Finite-State Analysis of SSL 3.0 and Related Protocols

John C. Mitchell Vitaly Shmatikov Ulrich Stern

Computer Science Department, Stanford University, Stanford, CA 94305

{mitchell, shmat, uli.stern}@cs.stanford.edu

August 1, 1997

1 Introduction

In previous work [8], a general-purpose finite-state analysis tool, Murφ [2], has been successfully applied to the verification of small protocols such as Needham-Schroeder, Kerberos, and TMN. In an effort to understand the difficulties involved in analyzing larger and more complex protocols, we use Murφ to analyze the SSL 3.0 handshake protocol. The choice of SSL 3.0 was determined by its size and relative complexity as well as its current popularity as a de facto standard for secure Internet communication.

In the process of studying the feasibility of SSL 3.0 verification by exhaustive state enumeration, we develop a "rational reconstruction" of the protocol. More specifically, after initially attempting to familiarize ourselves with the handshake protocol, we found that we could not easily identify the purpose of certain message fields (version number, nonce, etc.) in some steps of the protocol. Therefore, we set out to use our analysis tool to identify, for each field, an attack that could occur if that field were omitted from the protocol. The result of our study is an incremental presentation of the protocol, beginning with a simple, intuitive, and flawed protocol exchanging the required data, and culminating in a close approximation of the actual SSL 3.0 handshake protocol. Our final protocol does not have any errors identifiable by our exhaustive finite-state analysis.

In the process of conducting this study, we uncovered the major problems with SSL 2.0 that motivated the design of SSL 3.0. We are currently investigating various anomalies related to protocol resumption.

Another formal analysis of SSL 3.0, using a logic formalizing concepts of knowledge and belief, appears in [1]. We did not become aware of this other study until after we had completed our analysis.

This work was supported in part by the Defense Advanced Research Projects Agency through NASA contract NAG-2-891, the National Science Foundation through grants CCR-9303099 and CCR-9629754, and a fellowship to Shmatikov from the Hertz Foundation. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, NASA, NSF or the US Government.

2 Outline of the methodology

Our general methodology for modeling security protocols in Murφ is outlined in our previous paper [8]. The basic approach is similar to the CSP approach to model checking of cryptographic protocols outlined in [7, 9]. However, there are some differences between Murφ and FDR.

2.1 The Murφ verification system

Murφ [2] is a protocol verification tool that has been successfully applied to several industrial protocols, especially in the domains of multiprocessor cache coherence protocols and multiprocessor memory models [3, 11, 12].

To use Murφ for verification, one has to model the protocol in the Murφ language and augment this model with a specification of the desired properties. The Murφ system automatically checks, by explicit state enumeration, if all reachable states of the model satisfy the given specification. For the state enumeration, either breadth-first or depth-first search can be selected. Reached states are stored in a hash table to avoid redundant work when a state is revisited. The memory available for this hash table typically determines the largest tractable problem.

The Murφ language is a simple high-level language for describing nondeterministic finite-state machines. Many features of the language are familiar from conventional programming languages. The main features not found in a "typical" high-level language are described in the following paragraphs.

The *state* of the model consists of the values of all global variables. In a *startstate* statement, initial values are assigned to global variables. The transition from one state to another is performed by *rules*. Each rule has a Boolean condition and an action, which is a program segment that is executed atomically. The action may be executed if the condition is true (i.e., the rule is enabled) and typically changes global variables, yielding a new state. Most Murφ models are nondeterministic since states typically allow execution of more than one rule. For example, in the model of the SSL protocol, the intruder usually has the nondeterministic choice of several messages to replay.

Murφ has no explicit notion of *processes*. Nevertheless a process can be implicitly modeled by a set of related rules. The *parallel composition* of two processes in Murφ is simply done by using the union of the rules of the two processes. Each process can take any number of steps (actions) between the steps of the other. The resulting computational model is that of *asynchronous, interleaving* concurrency. Parallel processes communicate via shared variables; there are no special language constructs for communication.

