Inductive Analysis of the Internet Protocol TLS

Lawrence C. Paulson, University of Cambridge

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 proofs suggest minor changes to simplify the analysis.

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: six man-weeks of effort and three minutes of processor time.

Categories and Subject Descriptors: F.3.1 [**Theory of Computation**]: Logics and Meanings of Programs—*Specifying and Verifying and Reasoning about Programs* ; C.2.2 [**Computer-Communication Networks**]: Network Protocols—*Protocol Verification*

General Terms: Security, Verification

Additional Key Words and Phrases: TLS, authentication, proof tools, inductive method, Isabelle

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 alteration.

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, the *cipher-suite rollback attack*, whereby an intruder could get the parties to adopt a weak cryptosystem [Wagner and Schneier 1996]. The latest version of the protocol is called TLS (Transport Layer Security) [Dierks and Allen 1999]; it closely resembles SSL 3.0.

Is TLS really secure? My proofs suggest that it is, but one should draw no

The research was funded by the U.K.'s Engineering and Physical Sciences Research Council, grants GR/K77051 'Authentication Logics' and GR/K57381 'Mechanizing Temporal Reasoning.'
Address: Computer Laboratory, University of Cambridge, Cambridge CB2 3QG, England
email lcp@cl.cam.ac.uk

conclusions without reading the rest of this paper, 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 [Paulson 1994], an interactive theorem prover for higher-order logic. They use the inductive method [Paulson 1998], which has a simple 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 other possibilities). Then A generates a *pre-master-secret*, 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 master-secret. 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 and to check that cleartext parts of messages have not been altered.

A full handshake 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. Both parties compute fresh session keys from these nonces and the stored master-secret, M. Both sides confirm this shorter run using **finished** messages.

TLS is highly complex. My version leaves out many details for the sake of simplicity:

—Record formats, field widths, cryptographic algorithms, etc. are irrelevant in an abstract analysis.

—Alert and failure messages are unnecessary because bad sessions can simply be abandoned.

—The **server key exchange** message allows anonymous sessions among other things, but it is not an essential part of the protocol.

