Inductive Analysis of the Internet Protocol TLS

Lawrence C. Paulson
Computer Laboratory
University of Cambridge
Pembroke Street
Cambridge CB2 3QG
England

lcp@cl.cam.ac.uk

16 December 1997

Abstract

Internet browsers use security protocols to protect confidential messages. An inductive analysis of TLS (a descendant of SSL 3.0) has been performed using the theorem prover Isabelle. Proofs are based on higher-order logic and make no assumptions concerning beliefs or finiteness. All the obvious security goals can be proved; session resumption appears to be secure even if old session keys have been compromised. The analysis suggests modest changes to simplify the protocol.

TLS, even at an abstract level, is much more complicated than most protocols that researchers have verified. Session keys are negotiated rather than distributed, and the protocol has many optional parts. Nevertheless, the resources needed to verify TLS are modest. The inductive approach scales up.

Contents

1 Introduction

Internet commerce requires secure communications. To order goods, a customer typically sends credit card details. To order life insurance, the customer might have to supply confidential personal data. Internet users would like to know that such information is safe from eavesdropping or tampering.

Many Web browsers protect transmissions using the protocol SSL (Secure Sockets Layer). The client and server machines exchange nonces and compute session keys from them. Version 3.0 of SSL has been designed to correct a flaw of previous versions, where an attacker could induce the parties into choosing an unnecessarily weak cryptosystem. The latest version of the protocol is called TLS (Transport Layer Security) [1]; it closely resembles SSL 3.0.

Is TLS really secure? My proofs suggest that it is, but one should draw no conclusions without reading the rest of this report, which describes how the protocol was modelled and what properties were proved. I have analyzed a much simplified form of TLS; I assume hashing and encryption to be secure.

My abstract version of TLS is simpler than the concrete protocol, but it is still more complex than the protocols typically verified. We have not reached the limit of what can be analyzed formally.

The proofs were conducted using Isabelle/HOL [4], an interactive theorem prover for higher-order logic. They follow the inductive method [7], which has a clear semantics and treats infinite-state systems. Model-checking is not used, so there are no restrictions on the agent population, numbers of concurrent runs, etc.

The paper gives an overview of TLS (§2) and of the inductive method for verifying protocols (§3). It continues by presenting the Isabelle formalization of TLS (§4) and outlining some of the properties proved (§5). Finally, the paper discusses related work (§6) and concludes (§7).

2 Overview of TLS

A TLS *handshake* involves a *client*, such as a World Wide Web browser, and a Web *server*. Below, I refer to the client as A ('Alice') and the server as B ('Bob'), as is customary for authentication protocols, especially since C and S often have dedicated meanings in the literature.

At the start of a handshake, A contacts B, supplying a session identifier and nonce. In response, B sends another nonce and his public-key certificate (my model omits the other possibilities). Then A generates a *pre-master-secret*, which is a 48-byte random string, and sends it to B encrypted with his public key. A optionally sends a signed message to authenticate herself. Now, both parties calculate the *master-secret* M from the nonces and the pre-master-secret, using a secure pseudo-random-number function (PRF).

They calculate session keys from the nonces and M. Each session involves a pair of symmetric keys; A encrypts using one and B encrypts using the other. Before sending application data, both parties exchange **finished** messages to confirm all details of the handshake. Any tampering, for example involving early messages sent in clear, is detected at this point.

The full handshake outlined above is not always necessary. At some later time, A can resume a session by quoting an old session identifier along with a fresh nonce. If B is willing to resume the designated session, then he replies with a fresh nonce and both parties compute fresh session keys from these nonces and the stored master-secret, M. Both sides confirm this shorter run using **finished** messages.

My version of TLS leaves out details of record formats, cryptographic algorithms, etc.: they are irrelevant in an abstract analysis. Alert and failure messages are omitted: bad sessions are simply abandoned. I also omit the **server key exchange** message, which is used in anonymous sessions amongst other things.

Here are the handshake messages in detail, as I model them, along with comments about their relation to full TLS. Section numbers, such as tls§7.3, refer to the TLS specification [1]. In Fig. 1, dashed lines indicate optional parts.

$$\textbf{client hello} \quad A \to B : A, Na, Sid, Pa$$

The items in this message include the nonce Na, called **client random**, and the session identifier Sid. Item Pa is A's set of preferences for encryption and compression. For our purposes, all that matters is that both parties can detect if this value has been altered during transmission (tls§7.4.1.2).

$$\textbf{server hello} \quad B \to A : Nb, Sid, Pb$$

B, in his turn, replies with nonce Nb (**server random**) and his preferences Pb.

$$\textbf{server certificate} \quad B \to A : \mathsf{certificate}(B, Kb)$$

The server's public key, Kb, is delivered in a certificate signed by a trusted third party. (The TLS proposal (tls§7.4.2) specifies a chain of X.509v3 certificates, but I assume a single certification authority and omit lifetimes and similar details.) Making the certificate mandatory and eliminating the **server key exchange** message (tls§7.4.5) simplifies **server hello**. I leave **certificate request** (tls§7.4.4) implicit: A herself decides whether or not

