Modelling ciphersuite and version negotiation in the TLS protocol

Benjamin Dowling and Douglas Stebila

Queensland University of Technology, Brisbane, Australia
{b1.dowling,stebila}@qut.edu.au

Abstract. Real-world cryptographic protocols such as the widely used Transport Layer Security (TLS) protocol support many different combinations of cryptographic algorithms (called ciphersuites) and simultaneously support different versions. Recent advances in provable security have shown that most modern TLS ciphersuites are secure authenticated and confidential channel establishment (ACCE) protocols, but these analyses generally focus on single ciphersuites in isolation. In this paper we extend the ACCE model to cover protocols with many different sub-protocols, capturing both multiple ciphersuites and multiple versions, and define a security notion for secure negotiation of the optimal sub-protocol. We give a generic theorem that shows how secure negotiation follows, with some additional conditions, from the authentication property of secure ACCE protocols. Using this framework, we analyse the security of ciphersuite and three variants of version negotiation in TLS, including a recently proposed mechanism for detecting fallback attacks.

Keywords: Transport Layer Security (TLS); ciphersuite negotiation; version negotiation; downgrade attacks; cryptographic protocols

1 Introduction

The security of much communication on the Internet depends on the Transport Layer Security (TLS) protocol [1,2,3], previously known as the Secure Sockets Layer (SSL) protocol [4]. TLS allows two parties to authenticate each other using public keys and subsequently establish a secure channel which provides confidentiality and integrity of messages. The general structure of all versions of SSL and TLS is that a *handshake protocol* is run, in which a set of cryptographic preferences are first negotiated, then an authenticated key exchange protocol is used to perform mututal or server-to-client authentication and establish a shared session key; and then the *record layer* is active, in which the shared session key is used with authenticated encryption for secure communication. TLS supports many combinations of cryptographic parameters, called *ciphersuites*: as of this writing, more than 300 ciphersuites have been standardized, with various combinations of digital signature algorithms, key exchange methods, hash functions, ciphers and modes, and authentication codes.

Given the paramount importance of TLS, formal understanding of its security is an important goal of cryptography. Wagner and Schneier [5] were among the

first to study SSL, and in particular compared SSLv3 [4] to SSLv2 [6]. A key difference was that SSLv3 provided authentication of the full handshake, whereas SSLv2 omitted the ciphersuite negotiation messages, leaving SSLv2 vulnerable to ciphersuite rollback attacks: an active attack could force clients and servers to negotiate weaker ciphersuites than the best they mutually support.

Provable security of TLS. A significant body of work is devoted to studying the provable security of TLS: the majority of it focuses on individual ciphersuites. Early work on the provable security of TLS analyzed truncated forms of the TLS handshake [7,8] and a simplified record layer [9]. More recently, unmodified versions of the TLS constructions have been studied by introducing suitable security definitions. Paterson et al. [10] showed that certain modes of authenticated encryption in the TLS record layer satisfy a property known as *secure length-hiding authenticated encryption.* In 2012, Jager et al. [11] showed that, under suitable assumptions on the underlying cryptographic primitives, the signed-Diffie–Hellman TLS ciphersuite is a secure *authenticated and confidential channel establishment (ACCE) protocol*, yielding the first full proof of security of an unmodified TLS ciphersuite. Subsequent efforts [12,13,14] have shown that most other TLS ciphersuites (using static or ephemeral Diffie–Hellman, RSA key transport, or pre-shared keys) are also secure. Other recent approaches to analyzing TLS include an alternative composability notion [15] and formal verification of an implementation [16].

Previous security results on TLS all focus on analyzing a single ciphersuite in isolation. Among other things, TLS allows for versions and ciphersuites to be negotiated within the protocol, sessions to be resumed, renegotiation within a session. Moreover, in practice servers often use the same long-term key in many different ciphersuites, and browsers re-attempt failed handshakes with lower versions. This variety of complex functionality leaves a gap between single-ciphersuite results and real-world security. Some work has tried to bridge that gap: Giesen et al. [17] extended the ACCE model to analyze the renegotiation security of TLS in light of the attack of Ray and Dispensa [18]; Mavrogiannopoulos et al. [19] demonstrated a cross-ciphersuite attack first suggested by Wagner and Schneier [5] when the same long-term signing key is used in two different key exchange methods; Bergsma et al. [20] developed an ACCE-based model for multi-ciphersuite security and showed that the Secure Shell (SSH) protocol is multi-ciphersuite security, though the Mavrogiannopolous et al. attack rules out a general proof that TLS is multi-ciphersuite secure; and Bhargavan et al. [21] showed that some combinations of ciphersuites do support key agility (a concept related to multi-ciphersuite security).

Ciphersuite and version negotiation. This work aims to give a formal treatment of the negotiation of ciphersuites and versions in real-world protocols like TLS. For *ciphersuite negotiation* in TLS, the client sends in its first handshake message a list of its supported ciphersuites in order of preference, and the server responds with one of those that it also supports. With regards to *version negotiation*, most browsers and servers support multiple versions of SSL and TLS, with

the majority supporting and accepting SSLv3 and TLSv1.0 (with more modern software also supporting TLSv1.1 and TLSv1.2). The differences between versions can significantly affect security: TLSv1.1 and TLSv1.2 are less vulnerable to certain weaknesses in record layer encryption in some ciphersuites; SSLv3 does not support extensions in the `ClientHello` and `ServerHello` negotiation messages; and some extensions such as the Renegotiation Information Extension [22] are essential to prevent certain types of attacks; and some ciphersuites with newer, more efficient and secure algorithms are only supported in TLSv1.2.

The TLS protocol standards support a limited version negotiation mechanism at present: the client sends the highest version it supports, and the server responds with the highest version it supports that is less than or equal to the client's version, and that is the version the parties continue to use. However, some server implementations do not correctly respond to `ClientHello` messages containing higher versions, and instead of returning their highest supported version in the `ServerHello` message will instead fail and return an error. Thus, in practice a more complex version negotiation mechanism is often employed by web browsers, sometimes called the "downgrade dance". The client's browser will try to negotiate the highest version it supports (say, TLSv1.2); if the handshake fails, then the browser will retry with each lower enabled version (TLSv1.1, TLSv1.0, SSLv3) until it succeeds. This improved compatibility with incorrect server implementation comes at the cost of decreased efficiency and more importantly decreased security: the client and server have no way of detecting whether the negotiated version is actually the highest version they both support or a lower version due to an attacker maliciously injecting failure messages. In light of this potential downgrade attack, a very recent Internet-Draft by Möller and Langley has proposed a new backwards-compatible mechanism for detecting such attacks [23], but as of this writing has yet to be standardized or deployed. The SCSV extension is proposed to work as follows: If the client is falling back to an earlier version due to a handshake failure, the client includes the SCSV value indicating that it has fallen back; if the server observes the fallback SCSV but supports a higher version than the client requests, the server returns an error indicating that inappropriate fallback has been detected.