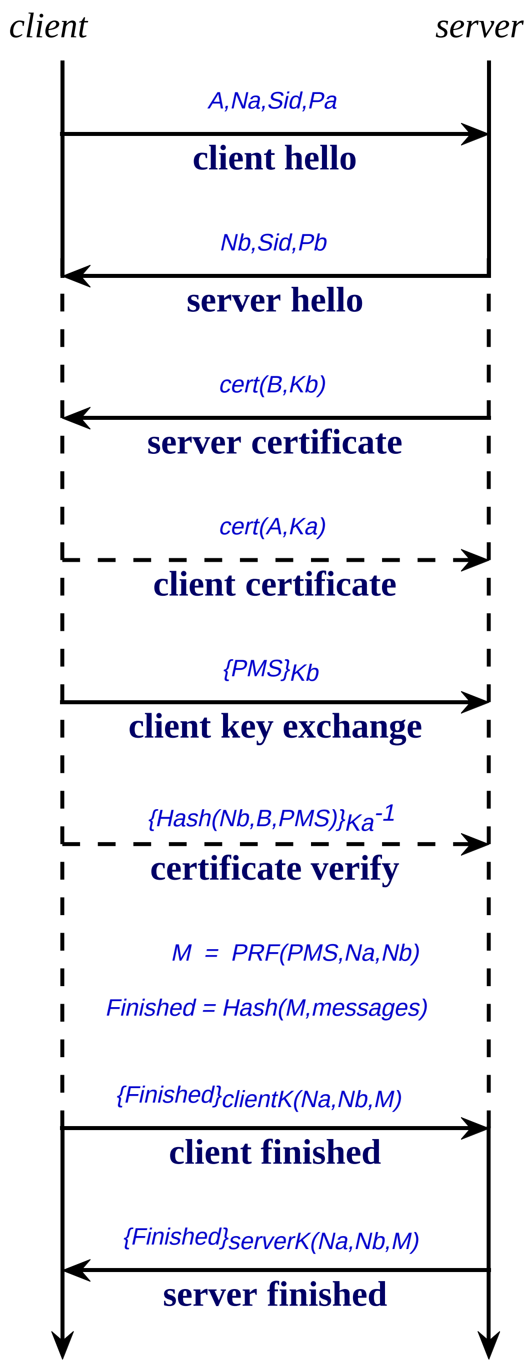

Fig. 1. The TLS Handshake Protocol as Modelled

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 [Dierks and Allen 1999]. 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; due to export controls, for example, some clients cannot support certain encryption methods. For our purposes, all that matters is that both parties can detect if Pa has been altered during transmission (tls§7.4.1.2).

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

Agent B, in his turn, replies with his nonce Nb (**server random**). He repeats the session identifier and returns as Pb his cryptographic preferences, selected from Pa.

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

The server's public key, Kb, is delivered in a certificate signed by a trusted third party. (The TLS proposal (tls§7.4.2) says it is 'generally an X.509v3 certificate.' I assume a single certification authority and omit lifetimes and similar details.) Making the certificate mandatory and eliminating the **server key exchange** message (tls§7.4.3) simplifies **server hello**. I leave **certificate request** (tls§7.4.4) implicit: A herself decides whether or not to send the optional messages **client certificate** and **certificate verify**.

$$\begin{array}{ll}
\textbf{client certificate*} & A \to B : \mathsf{certificate}(A, Ka) \\
\textbf{client key exchange} & A \to B : \{\!|PMS|\!\}_{Kb} \\
\textbf{certificate verify*} & A \to B : \{\!|\mathsf{Hash}\{\!|Nb, B, PMS|\!\}|\!\}_{Ka^{-1}}
\end{array}$$

For simplicity, I do not model the possibility of arriving at the pre-master-secret via a Diffie-Hellman exchange (tls§7.4.7.2). 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 states that all handshake messages should be hashed, but my proofs suggest that only Nb, B and PMS are essential.

$$\begin{array}{ll}
\textbf{client finished} & A \to B : \{\!|Finished|\!\}_{\mathsf{clientK}(Na,Nb,M)} \\
\textbf{server finished} & A \to B : \{\!|Finished|\!\}_{\mathsf{serverK}(Na,Nb,M)}
\end{array}$$

Both parties compute the master-secret M from PMS, Na and Nb and compute $Finished$ as the hash of Sid, M, 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. My formalization hashes message components rather than messages in order to simplify the inductive definition. It is vulnerable to an attack in which the spy intercepts **certificate verify**, downgrading the session so that the client appears to be unauthenticated.

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 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 and the preferences Pa and Pb. Now she may begin sending confidential data. The SSL specification [Freier et al. 1996] erroneously states that she can send data immediately after sending her own **finished** message, before confirming these parameters; there she takes a needless risk, since an attacker may have changed the preferences to request weak encryption. This is the cipher-suite rollback attack, precisely the one that the **finished** messages are intended to prevent.

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 when resuming a session.

3. PROVING PROTOCOLS USING ISABELLE

Isabelle [Paulson 1994] 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 [Paulson 1998].

3.1 Messages

Messages 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
$\{\!\{X_1, \ldots, X_n\}\!\}$	concatenation of messages

Attributes such as non-guessable concern 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. Moreover, reasoning about a cryptosystem such as DES down to the bit level is infeasible. However, this is a weakness of the method: certain combinations of protocols and encryption methods can be vulnerable [Ryan and Schneider 1998].

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 (assuming all ciphers could be broken).

—$\mathsf{analz}\, H$ 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.

3.2 Traces

A trace is a list of *events* such as $\mathsf{Says}\, A\, B\, X$, meaning '$A$ sends message X to B,' or $\mathsf{Notes}\, A\, X$, meaning 'A stores X internally.' Each trace is built in reverse order by prefixing ('consing') events to the front of the list, where # is the 'cons' operator.

The set bad comprises those agents who are under the spy's control.