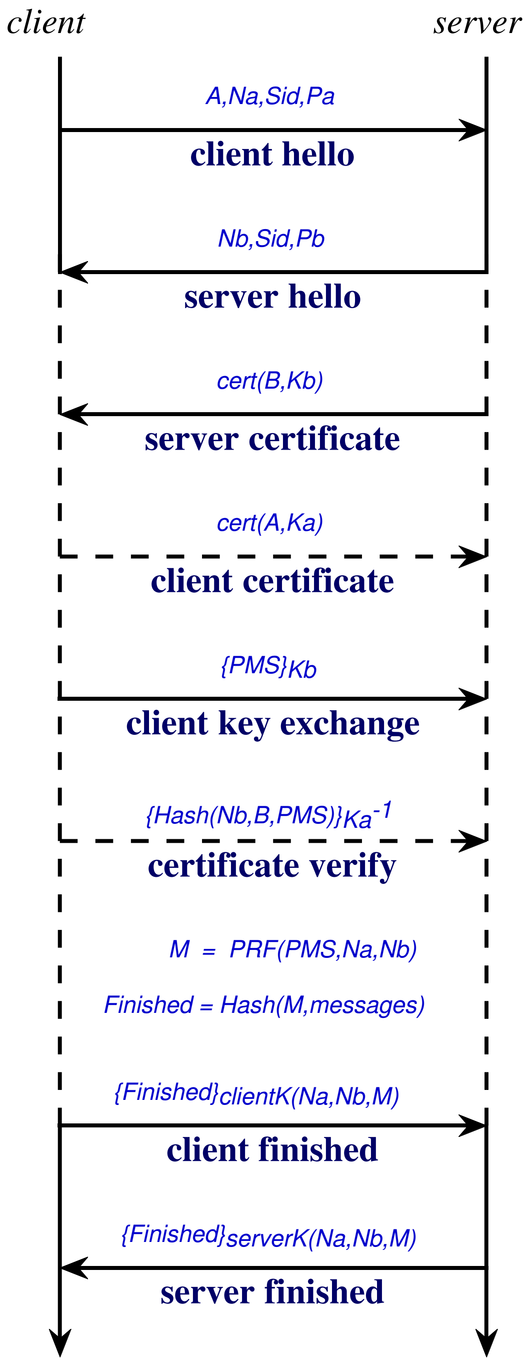

Figure 1: The TLS Handshake Protocol as Modelled

to send the optional messages **client certificate** and **certificate verify**.

client certificate*	$A \to B : \mathsf{certificate}(A, Ka)$
client key exchange	$A \to B : \{PMS\}_{Kb}$
certificate verify*	$A \to B : \{\mathsf{Hash}\{Nb, B, PMS\}\}_{Ka^{-1}}$

I do not model the possibility of arriving at the pre-master-secret via a Diffie-Hellman exchange (tls§7.4.7). Optional messages are starred (*) above; in **certificate verify**, A authenticates herself to B by signing the hash of some items relevant to the current session. The specification is that all handshake messages should be hashed, but my proofs suggest that only items Nb, B and PMS are essential.

client finished	$A \to B : \{Finished\}_{\mathsf{clientK}(Na,Nb,M)}$
server finished	$A \to B : \{Finished\}_{\mathsf{serverK}(Na,Nb,M)}$

Both parties compute the master-secret M from PMS, Na and Nb and compute *Finished* as the hash of M, Sid, Na, Pa, A, Nb, Pb, B. According to the specification (tls§7.4.9), M should be hashed with all previous handshake messages using PRF.

The symmetric key $\mathsf{clientK}(Na, Nb, M)$ is intended for client encryption, while $\mathsf{serverK}(Na, Nb, M)$ is for server encryption; each party decrypts using the other's key (tls§6.3). (The corresponding MAC secrets are implicit because my formal model assumes strong encryption.)

Once a party has received the other's **finished** message and compared it with her own, she is assured that both sides agree on all critical parameters, including M, Pa and Pb. Now she may begin sending confidential data. The SSL specification [2] erroneously states that she can send data immediately after sending her own **finished** message, before confirming these parameters.

For session resumption, the **hello** messages are the same. After checking that the session identifier is recent enough, the parties exchange **finished** messages and start sending application data.

On paper, then, session resumption does not involve any new message types. But in the model, four further events are involved. Each party stores the session parameters after a successful handshake and looks them up in order to resume a session.

I obtained the abstract protocol above by reverse engineering the TLS specification. This process took about two weeks. SSL must have originated in a message exchange described abstractly, in the same style as my description above, but I was unable to find any document describing TLS or SSL abstractly. If security protocols are to gain any trust, their design process must be transparent. The underlying abstract protocol should be exposed to public scrutiny. The concrete protocol should be presented as a faithful realization of the abstract one. Attacks against the abstract protocol should be dealt with by first correcting the abstract protocol and only then the concrete protocol. Designers should distinguish between attacks against the abstract message exchange and those against the concrete protocol, at the record layer or even at the byte level.

3 Proving Protocols Using Isabelle

Isabelle [4] is an interactive theorem prover supporting several formalisms, one of which is higher-order logic (HOL). Protocols can be modelled in Isabelle/HOL as inductive definitions. Isabelle's simplifier and classical reasoner automate large parts of the proofs.