Contributions. We investigate the security of version and ciphersuite negotiation in TLS. We do so by introducing an extension to the ACCE security model that generically captures negotiation of "sub-protocols". In particular, using ideas from the multi-ciphersuite ACCE security experiment of Bergsma et al. [20], we extend the ACCE security experiment to include "sub-protocols": a single protocol (such as TLS) consists of a *negotiation protocol* $\mathtt{NP}$ and several *sub-protocols* $\overrightarrow{\mathtt{SP}}$ (such as different ciphersuites or different versions), and in each session the parties use the negotiation protocol to identify which sub-protocol they will use for that session. We define **secure negotiation** for a negotiable protocol, and use this to derive a negotiation-authentication theorem which allows us to relate the security of sub-protocol negotiation to ACCE authentication under certain conditions. Intuitively, if each sub-protocol individually is a secure ACCE protocol with an independent long-term key, and if the transcript of all of the

messages in the negotiation protocol is authenticated by the sub-protocol, then the authentication detects any attempt by an attacker to carry out a downgrade attack. It is important to note that the aforementioned cross-ciphersuite attack breaks ACCE authentication security under long-term key reuse setting; thus, in order to obtain results on multi-ciphersuite TLS, our framework assume long-term keys are independent for each sub-protocol. Existing analyses of TLS show ([11,12,13,14]) that authentication security of TLS holds under independent long-term key assumptions.

Having established the secure negotiation framework and tools we proceed to study **version and ciphersuite negotiation in TLS** in several forms:

1. *Ciphersuite negotiation within a single version:* For a fixed version of TLS, by application of the negotiation-authentication theorem we show that TLS provides secure ciphersuite negotiation.
2. *Version negotiation, no fallback:* For clients and servers that support multiple versions of TLS but do not attempt to fallback to earlier versions upon handshake failure, we show that TLS also provides secure version negotiation via the negotiation-authentication theorem.
3. *Version negotiation, with fallback:* For clients and servers that support multiple versions of TLS and where the client *will* fallback to earlier versions if the handshake fails, we see that secure negotiation is not provided demonstrating that our secure negotiation definition does detect this undesired behaviour.
4. *Version negotiation, with fallback using signalling ciphersuite value (SCSV):* A recent Internet-Draft [23] proposes the use of a special flag in the `ClientHello` message. We show that this SCSV does provide TLS with a secure version negotiation mechanism even when fallbacks are used.

2 The TLS Protocol

In this section, we give the details for ciphersuite negotiation and three variants of version negotiation in the TLS protocol. The following is a description of the two messages most relevant to TLS ciphersuite and version negotiation: the `ClientHello` and `ServerHello` messages; descriptions of the subsequent messages can be found in the TLS protocol specification [3].

- `ClientHello`: Sent by the client to begin the TLS handshake. Consists of: the highest version that the client supports v; a random nonce r_c; the optional identifier of previous session that the client wishes to resume; a list of client ciphersuite preferences $\vec{c}$; and an optional list of extensions `extensions` describing additional options or functionality.
- `ServerHello`: Sent by the server in response to `ClientHello`. Consists of: the negotiated choice of version v; a random nonce r_s; a session identifier; the negotiated choice of ciphersuite c^*; and an optional list of extensions.

2.1 Ciphersuite negotiation in TLS

As indicated above, in TLS the client sends in `ClientHello`.$\vec{c}$ a list of supported ciphersuites, ordered from most preferred to least preferred. The server also has a

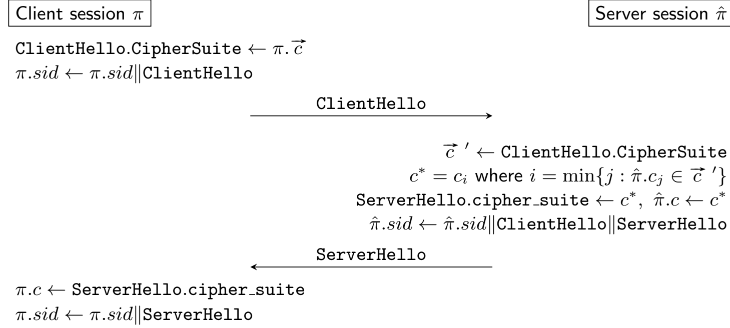


Fig. 1. $\mathtt{NP}_{\mathsf{cs}}$: Ciphersuite negotiation protocol in TLS

list of supported ciphersuites ordered by preference, and selects its most preferred ciphersuite that the client also supports. This ciphersuite negotiation protocol $\mathtt{NP}_{\mathsf{cs}}$ is described algorithmically in Figure 1. In our formalism, the adversary activates each party with the vector $\vec{c}$ of their ordered ciphersuite preferences for that session.

2.2 Version negotiation in TLS

As indicated in the standards, in TLS the client sends in `ClientHello`.v the highest version of TLS supports, and the server responds in its `ServerHello` message with the chosen version. In practice, buggy TLS server implementations sometimes reject unrecognised versions rather than negotiating a lower version, so some TLS clients will carry out fallback, where they try again with a lower supported version. We identify three variants of TLS version negotiation as follows. Recall again that in our formalism, the adversary activates each party with a vector $\vec{v}$ of their supported versions for that session.

– *No-fallback version negotiation*, denoted $\mathtt{NP}_{\mathsf{v}}$: Version negotiation as defined by the TLS standards (Figure 1).
– *Fallback version negotiation (the "downgrade dance")*, denoted $\mathtt{NP}_{\mathsf{v\text{-}fb}}$: Version negotiation as defined by the TLS standards, but allowing version fallback (Figure 3).
– *Fallback version negotiation with SCSV*, denoted $\mathtt{NP}_{\mathsf{v\text{-}fb\text{-}scsv}}$: The client proceeds as in fallback version negotiation, but when falling back to a lower version, the client also includes in its ciphersuite list a *fallback signalling ciphersuite value* (SCSV) to indicate that it has fallen back; this ciphersuite cannot be negotiated, and instead simply serves as a flag. If the server sees that it would negotiate a version lower than its highest version and the client has included the fallback SCSV, the server aborts and responds with `inappropriate_fallback` (Figure 4).

Note that the transcript ($\pi.sid$ in our formalism) "resets" in fallback version negotiation: matching conversations are based solely on the last handshake, rather than all handshakes that may have fallen back.