The function spies yields the set of messages the spy can see in a trace: all messages sent across the network and the internal notes and private keys of the bad agents.

$$
\begin{aligned}
\mathsf{spies}\,((\mathsf{Says}\, A\, B\, X) \,\#\, evs) &= \{X\} \cup \mathsf{spies}\, evs \\
\mathsf{spies}\,((\mathsf{Notes}\, A\, X) \,\#\, evs) &= \begin{cases} \{X\} \cup \mathsf{spies}\, evs & \text{if } A \in \mathsf{bad} \\ \mathsf{spies}\, evs & \text{otherwise} \end{cases}
\end{aligned}
$$

The set $\mathsf{used}\, evs$ includes the parts of all messages in the trace, whether they are visible to other agents or not. Now $Na \notin \mathsf{used}\, evs$ expresses that Na is fresh with respect to the trace evs.

$$
\begin{aligned}
\mathsf{used}\,((\mathsf{Says}\, A\, B\, X) \,\#\, evs) &= \mathsf{parts}\{X\} \cup \mathsf{used}\, evs \\
\mathsf{used}\,((\mathsf{Notes}\, A\, X) \,\#\, evs) &= \mathsf{parts}\{X\} \cup \mathsf{used}\, evs
\end{aligned}
$$

4. FORMALIZING THE PROTOCOL IN ISABELLE

With the inductive method, each protocol step is translated into a rule of an inductive definition. A rule's premises describe the conditions under which the rule may apply, while its conclusion adds new events to the trace. Each rule allows a protocol step to occur but does not force it to occur—just as real world machines crash and messages get intercepted. The inductive definition has further rules to model intruder actions, etc.

For TLS, the inductive definition 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.

4.1 Basic Constants

TLS uses both public-key and shared-key encryption. Each agent A has a private key $\mathsf{priK}\,A$ and a 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 [Paulson 1998].

The common properties of clientK and serverK are captured in the function sessionK, which is assumed to be an injective (collision-free) source of session keys. In an Isabelle theory file, functions are declared as constants that have a function type. Axioms about them can be given using a `rules` section.

```
datatype role = ClientRole | ServerRole
consts
  sessionK         :: "(nat*nat*nat) * role => key"
  clientK, serverK :: "nat*nat*nat => key"
rules
  inj_sessionK   "inj sessionK"
  isSym_sessionK "isSymKey (sessionK nonces)"
```

The enumeration type, role, indicates the use of the session key. We ensure that clientK and serverK have disjoint ranges (no collisions between the two) by defining

$$\mathsf{clientK}\,X = \mathsf{sessionK}(X,\ \mathsf{ClientRole})$$
$$\mathsf{serverK}\,X = \mathsf{sessionK}(X,\ \mathsf{ServerRole}).$$

We must also declare the pseudo-random function PRF. In the real protocol, PRF 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"
```

We have also declared the constant tls to be the set of possible traces in a system running the protocol. The inductive definition of tls specifies it to be the least set of traces that is closed under the rules supplied below. A trace belongs to tls only if it can be generated by finitely many applications of the rules. Induction over tls amounts to considering every possible way that a trace could have been extended.

4.2 The Spy

Figure 2 presents the first three rules, two of which are standard. Rule *Nil* allows the empty trace. Rule *Fake* says that the spy may invent messages using past traffic and send them to any other agent. A third rule, *SpyKeys*, augments *Fake* by letting the spy use the TLS-specific functions sessionK and PRF. 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; however, we should then have to rework the underlying theory of messages, which is common to all protocols.