A security protocol is modelled as the set of traces that could arise when a population of agents run it. Among the agents is a spy who controls some subset of them as well as the network itself. The population is infinite, and the number of interleaved sessions is unlimited. This section summarizes the approach, described in detail elsewhere [7].

Message are composed of agent names, nonces, keys, etc:

$\mathsf{Agent}\,A$	identity of an agent
$\mathsf{Number}\,N$	guessable number
$\mathsf{Nonce}\,N$	non-guessable number
$\mathsf{Key}\,K$	cryptographic key
$\mathsf{Hash}\,X$	hash of message X
$\mathsf{Crypt}\,K\,X$	encryption of X with key K
$\{\!\vert X_1, \ldots, X_n \vert\!\}$	concatenation of messages

Attributes such as non-guessable are related to the model of the spy. The protocol's **client random** and **server random** are modelled using Nonce because they are 28-byte random values, while **session identifiers** are modelled using Number because they may be any strings. TLS sends these items in clear, so whether they are guessable or not makes little difference to what can be proved. The pre-master-secret must be modelled as a nonce; we shall prove no security properties by assuming it can be guessed.

The model assumes strong encryption. Hashing is collision-free, and nobody can recover a message from its hash. Encrypted messages can neither be read nor changed without using the corresponding key. The protocol verifier makes such assumptions not because they are true but because making them true is the responsibility of the cryptographer.

Three operators are used to express security properties. Each maps a set H of messages to another such set.

- $\mathsf{parts}\,H$ is the set of message components potentially recoverable from H, including all encrypted data but excluding the bodies of hashed messages.

- $\mathsf{analz}\, H$ is the set of message components recoverable from H by means of decryption using keys available (recursively) in $\mathsf{analz}\, H$.
- $\mathsf{synth}\, H$ is the set of messages that could be expressed, starting from H and guessable items, using hashing, encryption and concatenation.

4 Formalizing the Protocol in Isabelle

TLS uses both public- and shared-key encryption. Each agent A has a private key $\mathsf{priK}\, A$ and public key $\mathsf{pubK}\, A$. The operators clientK and serverK create symmetric keys from a triple of nonces. Modelling the underlying pseudo-random-number generator causes some complications compared with the treatment of simple public-key protocols such as Needham-Schroeder [5].

The common properties of clientK and serverK are captured in the constant sessionK, which is merely assumed to be injective and to generate session keys.

```
consts
  sessionK         :: "(nat*nat*nat)*nat => key"
  clientK, serverK :: "nat*nat*nat => key"

rules
  inj_sessionK   "inj sessionK"
  isSym_sessionK "isSymKey (sessionK nonces)"
```

Now define

$$\begin{aligned} \mathsf{clientK}\, X &= \mathsf{sessionK}(X, 0) \\ \mathsf{serverK}\, X &= \mathsf{sessionK}(X, 1). \end{aligned}$$

It follows that the ranges of clientK and serverK are disjoint (there can be no collisions between the two).

Next come declarations of the constant tls to be a set of traces (lists of events) and of the pseudo-random function PRF. The latter has an elaborate definition in terms of the hash functions MD5 and SHA-1 (see tls§5). At the abstract level, we simply assume PRF to be injective.

```
consts
  PRF :: "nat*nat*nat => nat"
  tls :: "event list set"

rules
  inj_PRF       "inj PRF"
```

The inductive definition of tls comprises fifteen rules, compared with the usual six or seven for simpler protocols. The computational cost of proving theorems seems to be only linear in the number of rules, but it can be exponential in the complexity of a rule, for example if there is multiple encryption. Combining rules in order to reduce their number is therefore counterproductive.

Figure 2 presents the first three rules. The first two are common to all protocols. Rule *Nil* says that the empty trace is allowed. Rule *Fake* says that the spy may send, to any other agent B, any message he is able to invent given past traffic. A third rule, *SpyKeys*, augments *Fake* by allowing the spy to use the TLS-specific pseudo-random-number functions sessionK and PRF. Its form may look odd, but in conjunction with the spy's other powers, it allows him to apply sessionK and PRF to any three nonces previously available to him. It does not let him invert these functions, which we assume to be one-way. We could replace *SpyKeys* by defining a TLS version of the function synth, but we should then have to rework the underlying theory of messages, which is common to all protocols.

Nil
```
[] ∈ tls
```

Fake
```
[| evs ∈ tls;  B ≠ Spy;
   X ∈ synth (analz (spies evs)) |]
⟹ Says Spy B X  # evs ∈ tls
```

SpyKeys
```
[| evsSK ∈ tls;
   Says Spy B {|Nonce NA, Nonce NB, Nonce M|} ∈ set evsSK |]
⟹ Notes Spy {| Nonce (PRF(M,NA,NB)),
                 Key (sessionK((NA,NB,M),b)) |} # evsSK ∈ tls
```

Figure 2: Specifying TLS: Basic Rules

Figure 3 presents three rules for the **hello** messages. **Client hello** lets any agent A send the nonce Na, session identifier Sid and preferences Pa to any other agent, B. The assumptions

$$Na \notin \mathsf{used}\, evsCH \qquad Na \notin \mathsf{range}\,\mathsf{PRF}$$

state that Na is fresh and outside the range of the function PRF: distinct from all possible master-secrets. The latter assumption precludes the possibility that A might by chance choose a nonce identical to some master-secret. (The standard function used does not cope with master-secrets because they are calculated rather than chosen and never appear in traffic.) Both assumptions are reasonable because a 28-byte random string is highly unlikely to clash with any existing nonce or future master-secret.[1]

[1] The condition seems stronger than necessary. It refers to all conceivable master-secrets because there is no way of referring to the future of this run. As an alternative, a 'no coincidences' condition might be imposed later in the protocol, but the form it should take is not obvious.