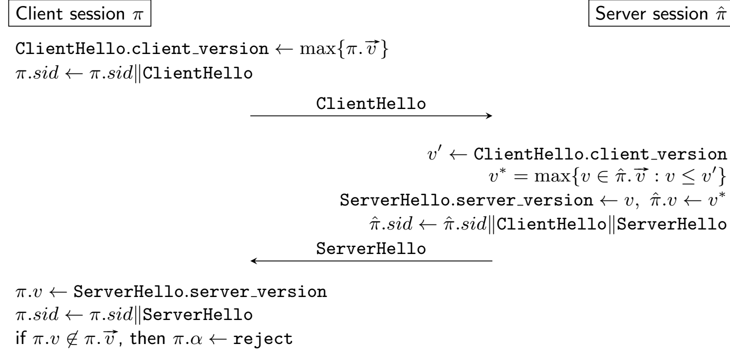
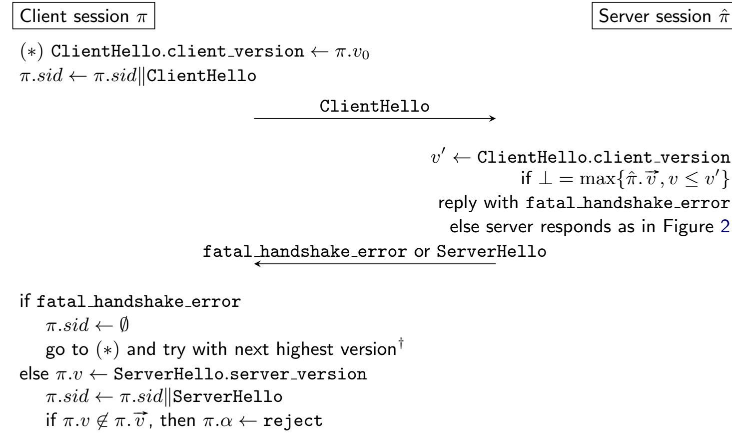


Fig. 2. $\mathtt{NP}_{\mathsf{v}}$: No-fallback version negotiation protocol in standard TLS



† Note that the "go to $(*)$" step in the client execution means that execution remains in the same session for the client; however, the server, receiving a new `ClientHello`, will start a new session.

Fig. 3. $\mathtt{NP}_{\mathsf{v\text{-}fb}}$: Fallback version negotiation in TLS (the "downgrade dance")

3 Security definitions

We begin by introducing the standard *authenticated and confidential channel establishment (ACCE)* protocol framework as introduced by Jager et al. [11]. We then extend the definition to cover protocols which negotiate a sub-protocol, and define the secure negotiation property.

3.1 Authenticated and confidential channel establishment (ACCE) protocols

An ACCE protocol is a multi-party protocol. Each instance of the protocol is executed between two parties: during the *pre-accept phase*, the parties establish a shared secret key and mutually authenticate each other; this is followed by a *post-accept phase*, which allows parties to transmitted authenticated and encrypted

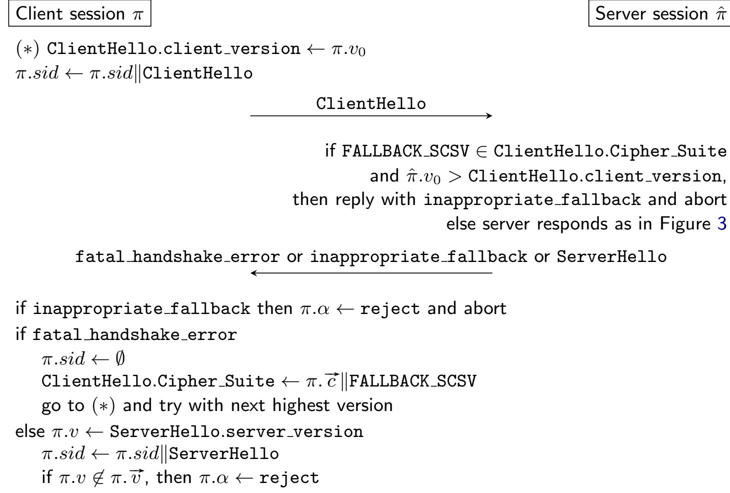


Fig. 4. $\mathtt{NP}_{\mathsf{v\text{-}fb\text{-}scsv}}$: Fallback negotiation in TLS with signalling ciphersuite value

payload data. We now proceed to describe the ACCE security model in detail, beginning with the per-session variables and adversary interaction. Note that, for simplicity, we restrict to the mutual authentication setting as in the original ACCE definition of Jager et al. [11], but our results apply equally to server-only authenticated ACCE [12,13]. Each ciphersuite in TLS is considered a separate ACCE protocol with independent long-term keys, which limits the application of the framework to implementations of TLS with no long-term key reuse.

Parties and sessions. The execution environment consists of n_P parties, denoted $P_1, P_2, \ldots P_{n_P}$. Each party P_i has a long-term public/private key pair (pk_i, sk_i), generated according to the protocol specification. Each party can execute multiple runs of the protocol either sequentially or in parallel; each run is referred to as a *session*, and π_i^s denotes the sth session at party i. For each session, the party maintains a collection of the following *per-session variables*, and we overload the notation π_i^s to refer to both the session itself and the corresponding collection of per-session variables.

- $\rho \in \{\mathtt{init}, \mathtt{resp}\}$: The role of the party in the session.
- $\mathsf{pid} \in [n_P]$: The index of the intended peer of this session.
- $\alpha \in \{\mathtt{in\text{-}progress}, \mathtt{accept}, \mathtt{reject}\}$: The execution status of the session.
- k: A session key, or $\perp$; k may for example consist of sub-keys for bi-directional authentication and encryption.
- T: Transcript of all messages sent and received by the party in this session.
- sid: A *session identifier*, consisting of an ordered subset of messages in T as defined by the protocol specification.[1]

[1] Our separation of the transcript and session identifier follows [20] and is a slight change compared to the original ACCE model [11] to allow for consideration of protocols where some messages are not authenticated.

– Any additional state specific to the protocol (such as ephemeral Diffie–Hellman exponents).
– Any additional state specific to the security experiment.

We use the notation $\pi_i^s.\rho$ etc. to denote each variable of a particular session.

While a session has set $\alpha \leftarrow$ `in-progress`, we say that the session is in the *pre-accept phase*; after the session has set $\alpha \leftarrow$ `accept`, we say that the session is in the *post-accept phase.*

Definition 1 (ACCE protocol). *An* ACCE protocol `P` *consists of a probabilistic long-term public-private key pair generation algorithm, as well as probabilistic algorithms defining how the party generates and responds to protocol messages. The protocol specification also includes a* stateful length-hiding authenticated encryption (sLHAE) *scheme* `StE` *[10,11] for sending and receiving payload data on the record layer.*

Adversary interaction. In the security experiment, the adversary controls all interactions between parties: the adversary activates sessions with initialization information; it delivers messages to parties, and can reorder, alter, delete, replace, and create messages. The adversary can also compromise certain long-term and per-session values. The adversary interacts parties using the following queries.

The first query models normal, unencrypted operation of the protocol, generally corresponding to the pre-accept phase.

– $\mathsf{Send}(i, s, m) \xrightarrow{\$} m'$: The adversary sends message m to session π_i^s. Party P_i processes m according to the protocol specification and its per-session variables π_i^s, updates its per-session state, and optionally outputs an outgoing message m'.
  There is a distinguished initialization message which allows the adversary to activate the session with certain information, such as the intended role ρ the party in the session, the intended communication partner pid, and any additional protocol-specific information; when we extend to the negotiable setting in the next subsection, this will include ciphersuite and/or version preferences.
  This query may return error symbol $\perp$ if the session has entered state $\alpha =$ `accept` and no more protocol messages are to be transmitted over the unencrypted channel.