```
Nil
[] ∈ tls

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


SpyKeys
[| evsSK ∈ tls;
   {|Nonce NA, Nonce NB, Nonce M|} ⊆ analz (spies evsSK) |]
⟹ Notes Spy {| Nonce (PRF(M,NA,NB)),
                 Key (sessionK((NA,NB,M),role)) |} # evsSK ∈ tls
```

Fig. 2. Specifying TLS: Basic Rules

4.3 Hello Messages

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. **Server hello** is modelled similarly. Its precondition is that B has received a suitable instance of **Client hello**.

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

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

Fig. 3. Specifying TLS: **Hello** Messages

In **Client hello**, the assumptions $Na \notin \mathsf{used}\, evsCH$ and $Na \notin \mathsf{range\,PRF}$ state that Na is fresh and distinct from all possible master-secrets. The latter assumption precludes the possibility that A might choose a nonce identical to some master-secret. (The standard function used does not cope with master-secrets because they 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. Still, the condition seems stronger than necessary. It refers to all conceivable master-secrets because there is no way of referring to one single future. As an alternative, a 'no coincidences' condition might be imposed later in the protocol, but the form it should take is not obvious; if it is wrong, it might exclude realistic attacks.

The *Certificate* rule handles both **server certificate** and **client certificate**. It is more liberal than real TLS, for any agent may send his public-key certificate to any other agent. A certificate is represented by an (agent, key) pair signed by the authentication server. Freshness of certificates and other details are not modelled.

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

4.4 Client Messages

The next two rules concern **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
[| evsCX ∈ tls;  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;
   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
```

Fig. 4. **Client key exchange** and **certificate verify**

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

In my initial work on TLS, I modelled A's knowledge by referring to the event of her sending $\{\!|PMS|\!\}_{Kb}$ to B. However, this approach did not correctly model the sender's knowledge: the spy can intercept and send the ciphertext $\{\!|PMS|\!\}_{Kb}$ without knowing *PMS*. (The approach does work for shared-key encryption. A ciphertext such as $\{\!|PMS|\!\}_{Kab}$ identifies the agents who know the plaintext, namely A and B.) I discovered this anomaly when a proof failed. The final proof state indicated that the spy could gain the ability to send **client finished** merely by replaying A's message $\{\!|PMS|\!\}_{Kb}$.

Anomalies like this one can creep into any formalization. The worst are those that make a theorem hold vacuously, for example by mis-stating a precondition. There is no remedy but constant vigilance, noticing when a result is too good to be true or is proved too easily. We must also check that the assumptions built into the model, such as strong encryption, reasonably match the protocol's operating environment.

4.5 Finished Messages

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

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
```

Fig. 5. **Finished** messages

4.6 Session Resumption

That covers all the protocol messages, but the specification is not 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. 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 references to *PMS* in the *Accepts* rules appear to contradict the protocol specification (tls§8.1): 'the pre-master-secret should be deleted from memory once

```
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;  A ≠ B;
   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
```

Fig. 6. Agent acceptance events

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 (recall the comment at the end of §4.4). They can probably be replaced by references to the master-secret, which the agents keep in memory. We would have to add further events to the inductive definition. Complicating the model in this way brings no benefits: the loss of either secret is equally catastrophic.

Four further rules (omitted here) model agents' confirmation of a session and a subsequent session resumption.

4.7 Security Breaches

The final rule, *Oops*, models security breaches. Any session key, if used, may end up in the hands of the spy. Session resumption turns out to be safe even if the spy has obtained session keys from earlier sessions.

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

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 [1996, §4.7] suggest a strategy for discovering the master-secret. 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 too liberal and is highly nondeterministic. As in TLS itself, some messages are optional (**client certificate**, **certificate verify**).