Server hello is modelled similarly. It takes as precondition that B has received an instance of **Client hello**, purportedly from A.

The *Certificate* rule handles both **server certificate** and **client certificate**. Any agent may send his public-key certificate to any other agent. In the model, a certificate is simply an (agent, key) pair signed by the authentication server. Freshness of certificates, chains of certificates and similar details are not modelled.

```
constdefs
  certificate :: "[agent,key] => msg"
    "certificate A KA == Crypt (priK Server) {|Agent A, Key KA|}"
```

ClientHello

```
[| evsCH ∈ tls;  A ≠ B;  Nonce NA ∉ used evsCH;  NA ∉ range PRF |]
⟹ Says A B {|Agent A, Nonce NA, Number SID, Number PA|}
            # evsCH ∈ tls
```

ServerHello

```
[| evsSH ∈ tls;  A ≠ B;  Nonce NB ∉ used evsSH;  NB ∉ range PRF;
   Says A' B {|Agent A, Nonce NA, Number SID, Number PA|}
    ∈ set evsSH |]
⟹ Says B A {|Nonce NB, Number SID, Number PB|} # evsSH ∈ tls
```

Certificate

```
[| evsC ∈ tls;  A ≠ B |]
⟹ Says B A (certificate B (pubK B)) # evsC ∈ tls
```

Figure 3: Specifying TLS: **Hello** Messages

The next two rules are **client key exchange** and **certificate verify** (Fig. 4). Rule *ClientKeyExch* chooses a *PMS* that is fresh and differs from all master-secrets, like the nonces in the **hello** messages. It requires **server certificate** to have been received. No agent is allowed to know the true sender of a message, so *ClientKeyExch* might deliver the *PMS* to the wrong agent. Similarly, *CertVerify* might use the *Nb* value from the wrong instance of **server hello**. Security is not compromised because the run will fail in the **finished** messages.

ClientKeyExch not only sends the encrypted *PMS* to B but also stores it internally using the event $\mathsf{Notes}\, A\, \{\!\!|B, \mathit{PMS}|\!\!\}$. Other rules model A's referring to this note.[2] For instance, *CertVerify* states that if A chose *PMS* for B and has received a **server hello** message, then she may send a **certificate verify** message.

Next come the **finished** messages (Fig. 5). *ClientFinished* states that if A has sent **client hello** and has received a plausible instance of **server**

[2]With shared-key encryption, a ciphertext such as $\{\!\!|N|\!\!\}_{Kab}$ identifies the agents who know the plaintext, namely A and B. With public-key encryption, the ciphertext $\{\!\!|N|\!\!\}_{Kb}$ does not identify the sender; to model the sender's knowledge, a Notes event must be used.

```
ClientKeyExch
[| evsCX ∈ tls;  A ≠ B;  Nonce PMS ∉ used evsCX;  PMS ∉ range PRF;
   Says B' A (certificate B KB) ∈ set evsCX |]
⟹ Says A B (Crypt KB (Nonce PMS))
      # Notes A {|Agent B, Nonce PMS|}
      # evsCX ∈ tls

CertVerify
[| evsCV ∈ tls;  A ≠ B;
   Says B' A {|Nonce NB, Number SID, Number PB|} ∈ set evsCV;
   Notes A {|Agent B, Nonce PMS|} ∈ set evsCV |]
⟹ Says A B (Crypt (priK A) (Hash{|Nonce NB, Agent B, Nonce PMS|}))
              # evsCV ∈ tls
```

Figure 4: **Client key exchange** and **certificate verify**

hello and has chosen a *PMS* for B, then she can calculate the master-secret and send a **finished** message using her **client write key**. *ServerFinished* is analogous and may occur if B has received a **client hello**, sent a **server hello**, and received a **client key exchange** message.

That covers all the protocol messages, but the specification is not yet complete. Next come two rules to model agents' confirmation of a session (Fig. 6). Each agent, after sending its finished message and receiving a matching finished message apparently from its peer, records the session parameters to allow resumption. The references to *PMS* in the rules appear to contradict the protocol specification (tls§8.1): 'the pre-master-secret should be deleted from memory once the master-secret has been computed.' The purpose of those references is to restrict the rules to agents who actually know the secrets, as opposed to a spy who merely has replayed messages. They can probably be replaced by references to the master-secret, which the agents keep in memory. Complicating the model in this way brings no benefits: the loss of either secret is equally catastrophic.

Next come two rules for session resumption (Fig. 7). Like *clientFinished* and *serverFinished*, they refer to two previous hello messages. But instead of calculating the master-secret from a *PMS* just sent, they use the master-secret stored by *clientAccepts* or *serverAccepts* with the same session identifier. They calculate new session keys using the fresh nonces.

