. In *Proceedings of CRYPTO 2013*, volume 8042 of *LNCS*, pages 429–448, 2013.

[11] Hugo Krawczyk and Hoeteck Wee. The OPTLS protocol and tls 1.3. In *Proceedings of Euro S&P*. IEEE, 2016.

[12] David Naylor, Kyle Schomp, Matteo Varvello, Ilias Leontiadis, Jeremy Blackburn, Diego R. López, Konstantina Papagiannaki, Pablo Rodriguez Rodriguez, and Peter Steenkiste. Multi-Context TLS (mcTLS): Enabling Secure In-Network Functionality in TLS. In *Proceedings of SIGCOMM 2015*, pages 199–212. ACM, 2015.

[13] Douglas Stebila and Nick Sullivan. An analysis of TLS handshake proxying. In *Proceedings of TrustCom 2015*. IEEE, 2015. To appear.