```
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
```

Fig. 7. Agent resumption events

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. Safety theorems proved under a permissive regime will continue to hold under a strict one.

5. PROPERTIES PROVED OF TLS

One difficulty in protocol verification is knowing what to prove. Protocol goals are usually stated informally. The TLS memo states 'three basic properties' (tls§1):

(1) 'The peer's identity can be authenticated using ... public key cryptography'
(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) 'no attacker can modify the negotiation communication without being detected by the parties'

Authentication can mean many things [Gollmann 1996]; it is a pity that the memo does not go into more detail. I have taken 'authenticated connection' to mean one in which both parties use their private keys. My model allows A to be unauthenticated, since **certificate verify** is optional. However, B must be authenticated: the model does not support Diffie-Hellman, so Kb^{-1} must be used to decrypt **client key exchange**. Against an active intruder, an unauthenticated connection is vulnerable to the usual man-in-the-middle attack. Since the model does not support unauthenticated connections, I cannot investigate whether they are secure against passive eavesdroppers.

Some of the results discussed below relate to authentication. A pair of honest agents can establish the master-secret securely and use it to generate uncompromised session keys. Session resumption is secure even if previous session keys from that session have been compromised.

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, 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 traffic. For example, no protocol step compromises a private key. From our specification of TLS, it is easy to prove that all certificates are valid. (This property is overly strong, but adding false certificates seems pointless: B might be under the spy's control anyway.) If $\mathsf{certificate}(B, K)$ appears in traffic, then K really is B's public key:

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

The set $\mathsf{parts}(\mathsf{spies}\, evs)$ includes the components of all messages that have been sent; in the inductive method, regularity lemmas often mention this set. Sometimes the lemmas merely say that events of a particular form never occur.

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.[1]

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

Client authentication, one of the protocol's goals, is easily proved. 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
```

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

5.2 Secrecy Goals

Other goals of the protocol relate to secrecy: certain items are available to some agents but not to others. They are usually the hardest properties to establish. With the inductive method, they seem always to require, as a lemma, some form of *session key compromise theorem.* This theorem imposes limits on the message components that can become compromised by the loss of a session key. Typically we require that these components contain no session keys, but for TLS, they must 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 seven seconds to run. Like other secrecy proofs, it involves a large, though automatic, case analysis.

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

Note that $\mathsf{insert}\, x\, A$ denotes $\{x\} \cup A$. The set $\mathsf{analz}(\mathsf{spies}\, evs)$ includes all message components available to the spy, and likewise $\mathsf{analz}(\{K\} \cup \mathsf{spies}\, evs)$ includes all message components that the spy could get with the help of key K. The theorem states that session keys do not help the spy to learn new nonces.

Other secrecy proofs follow easily from the session key compromise theorem, using induction and simplification. Provided A and B are honest, the client's session key will be secure unless A herself gives it to the spy, using Oops.

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

An analogous theorem holds for the server's session key. However, the server cannot check the Notes assumption; see §5.3.2.

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

```
[| 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

Other 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. These components include the cryptographic preferences, which an intruder might like to downgrade. Naturally, the guarantees are conditional on both agents' being uncompromised.

5.3.1 *Client's guarantee.* The client's guarantee has several preconditions. The client, A, has chosen a pre-master-secret *PMS* for B. The traffic contains a **finished** message encrypted with a **server write key** derived from *PMS*. The server, B, has not given that session key to the spy (via *Oops*). The guarantee then states that 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
```

One of the preconditions may seem to be too liberal. The guarantee applies to any occurrence of the **finished** message in traffic, but it is needed only when A has received that message. The form shown, expressed using parts(spies *evs*), streamlines the proof; in particular, it copes with the spy's replaying a **finished** message concatenated with other material. It is well known that proof by induction can require generalizing the theorem statement.