The final rule, *Oops*, models security breaches. Any session key, if used, may end up in the hands of the spy:

```
Oops
[| evso ∈ tls;  A ≠ Spy;
   Says A B (Crypt (sessionK((NA,NB,M),b)) X) ∈ set evso |]
⟹ Says A Spy (Key (sessionK((NA,NB,M),b))) # evso ∈ tls
```

Adding *Oops* to the model lets us prove that session resumption is safe even if the spy has obtained session keys from earlier sessions.

ClientFinished
```
[| evsCF ∈ tls;
   Says A  B {|Agent A, Nonce NA, Number SID, Number PA|}
    ∈ set evsCF;
   Says B' A {|Nonce NB, Number SID, Number PB|} ∈ set evsCF;
   Notes A {|Agent B, Nonce PMS|} ∈ set evsCF;
   M = PRF(PMS,NA,NB) |]
⟹ Says A B (Crypt (clientK(NA,NB,M))
             (Hash{|Number SID, Nonce M,
                    Nonce NA, Number PA, Agent A,
                    Nonce NB, Number PB, Agent B|}))
    # evsCF ∈ tls
```

ServerFinished
```
[| evsSF ∈ tls;
   Says A' B  {|Agent A, Nonce NA, Number SID, Number PA|}
    ∈ set evsSF;
   Says B  A  {|Nonce NB, Number SID, Number PB|} ∈ set evsSF;
   Says A' B (Crypt (pubK B) (Nonce PMS)) ∈ set evsSF;
   M = PRF(PMS,NA,NB) |]
⟹ Says B A (Crypt (serverK(NA,NB,M))
             (Hash{|Number SID, Nonce M,
                    Nonce NA, Number PA, Agent A,
                    Nonce NB, Number PB, Agent B|}))
    # evsSF ∈ tls
```

Figure 5: **Finished** messages

ClientAccepts

```
[| evsCA ∈ tls;
   Notes A {|Agent B, Nonce PMS|} ∈ set evsCA;
   M = PRF(PMS,NA,NB);
   X = Hash{|Number SID, Nonce M,
             Nonce NA, Number PA, Agent A,
             Nonce NB, Number PB, Agent B|};
   Says A  B (Crypt (clientK(NA,NB,M)) X) ∈ set evsCA;
   Says B' A (Crypt (serverK(NA,NB,M)) X) ∈ set evsCA |]
⟹
   Notes A {|Number SID, Agent A, Agent B, Nonce M|} # evsCA ∈ tls
```

ServerAccepts

```
[| evsSA ∈ tls;
   Says A' B (Crypt (pubK B) (Nonce PMS)) ∈ set evsSA;
   M = PRF(PMS,NA,NB);
   X = Hash{|Number SID, Nonce M,
             Nonce NA, Number PA, Agent A,
             Nonce NB, Number PB, Agent B|};
   Says B  A (Crypt (serverK(NA,NB,M)) X) ∈ set evsSA;
   Says A' B (Crypt (clientK(NA,NB,M)) X) ∈ set evsSA |]
⟹
   Notes B {|Number SID, Agent A, Agent B, Nonce M|} # evsSA ∈ tls
```

Figure 6: Agent acceptance events

ClientResume

```
[| evsCR ∈ tls;
   Says A  B {|Agent A, Nonce NA, Number SID, Number PA|}∈ set evsCR;
   Says B' A {|Nonce NB, Number SID, Number PB|} ∈ set evsCR;
   Notes A {|Number SID, Agent A, Agent B, Nonce M|} ∈ set evsCR |]
⟹ Says A B (Crypt (clientK(NA,NB,M))
             (Hash{|Number SID, Nonce M,
                    Nonce NA, Number PA, Agent A,
                    Nonce NB, Number PB, Agent B|}))
    # evsCR ∈ tls
```

ServerResume

```
[| evsSR ∈ tls;
   Says A' B {|Agent A, Nonce NA, Number SID, Number PA|}∈ set evsSR;
   Says B  A {|Nonce NB, Number SID, Number PB|} ∈ set evsSR;
   Notes B {|Number SID, Agent A, Agent B, Nonce M|} ∈ set evsSR |]
⟹ Says B A (Crypt (serverK(NA,NB,M))
             (Hash{|Number SID, Nonce M,
                    Nonce NA, Number PA, Agent A,
                    Nonce NB, Number PB, Agent B|})) # evsSR
     ∈ tls
```

Figure 7: Agent resumption events

Other security breaches could be modelled. The pre-master-secret might be lost to a cryptanalytic attack against the **client key exchange** message, and Wagner and Schneier suggest an attack that could reveal the master-secret [8, §4.7]. Loss of the *PMS* would compromise the entire session; it is hard to see what security goal could still be proved (in contrast, loss of a session key compromises that key alone). Recall that the spy already controls the network and an unknown number of agents.

The protocol, as modelled, is highly nondeterministic. As in TLS itself, some messages are optional (**client certificate**, **certificate verify**). Either client or server may be the first to commit to a session or to send a **finished** message. One party might attempt session resumption while the other runs the full protocol. Nothing in the rules above stops anyone from responding to any message repeatedly. Anybody can send a certificate to anyone else at any time.