The next two queries model adversarial compromise of long-term and per-session secrets.

– $\mathsf{Corrupt}(i) \xrightarrow{\$} sk_i$: Returns long-term secret key sk_i of party P_i.
– $\mathsf{Reveal}(i, s) \xrightarrow{\$} \pi_i^s.k$: Returns session key $\pi_i^s.k$.

The final two queries, Encrypt and Decrypt, model communication over the encrypted channel. The adversary can direct parties to encrypt plaintexts and obtains the corresponding ciphertext. The adversary can deliver ciphertexts to parties, which are then decrypted. To accommodate defining the security property of indistinguishability of ciphertexts, the Encrypt query takes two messages, and one of the tasks of the adversary is to distinguish which was encrypted. The

exact specification of $\mathsf{Encrypt}$ and $\mathsf{Decrypt}$ is specified in Figure 4 of [24] (the full version of [11]), and is omitted in this paper as these queries are not required for defining negotiable security.

ACCE security definitions. We now present the two sub-properties that define security of ACCE protocols. Like authenticated key exchange (AKE) security definitions, the ACCE framework requires that the protocol provides secure mutual authentication. The difference lies in the encryption-challenge: instead of key indistinguishability (found in AKE experiments) the ACCE framework requires that all payload data transmitted between parties (during the post-accept stage) is over an authenticated and confidential channel. The original motivation for this distinction is that real-world protocols often have key confirmation messages (for example, TLS's `Finished` message), which can act as a key-distinguisher in a AKE security framework. ACCE solves this by focusing on message confidentiality and integrity instead of key indistinguishability.

We start by defining matching conversations and the mutual authentication property of an ACCE protocol. Matching conversations is a property useful for describing the correctness and authentication of a protocol, first introduced by Bellare and Rogaway [25]. [2]

Definition 2 (Matching sessions). *A session* π_j^t matches *session* π_i^s *if:*

- *if* P_i *sent the last message in* $\pi_i^s.sid$*, then* $\pi_j^t.sid$ *is a prefix of* $\pi_i^s.sid$*; or*
- *if* P_i *received the last message in* $\pi_i^s.sid$*, then* $\pi_i^s.sid = \pi_j^t.sid$*,*

where X *is a* prefix *of* Y *if* X *contains at least one message and the messages in* X *are identical to and in the same order as the first* $|X|$ *messages in* Y*.*

Definition 3 (Mutual authentication). *A session* π_i^s accepts maliciously *if*

- $\pi_i^s.\alpha = \mathtt{accept}$*;*
- $\pi_i^s.\mathsf{pid} = j$ *and no* $\mathsf{Corrupt}(j)$ *query was issued before* $\pi_i^s.\alpha$ *was updated to* $\mathtt{accept}$*; and*
- *there is not a unique session* π_j^t *that matches* π_i^s*.*

We define $\mathrm{Adv}_{\mathtt{P}}^{\mathsf{acce\text{-}auth}}(\mathcal{A})$ *as the probability that, when probabilistic adversary algorithm* $\mathcal{A}$ *terminates in the ACCE experiment for protocol* $\mathtt{P}$*, there exists a session that has accepted maliciously.*

Channel security for ACCE protocols is defined as the ability of the adversary to break confidentiality or integrity of the channel. As the channel security definition does not play a role in the remainder of this paper, we omit the definition and refer the reader to Definition 5.2 of [11] for details. Using the notation of Bergsma et al. [20], the expression $\mathrm{Adv}_{\mathtt{P}}^{\mathsf{acce\text{-}aenc}}(\mathcal{A})$ denotes the probability that the adversary $\mathcal{A}$ breaks channel security of protocol $\mathtt{P}$.

Definition 4 (ACCE-secure). *A protocol* $\mathtt{P}$ *is said to be* ϵ*-ACCE-secure against an adversary* $\mathcal{A}$ *if we have that* $\mathrm{Adv}_{\mathtt{P}}^{\mathsf{acce\text{-}auth}}(\mathcal{A}) \le \epsilon$ *and* $\mathrm{Adv}_{\mathtt{P}}^{\mathsf{acce\text{-}aenc}}(\mathcal{A}) \le \epsilon$*.*

[2] Our formulation is a slight variant of Jager et al. [11]: we match on *session identifiers* (a well-defined subset of messages sent and received) rather than the full transcript.

3.2 Negotiable ACCE protocols

In this section we define formally a negotiable ACCE protocol and the corresponding security notions. We do so by explaining the differences with Section 3.1. The basis of our definition is the multi-ciphersuite ACCE definition of Bergsma et al. [20], but like the ACCE definitions above we do not consider use of the same long-term key in multiple sub-protocols. We then define the secure negotiation property.

Differences in execution environment. A *negotiable ACCE protocol* is composed of a *negotiation protocol* $\mathtt{NP}$ and a collection of *sub-protocols* $\overrightarrow{\mathtt{SP}}$; we use the notation $\mathtt{NP}\|\overrightarrow{\mathtt{SP}}$ to denote the combined protocol. For example:

- In TLS with multiple ciphersuites, the negotiation protocol $\mathtt{NP}_{\mathtt{cs}}$ consists of the sending and receiving of the `ClientHello` and `ServerHello` messages as shown in Figure 1, and each sub-protocol $\mathtt{SP}_i$ corresponds to the remaining messages in ciphersuite i.
- For TLS with multiple versions, each sub-protocol $\mathtt{SP}_i$ corresponds to a different version of TLS; the description of the negotiation protocol depends on whether and how fallback is handled, and is described in Section 2.

Parties and sessions. In a negotiable ACCE protocol, each party P_i has a vector of long-term public/private key pairs $(\overrightarrow{pk_i}, \overrightarrow{sk_i})$, one for each sub-protocol.

Each session in a negotiable ACCE protocol maintains two additional per-session variables:

- $\vec{n}$: An ordered list of negotiation preferences.
- n: The index of the negotiated sub-protocol.

In the execution of $\mathtt{NP}\|\overrightarrow{\mathtt{SP}}$, the protocol begins by running the negotiation protocol $\mathtt{NP}$, which has as input the ordered list $\vec{n}$ of negotiation preferences; the negotiation protocol updates per-session variables, and in particular updates the index n of the negotiated sub-protocol. Once the negotiation protocol completes, subprotocol $\mathtt{SP}_n$ is run, operating on the same per-session variables.

Adversary interaction. The adversary can interact with parties exactly as in Section 3.1. The only difference is that in the distinguished initialization message in the Send query, the adversary also includes an ordered list $\vec{n}$ of the sub-protocol preferences that the party should use in that session. For example, in ciphersuite negotiation, the adversary may direct the party to prefer RSA over Diffie–Hellman in one session and Diffie–Hellman over RSA in another session. For version negotiation in TLS, order of the list is descending and contiguous (i.e., if TLSv1.2 and TLSv1.0 are listed as supported, TLSv1.1 must be listed).