5.3.2 *Server's guarantee.* 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 the client has not authenticated herself, then B knows nothing about her true identity and must trust that she is indeed A. By sending **certificate verify**, the client can discharge the Notes 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
```

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. At present, the proof scripts include theorems for A's association of PMS with B, and weaker theorems for B's knowledge of PMS. With the suggested change, the weaker theorems could probably be discarded.

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 Security Breaches

The Oops rule makes the model much more realistic. It allows session keys to be lost to determine whether the protocol is robust: one security breach should not lead to a cascade of others. Sometimes a theorem holds only if certain Oops events are excluded, but Oops conditions should be weak. For the **finished** guarantees, the conditions they impose on Oops events are as weak as could 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.

6. RELATED WORK

Wagner and Schneier [1996] 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, which could lead to a way of 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.

In his PhD thesis, Dietrich [1997] 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 initial 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, Shmatikov, and Stern [1997] 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 eight-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 semantic framework, based on the actions agents can perform, has few of the peculiarities of belief logics. Proofs impose 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 150 seconds on a 300Mhz Pentium.

I obtained the abstract message exchange given in §2 by reverse engineering the TLS specification. This process took about two weeks, one-third of the time spent on this verification. SSL must have originated in such a message exchange, but I could not find one in the literature. If security protocols are to be trusted, 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. Designers should distinguish between attacks against the abstract message exchange and those against the concrete protocol.

All the expected security goals were proved: no attacks were found. This unexciting outcome might be expected in a protocol already so thoroughly examined. No unusual lines of reasoning were required, unlike the proofs of the Yahalom protocol [Paulson] and Kerberos IV [Bella and Paulson 1998]; we may infer that TLS is well-designed. The proofs did yield some insights into TLS, such as the possibility of strengthening **client key exchange** by including A's identity (§5). The main interest of this work lies in the modelling of TLS, especially its use of pseudo-random number generators.

The protocol takes the *explicitness principle* of Abadi and Needham [1996] to an extreme. In several places, it requires computing the hash of 'all preceding 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 was found to be essential [Wagner and Schneier 1996]. I suggest, then, that hashes should be computed not over everything but over selected items that the protocol designer 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.

Once session keys have been established, the parties have a secure channel upon which they must run a reliable communication protocol. Abadi tells me that the TLS *application data protocol* should also be examined, since this part of SSL once contained errors. I have considered only the TLS *handshake protocol*, where session keys are negotiated. Ideally, the application data protocol should 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.

Previous verification efforts have largely focussed on small protocols of academic interest. It is now clear that realistic protocols can be analyzed too, almost as a matter of routine. For protocols intended for critical applications, such an analysis should be required as part of the certification process.

ACKNOWLEDGMENTS

Martín Abadi introduced me to TLS and identified related work. James Margetson pointed out simplifications to the model. The referees and Clemens Ballarin made useful comments.

REFERENCES

Abadi, M. and Needham, R. 1996. Prudent engineering practice for cryptographic protocols. *IEEE Trans. Softw. Eng. 22*, 1 (Jan.), 6–15.

Bella, G. and Paulson, L. C. 1998. Kerberos version IV: Inductive analysis of the secrecy goals. In J.-J. Quisquater, Y. Deswarte, C. Meadows, and D. Gollmann Eds., *Computer Security — ESORICS 98*, LNCS 1485 (1998), pp. 361–375. Springer.

Dierks, T. and Allen, C. 1999. The TLS protocol: Version 1.0. Request for Comments: 2246, on the Internet at `ftp://ftp.isi.edu/in-notes/rfc2246.txt`.

Dietrich, S. 1997. *A Formal Analysis of the Secure Sockets Layer Protocol*. Ph. D. thesis, Adelphi University, Garden City, New York. Department of Mathematics and Computer Science.

Freier, A. O., Karlton, P., and Kocher, P. C. 1996. The SSL protocol version 3.0. Internet-draft `draft-freier-ssl-version3-02.txt`.

Gollmann, D. 1996. What do we mean by entity authentication? In *Symposium on Security and Privacy* (1996), pp. 46–54. IEEE Computer Society.

Mitchell, J. C., Shmatikov, V., and Stern, U. 1997. Finite-state analysis of SSL 3.0 and related protocols. In H. Orman and C. Meadows Eds., *Workshop on Design and Formal Verification of Security Protocols* (Sept. 1997). DIMACS.

Paulson, L. C. 1994. *Isabelle: A Generic Theorem Prover*. Springer. LNCS 828.

Paulson, L. C. Relations between secrets: Two formal analyses of the Yahalom protocol. *Journal of Computer Security*. in press.

Paulson, L. C. 1998. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security 6*, 85–128.

Ryan, P. Y. A. and Schneider, S. A. 1998. An attack on a recursive authentication protocol: A cautionary tale. *Information Processing Letters 65*, 1 (Jan.), 7–10.

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