Such nondeterminism is unacceptable in a real protocol, but it simplifies the model. Constraining a rule to follow some other rule or to apply at most once requires additional preconditions. A simpler model generally allows simpler proofs, and the additional traces it admits makes its theorems stronger.

5 Properties Proved of TLS

One difficulty in protocol verification is knowing what to prove. Protocol goals are usually stated informally at best, and the threat model (the type of attacker that is expected) is only hinted at. The TLS memo (tls§1) asserts 'three basic properties':

1. 'the peer's identity can be authenticated'
2. 'the negotiated secret is unavailable to eavesdroppers, and for any authenticated connection the secret cannot be obtained, even by an attacker who can place himself in the middle of the connection'[3]
3. 'no attacker can modify the negotiation communication without being detected by the parties'

I have proved formal versions of most of these properties along with many intermediate results.

5.1 Basic Lemmas

In the inductive method, results are of three sorts: possibility properties, regularity lemmas and secrecy theorems. Possibility properties merely exercise all the rules to check that the model protocol can run. For a simple protocol,

[3]Two distinct threat models are mentioned here. I model only the active attacker.

one possibility property suffices to show that message formats are compatible. For TLS, I proved four properties to check various paths through the main protocol, the **client verify** message, and session resumption.

Regularity lemmas assert properties that hold of all messages in the system. Nobody sends messages to himself and no private keys become compromised in any protocol step. From our specification of TLS, it is easy to prove that all certificates are valid. If $\mathsf{certificate}(B, K)$ appears in traffic then K is B's public key:

```
[| certificate B KB ∈ parts (spies evs);  evs ∈ tls |]
⟹ pubK B = KB
```

This property is overly strong, but adding false certificates seems pointless, since B might be under the spy's control anyway.

Many regularity lemmas are technical. Here are two typical ones. If a master-secret has appeared in traffic, then so has the underlying pre-master-secret. (Only the spy might send such a message.)

```
[| Nonce (PRF (PMS,NA,NB)) ∈ parts (spies evs);  evs ∈ tls |]
⟹ Nonce PMS ∈ parts (spies evs)
```

If a pre-master-secret is fresh, then no session key derived from it can either have been transmitted or used to encrypt.[4]

```
[| Nonce PMS ∉ parts (spies evs);
   K = sessionK((Na, Nb, PRF(PMS,NA,NB)), b);
   evs ∈ tls |]
⟹ Key K ∉ parts (spies evs) &
    (∀ Y. Crypt K Y ∉ parts (spies evs))
```

Client authentication, one of the protocol's goals, is an easily-proved regularity property. If **certificate verify** has been sent, apparently by A, then it really has been sent by A provided A is uncompromised (not controlled by the spy). Moreover, A has chosen the pre-master-secret that is hashed in **certificate verify**.

```
[| X ∈ parts (spies evs);
   X = Crypt KA⁻¹ (Hash{|nb, Agent B, pms|});
   certificate A KA ∈ parts (spies evs);
   evs ∈ tls;  A ∉ bad |]
⟹ Says A B X ∈ set evs
```

5.2 Secrecy Goals

Other protocol goals relate to secrecy. In the inductive method, secrecy theorems concern items that are available to some agents but not to others. Proving secrecy theorems seems always to require, as a lemma, some form

[4]The two properties must be proved in mutual induction because of interactions between the Fake and Oops rules.

of session key compromise theorem. Such theorems impose limits on the message components that can become compromised by the loss of a session key. For most protocols, we require that these components contain no session keys, but for TLS, what matters is that they contain no nonces. (Nonces are of critical importance because one of them is the pre-master-secret.) The theorem seems obvious; no honest agent encrypts nonces using session keys, and the spy can only send nonces that have already been compromised. However, its proof takes over 20 seconds to run. Such proofs typically involve large, though automatic, case analyses.

```
evs ∈ tls ⟹
Nonce N ∈ analz (insert (Key (sessionK z)) (spies evs)) =
(Nonce N ∈ analz (spies evs))
```

Other secrecy proofs follow easily from the session key compromise theorem, using induction and simplification. If the master-secret is secure from the spy then so are all session keys derived from it, except those lost by the Oops rule. In fact, session keys do not form part of any traffic.

```
[| ∀ A. Says A Spy (Key (sessionK((NA,NB,M),b))) ∉ set evs;
   Nonce M ∉ analz (spies evs);  evs ∈ tls |]
⟹ Key (sessionK((NA,NB,M),b)) ∉ parts (spies evs)
```

If A sends the **client key exchange** message to B, and both agents are uncompromised, then the pre-master-secret and master-secret will stay secret for ever.

```
[| Notes A {|Agent B, Nonce PMS|} ∈ set evs;
   evs ∈ tls;  A ∉ bad;  B ∉ bad |]
⟹ Nonce PMS ∉ analz (spies evs)

[| Notes A {|Agent B, Nonce PMS|} ∈ set evs;
   evs ∈ tls;  A ∉ bad;  B ∉ bad |]
⟹ Nonce (PRF(PMS,NA,NB)) ∉ analz (spies evs)
```

5.3 Finished Messages

The most important protocol goals concern authenticity of the **finished** message. If each party can know that the **finished** message just received indeed came from the expected agent, then they can compare the message components to confirm that no tampering has occurred. Naturally, these guarantees are conditional on both agents' being uncompromised.