Secure negotiation. Intuitively, a negotiable protocol has secure negotiation if the adversary cannot cause the parties to successfully negotiate a worse sub-protocol than the best one they both support. We formalize this via an optimality function, which will be different for each protocol (for example, the optimality function for TLS ciphersuite negotiation is different from that of TLS version negotiation).

Definition 5 (Optimal negotiation). *Let $\omega(\vec{x}, \vec{y}) \to z$ be a function taking as input two ordered lists and outputting an element of one of the lists or $\bot$. We say that two sessions π_i^s and π_j^t* do not have optimal negotiation with respect to *ω unless $\pi_i^s.n = \pi_j^t.n = \omega(\pi_i^s.\vec{n}, \pi_j^t.\vec{n})$.*

For TLS ciphersuite negotiation, the optimality function yields the first ciphersuite in the server's ordered list of preferences also supported by the client:

$$\omega_{\mathsf{cs}}(\vec{x}, \vec{y}) = y_i, \text{ where } i = \min\{j : y_j \in \vec{x}\} \ . \tag{1}$$

For TLS version negotiation, the optimality function yields the highest version that is supported by both the client and the server:

$$\omega_{\mathsf{vers}}(\vec{x}, \vec{y}) = \max\{\vec{x} \cap \vec{y}\} \ . \tag{2}$$

For TLS version negotiation, we impose the order TLSv1.2 $>$ TLSv1.1 $>$ TLSv1.0 $>$ SSLv3.0 $>$ SSLv2.0.

We can now define what it means for a protocol to have secure negotiation, either of a particular sub-protocol or over all sub-protocols.

Definition 6 (Secure negotiation of a sub-protocol). *We say that a session π_i^s* has negotiated a sub-protocol n^* insecurely with respect to *ω if*

- *$\pi_i^s.\alpha = \texttt{accept}$;*
- *$\pi_i^s.n = n^*$;*
- *π_i^s has not accepted maliciously (in the sense of Definition 3); and*
- *π_i^s and π_j^t do not have optimal negotiation with respect to ω, where π_j^t is the unique session that matches π_i^s.*

We define $\mathrm{Adv}^{\mathsf{neg},\omega}_{\mathtt{NP}\|\vec{\mathtt{SP}},n^*}(\mathcal{A})$ as the probability that, when $\mathcal{A}$ terminates in the negotiable-ACCE experiment for $\mathtt{NP}\|\vec{\mathtt{SP}}$, there exists a session that has negotiated sub-protocol n^* insecurely with respect to ω.

Remark 1 (Secure negotiation vs. authentication). Secure negotiation, as defined is a stronger property than authentication: the third condition of Definition 6 effectively incorporates the authentication security definition. Recall that authentication is based on matching session identifiers; if a protocol uses the full transcript as the session identifier, then negotiation generally reduces to authentication, which is shown in the theorem in the next section. However, if a protocol uses some subset of the transcript as the session identifier, or for example "resets" the session identifier partway through the handshake as in TLS version fallback, then negotiation becomes non-trivially different from authentication and requires further consideration, as we shall see in Section 6.

4 Negotiation-authentication theorem

We now present our negotiation-authentication theorem, which allows us under certain conditions to relate the probability of an adversary forcing a user to

insecurely negotiate to $\mathtt{NP}\|\mathtt{SP}_n$ to the probability of an adversary breaking authentication in $\mathtt{NP}\|\mathtt{SP}_n$. At first glance, this seems obvious: if all of the messages in a protocol are securely authenticated, then it should be impossible for an adversary to trick the parties into negotiating something sub-optimal. There is a reason why the application of the theorem is not trivial: In practise, not all protocols authenticate all messages in the handshake. As we will see Section 6, version fallback in TLS results in some parts of the negotiation not being authenticated. Historically, ciphersuite downgrade in SSLv2 was possible as the negotiation phase wasn't entirely authenticated.

To be able to apply this theorem, the protocol $\mathtt{P}$ has to satisfy certain conditions as shown in the theorem statement below. Precondition 1 captures the notion that protocols where all handshake message are authenticated, or at least all handshake messages related to negotiation are authenticated, should allow us to reduce negotiation security to authentication security. Precondition 2 is a simply that, in the absence of an active adversary, parties negotiate correctly.

Theorem 1. *Let $\mathtt{NP}\|\overrightarrow{\mathtt{SP}}$ be a negotiable ACCE protocol and let ω be an optimality function. Suppose that:*

1. *all message sent and received by a party in the negotiation phase are included in the session identifier; and*
2. *in the absence of an active adversary, negotiation is always optimal with respect to ω,*

then for all algorithms $\mathcal{A}$ and for all sub-protocols $\mathtt{SP}_n$, $\mathrm{Adv}^{\mathsf{neg},\omega}_{\mathtt{NP}\|\overrightarrow{\mathtt{SP}},n}(\mathcal{A}) = \mathrm{Adv}^{\mathsf{acce\text{-}auth}}_{\mathtt{NP}\|\mathtt{SP}_n}(\mathcal{A})$.

The proof of Theorem 1 appears in the full version [26]. The brief description of the argument is as follows: By condition 1, both parties can verify that in presence of a passive adversary that negotiation was optimal with respect to ω. Since both parties can verify (via the session identifier) that the negotiation sub-protocol $\mathtt{SP}_n$ is the optimal sub-protocol, and $\mathtt{NP}||\mathtt{SP}_n$ itself is an ACCE protocol with negligible adversary advantage over a passive adversary, then negotiating to $\mathtt{NP}||\mathtt{SP}_n$ is both optimal and authenticated with negligible adversary advantage. Once we have related the security of negotiation to the security of authentication as in the equation in the theorem, we can make use of existing results on ACCE authentication security, for example the bounds on $\mathrm{Adv}^{\mathsf{acce\text{-}auth}}_{\mathtt{P}}(\mathcal{B})$ given for ACCE authentication security of $\mathtt{P} =$ TLS signed-Diffie–Hellman ciphersuites [11], $\mathtt{P} =$ TLS RSA key transport and $\mathtt{P} =$ TLS static Diffie–Hellman ciphersuites [13,12].

5 Analysis of TLS ciphersuite negotiation

Using our negotiation-authentication theorem from Section 4, we can show that TLS is ciphersuite-negotiation secure. We do this by showing that ciphersuite negotiation in TLS satisfies the two preconditions outlined in our negotiation-authentication theorem, and hence secure negotiation of ciphersuites is, not surprisingly, guaranteed by security of authentication. All outputs of ciphersuite negotiation are included in the session identifier (as seen in Figure 1), thus

precondition 1 is satisfied, provided the ciphersuite has secure authentication. In addition, TLS ciphersuite negotiation is optimal in the presence of a passive adversary, so precondition 2 is also satisfied. Details appear in the full version [26].

Corollary 1. *For the TLS protocol with ciphersuite negotiation* $\mathtt{NP}_{\mathsf{cs}}$ *as described in Figure 1 and TLS ciphersuites* $\overrightarrow{\mathtt{SP}}$*, an adversary* $\mathcal{A}$ *who can force a user to negotiate insecurely to ciphersuite* $\mathtt{SP}_n$ *with respect to the TLS ciphersuite optimality function* ω_{cs} *from equation (1) can also break authentication of that ciphersuite:* $\mathrm{Adv}^{\mathsf{neg},\omega_{\mathsf{cs}}}_{\mathtt{NP}_{\mathsf{cs}}\|\overrightarrow{\mathtt{SP}},n}(\mathcal{A}) = \mathrm{Adv}^{\mathsf{acce\text{-}auth}}_{\mathtt{SP}_n}(\mathcal{A})$.

6 Analysis of TLS version negotiation

In this section, we consider the three variants of TLS version negotiation identified in Section 2.2. The no-fallback version negotiation mechanism specified by the TLS standard, can easily be seen to be secure using our negotiation-authentication mechanism. When version fallback is permitted, version negotiation is no longer secure, as we demonstrate with a counterexample, and thus our model successfully captures this weakness of fallback. Finally, when the signalling ciphersuite value (SCSV) version fallback detection mechanism is used, we can show that TLS becomes version-negotiable secure.

6.1 TLS no-fallback version negotiation is secure

It is straightforward to apply our negotiation-authentication theorem to show that TLS with no-fallback version negotiation ($\mathtt{NP}_{\mathsf{v}}$ described in Figure 2), provides secure version negotiation. Here the session identifier consists of the entire transcript, which includes the client and server's version information, so precondition 1 of Theorem 1 is satisfied. It is clear that TLS provides optimal version negotiation in the presence of a passive adversary, so precondition 2 is satisfied. Thus the negotiation-authentication theorem yields Corollary 2. Details appear in the full version [26].

Corollary 2. *For the TLS protocol with no-fallback version negotiation* $\mathtt{NP}_{\mathsf{v}}$ *as described in Figure 2 and TLS versions* $\overrightarrow{\mathtt{SP}}$*, an adversary* $\mathcal{A}$ *who can force a user to negotiate insecurely to version* $\mathtt{SP}_n$ *with respect to the TLS version optimality function* ω_{vers} *from equation (2) can also break authentication of that version:* $\mathrm{Adv}^{\mathsf{neg},\omega_{\mathsf{vers}}}_{\mathtt{NP}_{\mathsf{v}}\|\overrightarrow{\mathtt{SP}},n}(\mathcal{A}) = \mathrm{Adv}^{\mathsf{acce\text{-}auth}}_{\mathtt{SP}_n}(\mathcal{A})$.

6.2 TLS fallback version negotiation is not secure

When examining version negotiation in TLS with fallback ($\mathtt{NP}_{\mathsf{v\text{-}fb}}$ from Figure 3), notice that many different `ClientHello` messages may be sent by the client before the handshake is accepted by the server. An active adversary may force this behaviour: instead of delivering the first few `ClientHello` attempts at handshake messages to the server, the adversary responds with `fatal_handshake_error`,

until the client sends a `ClientHello` which has a sufficiently low version that the adversary is satisfied. In practise, this may mean a client and a server both supporting TLSv1.2 may be downgraded to TLSv1.0 by an adversary returning a handshake error until the client attempts a TLSv1.0 `ClientHello` with a successful response. In this scenario, the session clearly has sub-optimal version-negotiation—the client and server both support TLSv1.2, but the adversary has caused a version 1.0 negotiation—and this provides a example that TLS with fallback is not version-negotiable secure.

In terms of our negotiation-authentication theorem, it fails to apply here because not every output of the negotiation phase is authenticated by the sub-protocol: only the successful `ClientHello` message is included in the transcript and is considered for matching sessions. Much like the ciphersuite-downgrade vulnerability in SSLv2, this allows an active adversary to modify and delete any of the previous exchanges between the server and client.

6.3 TLS fallback version negotiation with SCSV is secure

Similar to TLS fallback version negotiation, TLS fallback version negotiation with SCSV ($\mathtt{NP}_{\mathsf{v\text{-}fb\text{-}scsv}}$ as described in Figure 4) does not acknowledge or authenticate any messages previous to the `fatal_handshake_message` in the session identifier, and as such does not satisfy precondition 1 of Theorem 1. Thus, we cannot use the negotiation-authentication theorem to show that that fallback version negotiation with SCSV securely negotiates version. Instead, we provide a direct argument to show that fallback version negotiation with SCSV is secure provided that no-fallback TLS version negotiation is secure.

Theorem 2. *For the TLS protocol with fallback version negotiation with SCSV* $\mathtt{NP}_{\mathsf{v\text{-}fb\text{-}scsv}}$ *as described in Figure 4 and TLS versions* $\overrightarrow{\mathtt{SP}}$*, an adversary who can force a user to negotiate insecurely to version* $\mathtt{SP}_n$ *with respect to the TLS version optimality function* ω_{vers} *from equation 2 can also break authentication of that version:* $\mathrm{Adv}^{\mathsf{neg},\omega_{\mathsf{vers}}}_{\mathtt{NP}_{\mathsf{v\text{-}fb\text{-}scsv}}\|\overrightarrow{\mathtt{SP}},n}(\mathcal{A}) \leq \mathrm{Adv}^{\mathsf{acce\text{-}auth}}_{\mathtt{SP}_n}(\mathcal{A})$.

Proof. The security argument proceeds by showing that an adversary who is successful in breaking fallback version negotiation with SCSV is also successful in breaking authentication of the underlying ACCE protocol. We give a high-level description of the simulator behaviour below.

The simulator $\mathcal{B}$ in our argument recreates the SCSV mechanisms described in Figure 4 and ref. [23] using a version negotiation TLS challenger $\mathcal{C}$ for TLS with no-fallback version negotiation; more precisely, $\mathcal{B}$ simulates the neg experiment for $\mathtt{NP}_{\mathsf{v\text{-}fb\text{-}scsv}}\|\overrightarrow{\mathtt{SP}}$ using a challenger for $\mathtt{NP}_{\mathsf{v}}\|\overrightarrow{\mathtt{SP}}$.

$\mathcal{B}$ initially forwards all adversarial queries to the challenger $\mathcal{C}$ for each session. After receiving the `ClientHello` message for a session π from the adversary $\mathcal{A}$, the simulator is able to determine whether the version in the `ClientHello` would cause a handshake error. If the error would occur, $\mathcal{B}$ replies to $\mathcal{A}$ directly with `fatal_handshake_error`. If the error would not occur, $\mathcal{B}$ faithfully forwards all queries for that session between $\mathcal{A}$ and $\mathcal{C}$.

Upon receiving a `fatal_handshake_error` from $\mathcal{A}$ intended for a session π, the simulator uses a Send query to activate a new session π' that is activated identically to π except `FALLBACK_SCSV` is also included in the list of supported ciphersuites and the list of supported versions for π' is modified to no longer include the highest supported version v of the session π. $\mathcal{B}$ also adds π to a fallback list FL to determine which sessions have performed version-fallback.

Note that from $\mathcal{A}$'s point-of-view, π' and π are the same continuous session, and $\mathcal{B}$ now directs all queries sent to π to π' instead.