The client's guarantee states that if A has chosen a pre-master-secret PMS for B and if any **finished** message is present that has been encrypted with a **server write key** derived from PMS, and if B has not given that session key to the spy (via *Oops*), then B himself has sent that message, and to A.

```
[| X = Crypt (serverK(Na,Nb,M))
             (Hash{|Number SID, Nonce M,
                    Nonce Na, Number PA, Agent A,
                    Nonce Nb, Number PB, Agent B|});
   M = PRF(PMS,NA,NB);
   X ∈ parts (spies evs);
   Notes A {|Agent B, Nonce PMS|} ∈ set evs;
   Says B Spy (Key (serverK(Na,Nb,M))) ∉ set evs;
   evs ∈ tls;  A ∉ bad;  B ∉ bad |]
⟹ Says B A X ∈ set evs
```

The server's guarantee is slightly different. If any message has been encrypted with a **client write key** derived from a given *PMS*—which we assume to have come from A—and if A has not given that session key to the spy, then A herself sent that message, and to B.

```
[| M = PRF(PMS,NA,NB);
   Crypt (clientK(Na,Nb,M)) Y ∈ parts (spies evs);
   Notes A {|Agent B, Nonce PMS|} ∈ set evs;
   Says A Spy (Key(clientK(Na,Nb,M))) ∉ set evs;
   evs ∈ tls;  A ∉ bad;  B ∉ bad |]
⟹ Says A B (Crypt (clientK(Na,Nb,M)) Y) ∈ set evs
```

The assumption (involving *Notes*) that A chose the *PMS* is essential. If A has not authenticated herself, then B must simply trust that she is present. By sending **certificate verify**, A can discharge this assumption:

```
[| Crypt KA⁻¹ (Hash{|nb, Agent B, Nonce PMS|}) ∈ parts (spies evs);
   certificate A KA ∈ parts (spies evs);
   evs ∈ tls;  A ∉ bad |]
⟹ Notes A {|Agent B, Nonce PMS|} ∈ set evs
```

Note that B's guarantee does not even require his inspecting the **finished** message. The very use of `clientK(Na,Nb,M)` is proof that the communication is from A to B. If we consider the analogous property for A, we find that using `serverK(Na,Nb,M)` only guarantees that the sender is B; in the absence of **certificate verify**, B has no evidence that the *PMS* came from A. If he sends **server finished** to somebody else then the session will fail, so there is no security breach. Still, changing **client key exchange** to include A's identity,

$$A \to B : \{\!|A, PMS|\!\}_{Kb},$$

would slightly strengthen the protocol and simplify the analysis.

The guarantees for **finished** messages apply to session resumption as well as to full handshakes. The inductive proofs cover all the rules that make up the definition of the constant tls, including those that model resumption.

5.4 Reasoning About Oops

The Oops rule makes the model more realistic. It causes security breaches to determine whether one breach leads to a cascade of others.

For the **finished** guarantees, the assumptions they make about oops are as weak as can be hoped for: that the very session key in question has not been lost by the only agent expected to use that key for encryption. The proofs of these guarantees appeal to a theorem discussed above (§5.2, which has a strong oops assumption: that *nobody* has lost the key. This general assumption is necessary because nothing in the theorem statement associates the key with any one agent. In their raw form, the **finished** guarantees inherit this general oops assumption; in effect, the agent getting the **finished** message must trust the entire agent population not to have lost the key. Such an assumption seems too strong; after all, the spy is among the population.

To weaken this assumption, further theorems are proved. If the rightful owner of the session key has not lost it, then nobody has. The rightful owner of a clientK is the creator of the underlying *PMS*.

```
[| evs ∈ tls;
   Says A Spy (Key(clientK(Na,Nb,PRF(PMS,NA,NB)))) ∉ set evs;
   Notes A {|Agent B, Nonce PMS|} ∈ set evs |]
⟹ ∀ A. Says A Spy (Key(clientK(Na,Nb,PRF(PMS,NA,NB)))) ∉ set evs
```

Under similar assumptions, the rightful owner of a serverK is B.

6 Related Work

Wagner and Schneier [8] analyze SSL 3.0 in detail. Much of their discussion concerns cryptanalytic attacks. Attempting repeated session resumptions causes the hashing of large amounts of known plaintext with the master-secret, perhaps revealing it (§4.7). They also report an attack against the Diffie-Hellman key-exchange messages, which my model omits (§4.4). Another attack involves deleting the **change cipher spec** message that (in a draft version of SSL 3.0) may optionally be sent before the **finished** message; TLS makes **change cipher spec** mandatory, and my model regards it as implicit in the **finished** exchange.

Wagner and Schneier's analysis appears not to use any formal tools. Their form of scrutiny, particularly concerning attacks against the underlying cryptosystems, will remain an essential complement to proving protocols at the abstract level. Most protocol proofs, including mine, assume strong encryption.