As well, $\mathcal{B}$, upon receiving a `ClientHello` from $\mathcal{A}$ that contains `FALLBACK_SCSV` in the list of supported ciphersuites, determines if the server's highest supported version is higher than the client's indicated version in the `ClientHello`. If so, $\mathcal{B}$ replies with an `inappropriate_fallback` error message. Note that the alert is fatal, so the simulator $\mathcal{B}$ will disregard all further Send queries directed to the server's session. If not, $\mathcal{B}$ forwards the `ClientHello` to $\mathcal{C}$ and continues to forward all messages for these sessions between $\mathcal{A}$ and $\mathcal{C}$.

This describes the simulator's behaviour during the experiment. Suppose at some point $\mathcal{A}$ breaks the negotiable security of a session π^*. There are two cases:

1. If π^* does not appear on $\mathcal{B}$'s fallback list FL, then all messages were forwarded faithfully between $\mathcal{A}$ and $\mathcal{C}$. An insecure version fallback to version $\mathtt{SP}_n$ in $\mathcal{B}$'s simulation of $\mathtt{NP}_{\mathsf{v\text{-}fb\text{-}scsv}}\|\overrightarrow{\mathtt{SP}}$ thus directly translates to insecure version negotiation to version $\mathtt{SP}_n$ in $\mathcal{C}$'s execution of $\mathtt{NP}_{\mathsf{v}}\|\overrightarrow{\mathtt{SP}}$. Hence, $\mathrm{Adv}^{\mathsf{neg},\omega_{\mathsf{vers}}}_{\mathtt{NP}_{\mathsf{v\text{-}fb\text{-}scsv}}\|\overrightarrow{\mathtt{SP}},n}(\mathcal{A}) \leq \mathrm{Adv}^{\mathsf{neg},\omega_{\mathsf{vers}}}_{\mathtt{NP}_{\mathsf{v}}\|\overrightarrow{\mathtt{SP}},n}(\mathcal{A})$. By Corollary 2, $\mathrm{Adv}^{\mathsf{neg},\omega_{\mathsf{vers}}}_{\mathtt{NP}_{\mathsf{v\text{-}fb\text{-}scsv}}\|\overrightarrow{\mathtt{SP}},n}(\mathcal{A}) \leq \mathrm{Adv}^{\mathsf{acce\text{-}auth}}_{\mathtt{SP}_n}(\mathcal{A})$.
2. If π^* does appear on $\mathcal{B}$'s fallback list FL, then the simulator will have rejected any non-optimal handshakes containing the SCSV. It follows then that the session must have accepted maliciously (either by the $\mathcal{A}$ impersonating the server party or by modifying the handshake of the fallback session $\pi^{*\prime}$). Thus an insecure fallback to version $\mathtt{SP}_n$ in $\mathcal{B}$'s simulation of $\mathtt{NP}_{\mathsf{v\text{-}fb\text{-}scsv}}\|\overrightarrow{\mathtt{SP}}$ directly translates to an authentication break in $\mathtt{SP}_n$. Hence, $\mathrm{Adv}^{\mathsf{neg},\omega_{\mathsf{vers}}}_{\mathtt{NP}_{\mathsf{v\text{-}fb\text{-}scsv}}\|\overrightarrow{\mathtt{SP}},n}(\mathcal{A}) \leq \mathrm{Adv}^{\mathsf{acce\text{-}auth}}_{\mathtt{SP}_n}(\mathcal{A})$. □

Need for contiguous support of TLS versions for fallback with SCSV. As shown above, SCSV does give additional protection against version downgrade attacks in TLS implementations that support version fallback. However, we observe that there is a drawback to the SCSV proposal as it stands: Non-contiguous support of versions in TLS implementations (a viable scenario in practise) can hamper interoperability between systems supporting checking for insecure fallback using SCSV.

In some implementations of TLS,[3] users can select a non-contiguous subset of TLS version support. For example, a user could—for some reason—enable TLSv1.2 and TLSv1.0, but not TLSv1.1.

[3] The current version of Microsoft Internet Explorer (11) and previous versions allow users to configure which subset of SSL/TLS versions are enabled (Internet options → Advanced → Security); Mozilla Firefox up to version 22 did as well. On the server side, Apache mod_ssl, Microsoft IIS, and nginx all allow the server administrate to select which subset of SSL/TLS versions to enable.

In relation to the SCSV, this can result in a connection attempt that could fail to accept without adversarial interaction. Consider the following scenario: suppose a client user selects TLSv1.2 and TLSv1.0 to support, and attempts to connect to a server that only supports TLSv1.1 and TLSv1.0, and will return a `fatal_handshake_error` for TLSv1.2. The client sends a `ClientHello` with TLSv1.2. After the server fails to parse the TLSv1.2 handshake correctly, it reply with a `fatal_handshake_error` message. The client falls back, sending a new `ClientHello` message with its next highest supported version, TLSv1.0, and includes `FALLBACK_SCSV` in the ciphersuite list to indicate it is falling back. The server notes the SCSV and rejects the handshake with `inappropriate_fallback` as recommended in the SCSV proposal because the server's highest supported version (TLSv1.1) is higher than the client's indicated version (TLSv1.0), despite the fact that the optimal negotiated version would be TLSv1.0.

An alternative mechanism for secure version fallback would be to include a signalling ciphersuite value for each version it supports, allow the parties to detect insecure fallback while allowing non-contiguous version support.

7 Discussion

We have introduced provable security notions for negotiation in Internet protocols, and extended the definition of ACCE protocols to utilise previous comprehensive ACCE proofs of TLS ciphersuites. We develop a negotiation-authentication theorem and show that ciphersuite negotiation in TLS is secure, under certain conditions about long-term key reuse. We follow by showing that the version negotiation in standards-defined TLS and the TLS implementation with the SCSV is also secure, but demonstrate that TLS implementations that utilise browser-based version fallback mechanisms are not version-negotiable secure. This analysis holds for TLS configurations that exclude sharing long-term keys across multiple versions. In practice, our analysis requires that TLS configurations (in order to have ciphersuite negotiation security) must use independent long-term keys and thus distinct digital certificates for each ciphersuite; this is currently a necessary cost in order to prevent cross-ciphersuite-like attacks from breaking authentication in TLS. To the best of our knowledge, no web server software currently permits configuring different certificate for different TLS ciphersuites with the same signing/key transport algorithm, nor different certificates for different TLS versions.

Future work. It seems possible that one could extend our analysis to include TLS configurations where long-term keys are shared across multiple *versions* but a single *fixed ciphersuite* (i.e. that TLS 1.2 and TLS 1.0 can reuse long-term keys in the same ciphersuite configuration). However in order to do so requires extensive modification of the negotiation framework to more closely resemble the multi-ciphersuite setting [20]. This remains a significant practical limitation on long-term key reuse across ciphersuites.

Proposed revisions to TLS in the current draft of TLS 1.3 [27] seem to make the protocol resistant to cross-ciphersuite and cross-version attacks. The main

change is that, in TLS 1.3, the value signed using the long-term secret key now includes (the hash of) all handshake messages, including the negotiated version and ciphersuite. As a result, the multi-ciphersuite composition framework of Bergsma et al. [20] should be applicable to both multi-version and multi-ciphersuite configurations of TLS: a signing oracle for a single sub-protocol could be constructed to avoid signing objects that would be valid in another sub-protocol, defeating the first step of the cross-ciphersuite attack. This could then imply negotiation-authentication security of TLS 1.3 with shared long-term keys. A thorough analysis is required to show this categorically, however.

Our techniques can also be applied to other protocols that negotiate cryptographic parameters or versions, the Secure Shell (SSH) protocol being a prime candidate. While SSH does have two versions, they are largely incompatible, and current best-practices including disabling v1 support, so there is little value in studying SSH version negotiation. However, SSH also supports multiple cryptographic algorithms, and our framework can easily be applied to SSH algorithm negotiation. Since the parties authenticate their entire transcript, including both the client's and server's algorithm preferences, our negotiation-authentication theorem readily implies that SSH has secure ciphersuite negotiation if it has secure authentication, which it does by the recent results of Bergsma et al. [20].

Acknowledgements

This research has been supported by Australian Research Council (ARC) Discovery Project grant DP130104304.

References

1. Dierks, T., Allen, C.: The TLS protocol version 1.0 (1999) RFC 2246.
2. Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) protocol version 1.1 (2006) RFC 4346.
3. Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) protocol version 1.2 (2008) RFC 5246.
4. Freier, A.O., Karlton, P., Kocher, P.C.: The Secure Sockets Layer (SSL) protocol version 3.0 (2011) RFC 6101; republication of original SSL 3.0 specification by Netscape of November 18, 1996.
5. Wagner, D., Schneier, B.: Analysis of the SSL 3.0 protocol. In: Proc. 2nd USENIX Workshop on Electronic Commerce. (1996)
6. Hickman, K.E.B.: The SSL protocol (version 0.2). http://www-archive.mozilla.org/projects/security/pki/nss/ssl/draft02.html (1995)
7. Jonsson, J., Kaliski Jr., B.S.: On the security of RSA encryption in TLS. In Yung, M., ed.: CRYPTO 2002. Volume 2442 of LNCS., Springer (2002) 127–142
8. Morrissey, P., Smart, N.P., Warinschi, B.: A modular security analysis of the TLS handshake protocol. In Pieprzyk, J., ed.: ASIACRYPT 2008. Volume 5350 of LNCS., Springer (2008) 55–73
9. Krawczyk, H.: The order of encryption and authentication for protecting communications (or: How secure is SSL?). In Kilian, J., ed.: CRYPTO 2001. Volume 2139 of LNCS., Springer (2001) 310–331

10. Paterson, K.G., Ristenpart, T., Shrimpton, T.: Tag size does matter: Attacks and proofs for the TLS record protocol. In Lee, D.H., Wang, X., eds.: ASIACRYPT 2011. Volume 7073 of LNCS., Springer (2011) 372–389
11. Jager, T., Kohlar, F., Schäge, S., Schwenk, J.: On the security of TLS-DHE in the standard model. In Safavi-Naini, R., Canetti, R., eds.: CRYPTO 2012. Volume 7417 of LNCS., Springer (2012) 273–293
12. Krawczyk, H., Paterson, K.G., Wee, H.: On the security of the TLS protocol: A systematic analysis. In Canetti, R., Garay, J.A., eds.: CRYPTO 2013, Part I. Volume 8042 of LNCS., Springer (2013) 429–448
13. Kohlar, F., Schäge, S., Schwenk, J.: On the security of TLS-DH and TLS-RSA in the standard model. Cryptology ePrint Archive, Report 2013/367 (2013) http://eprint.iacr.org/2013/367.
14. Li, Y., Schäge, S., Yang, Z., Kohlar, F., Schwenk, J.: On the security of the pre-shared key ciphersuites of TLS. In Krawczyk, H., ed.: PKC 2014. Volume 8383 of LNCS., Springer (2014) 669–684
15. Brzuska, C., Fischlin, M., Smart, N.P., Warinschi, B., Williams, S.C.: Less is more: Relaxed yet composable security notions for key exchange. International Journal of Information Security **12** (2013) 267–297
16. Bhargavan, K., Fournet, C., Kohlweiss, M., Pironti, A., Strub, P.Y.: Implementing TLS with verified cryptographic security. In: 2013 IEEE Symposium on Security and Privacy, IEEE Computer Society Press (2013) 445–459
17. Giesen, F., Kohlar, F., Stebila, D.: On the security of TLS renegotiation. In Sadeghi, A.R., Gligor, V.D., Yung, M., eds.: ACM CCS 13, ACM Press (2013) 387–398
18. Ray, M., Dispensa, S.: Renegotiating TLS (2009) http://extendedsubset.com/Renegotiating_TLS.pdf.
19. Mavrogiannopoulos, N., Vercauteren, F., Velichkov, V., Preneel, B.: A cross-protocol attack on the TLS protocol. In Yu, T., Danezis, G., Gligor, V.D., eds.: ACM CCS 12, ACM Press (2012) 62–72
20. Bergsma, F., Dowling, B., Kohlar, F., Schwenk, J., Stebila, D.: Multi-ciphersuite security of the Secure Shell (SSH) protocol. In Ahn, G.J., Yung, M., Li, N., eds.: ACM CCS 14, ACM Press (2014) 369–381
21. Bhargavan, K., Fournet, C., Kohlweiss, M., Pironti, A., Strub, P.Y., Zanella Béguelin, S.: Proving the TLS handshake secure (as it is). In Garay, J.A., Gennaro, R., eds.: CRYPTO 2014, Part II. Volume 8617 of LNCS., Springer (2014) 235–255
22. Rescorla, E., Ray, M., Dispensa, S., Oskov, N.: Transport Layer Security (TLS) renegotiation indication extension (2010) RFC 5746.
23. Möller, B., Langley, A.G.: TLS fallback Signaling Cipher Suite Value (SCSV) for preventing protocol downgrade attacks. https://tools.ietf.org/html/draft-ietf-tls-downgrade-scsv-05 (2015) Internet-Draft -05.
24. Jager, T., Kohlar, F., Schäge, S., Schwenk, J.: On the security of TLS-DHE in the standard model. Cryptology ePrint Archive, Report 2011/219 (2011) http://eprint.iacr.org/2011/219.
25. Bellare, M., Rogaway, P.: Entity authentication and key distribution. In Stinson, D.R., ed.: CRYPTO'93. Volume 773 of LNCS., Springer (1993) 232–249
26. Dowling, B., Stebila, D.: Modelling ciphersuite and version negotiation in the TLS protocol (full version). Cryptology ePrint Archive (2015)
27. Rescorla, E.: The Transport Layer Security (TLS) protocol version 1.3. https://tools.ietf.org/html/draft-ietf-tls-tls13-05 (2015) Internet-Draft -05.