In his PhD thesis, Sven Dietrich analyses SSL 3.0 using the belief logic NCP (non-monotonic cryptographic protocols). NCP allows beliefs to be deleted; in the case of SSL, a session identifier is forgotten if the session fails. (In my formalization, session identifiers are not recorded until the initially session reaches a successful exchange of **finished** messages. Once recorded, they persist forever.) Recall that SSL allows both authenticated and unauthenticated sessions; Dietrich considers the latter and shows them to be secure against a passive eavesdropper. Although NCP is a formal logic, Dietrich appears to have generated his lengthy derivations by hand.

Mitchell et al. [3] apply model checking to a number of simple protocols derived from SSL 3.0. Most of the protocols are badly flawed (no nonces, for example) and the model checker finds many attacks. The final protocol still omits much of the detail of TLS, such as the distinction between the pre-master-secret and the other secrets computed from it. An 8-hour model-checking run found no attacks against the protocol in a system comprising two clients and one server.

7 Conclusions

The inductive method has many advantages. Its clear semantic framework, based on the actions agents can perform, has few of the peculiarities of belief logics. Proofs cover systems of infinite size, with no limits on the number of simultaneous or resumed sessions. Isabelle's automatic tools allow the proofs to be generated with a moderate effort, and they run fast. The full TLS proof script runs in under 9 minutes.[5]

The analysis took about six weeks, but I spent two weeks of that trying to abstract a protocol specification from various Internet-drafts. Thus, a more disciplined approach to protocol design would have reduced the effort by one third. Designers should regard the abstract message exchange as part of the specification and justify its byte-level realization against it. In any event, the effort needed to analyze the protocol is much less than that needed to implement it.

In one sense, the outcome is unexciting, All the expected security goals were proved; no attacks turned up. This outcome might be expected in a protocol already so thoroughly examined. No unusual lines of reasoning were required to establish the results, unlike the case of the Yahalom protocol [6]. The proofs did yield some insights into TLS, such as the possibility of strengthening **client key exchange** by including A's identity (§5).

In several places, the protocol requires computing the hash of 'all previous handshake messages.' There is obviously much redundancy, and the requirement is ambiguous too; the specification is sprinkled with remarks that certain routine messages or components should not be hashed. One such message, **change cipher spec**, was thereby omitted and later turned out to be essential [8]. I suggest, therefore, that hashes should be computed not over everything but over selected items that the protocol designer

[5]On a Sun SuperSPARC model 61.

requires to be confirmed. An inductive analysis can help in selecting the critical message components. The TLS security analysis (tls§F.1.1.2) states that the critical components of the hash in **certificate verify** are the server's name and nonce, but my proofs suggest that the pre-master-secret is also necessary.[6]

Once session keys have been established, the parties have a secure channel upon which they must run a reliable communication protocol. Martín Abadi tells me that the part of TLS concerned with communication should also be verified, since this aspect of SSL once contained errors. I have considered only the security aspects of TLS. The communication protocol could be verified separately, assuming an unreliable medium rather than an enemy. My proofs assume that application data does not contain secrets associated with TLS sessions, such as keys and master-secrets; if it does, then one security breach could lead to many others.

The main interest of this work lies in the modelling of TLS, especially its use of pseudo-random number generators. The creation of the master-secret and session keys is modelled by postulating the injective functions PRF and $\mathsf{sessionK}$. The enemy is given the power to apply these functions. Other agents are assumed to choose nonces outside the range of PRF: in other words, nobody accidentally guesses any master-secret.

Previous verification efforts have largely focussed on simple key-distribution protocols. It is now clear that realistic protocols are well within our grasp.

Acknowledgement. Martín Abadi encouraged me to look at TLS, helped me understand its specification and referred me to related work. James Margetson pointed out some simplifications to the model. Clemens Ballarin commented on a draft. The research was funded by the EPSRC, grants GR/K77051 'Authentication Logics' and GR/K57381 'Mechanizing Temporal Reasoning.'

References

[1] Tim Dierks and Christopher Allen. The TLS protocol: Version 1.0, November 1997. Internet-draft `draft-ietf-tls-protocol-05.txt`.

[2] Alan O. Freier, Philip Karlton, and Paul C. Kocher. The SSL protocol version 3.0.

[3] John C. Mitchell, Vitaly Shmatikov, and Ulrich Stern. Finite-state analysis of SSL 3.0 and related protocols. In Hilarie Orman and

[6] My formalization hashes message components rather than messages for simplicity. It is vulnerable to an attack in which the spy intercepts **certificate verify**, downgrading the session so that the client appears to be unauthenticated.

Catherine Meadows, editors, *Workshop on Design and Formal Verification of Security Protocols*. DIMACS, September 1997.

[4] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer, 1994. LNCS 828.

[5] Lawrence C. Paulson. Mechanized proofs of security protocols: Needham-Schroeder with public keys. Technical Report 413, Computer Laboratory, University of Cambridge, January 1997.

[6] Lawrence C. Paulson. On two formal analyses of the Yahalom protocol. Technical Report 432, Computer Laboratory, University of Cambridge, July 1997.

[7] Lawrence C. Paulson. Proving properties of security protocols by induction. In *10th Computer Security Foundations Workshop*, pages 70–83. IEEE Computer Society Press, 1997.

[8] David Wagner and Bruce Schneier. Analysis of the SSL 3.0 protocol. On the Internet at `http://www.cs.berkeley.edu/~daw/ssl3.0.ps`, 1996.