The Murφ language supports *scalable* models. In a scalable model, one is able to change the size of the model by simply changing constant declarations. When developing protocols, one typically starts with a small protocol configuration. Once this configuration is correct, one gradually increases the protocol size to the largest value that still allows verification to complete. In many cases, an error in the general (possibly infinite state) protocol will also show up in a down-scaled (finite state) version of the protocol. Murφ can only guarantee correctness of the down-scaled version of the protocol, but not correctness of the general protocol. For example, in the model of the SSL protocol, the numbers of clients and servers are scalable and defined by constants.

The desired properties of a protocol can be specified in Murφ by invariants, which are Boolean conditions that have to be true in every reachable state. If a state is reached in which some invariant is violated, Murφ prints an error trace – a sequence of states from the start state to the state exhibiting the problem.

There are two main differences between Murφ and FDR. First, while communication is supported in FDR by the CSP notions of channels and events, it is modeled by shared variables in Murφ. Second, Murφ currently implements a richer set of methods for increasing the size of the protocols that can be verified – consisting of both several techniques to reduce the number of reachable states [6] and several techniques to perform the state space search more efficiently, reducing runtime and memory requirements [10].

2.2 The methodology

In outline, we have analyzed protocols using the following sequence of steps:

1. *Formulate the protocol.* This generally involves simplifying the protocol by identifying the key steps and primitives. The Murφ formulation of a protocol, however, is more detailed than the high-level descriptions often seen in the literature, since one has to decide exactly which messages will be accepted by each participant in the protocol. Since Murφ communication is based on shared variables, it is also necessary to define an explicit message format, as a Murφ type.

2. *Add an adversary to the system.* We generally assume that the adversary (or intruder) is a participant in the system, capable of initiating communication with an honest participant, for example. We also assume that the network is under control of the adversary and allow the adversary the following actions:

   - overhear every message, remember all parts of each message, and decrypt cyphertext when it has the key,
   - intercept (delete) messages,
   - generate messages using any combination of initial knowledge about the system and parts of overheard messages.

   Although it is simplest to formulate an adversary that nondeterministically chooses between all possible actions at every step of the protocol, it is more efficient to reduce the choices to those that actually have a chance of affecting other participants.

3. *State the desired correctness condition.* A typical correctness criterion includes, e.g., that no secret information can be learned by the intruder. More details are given in Section 3.

4. *Run the protocol* for some specific choice of system size parameters. Speaking very loosely, systems with 4 or 5 participants (including the adversary) and 3 to 5 intended steps in the original protocol (without adversary) are easily analyzed in minutes of computation time using a modest workstation. Doubling or tripling these numbers, however, may cause the system to run for many hours, or terminate inconclusively by exceeding available memory.

5. *Experiment with alternate formulations and repeat.* This is discussed in detail in Section 4.

3 The SSL 3.0 handshake protocol

The primary goal of the SSL 3.0 handshake protocol is to establish secret keys that "provide privacy and reliability between two communicating applications" [5]. Henceforth, we call the communicating applications the client (C) and the server (S). The basic approach taken by SSL is to have C generate a fresh random number (the *secret* or *shared secret*) and deliver it to S in a secure manner. The secret is then used to compute a so-called *master secret* (or *negotiated cipher*), from which, in turn, the keys that protect and authenticate subsequent communication between C and S are computed. While the SSL *handshake protocol* governs the secret key calculation, the SSL *record layer protocol* governs the subsequent secure communication between C and S.

As part of the handshake protocol, C and S exchange their respective *cryptographic preferences*, which are used to select a mutually acceptable set of algorithms for encrypting and signing handshake messages. In our analysis, we assume for simplicity that RSA is used for both encryption and signatures, and cryptographic preferences only indicate the desired lengths of keys. In addition, SSL 3.0 is designed to be backward-compatible so that a 3.0 server can communicate with a 2.0 client and vice versa. Therefore, the parties also exchange their respective version numbers.

The basic handshake protocol consists of three messages. With the *client hello* message, the client starts the protocol and transmits its version number and cryptographic preferences to the server. The server replies with the *server hello* message, also transmitting its version number and cryptographic preferences. Upon receipt of this message, the client generates the shared secret and sends it securely to the server in the *secret exchange message*.

Since we were not aware of any formal definition of SSL 3.0, we based our model of the handshake protocol on the Internet draft [5]. The draft does not include a precise list of requirements that must be satisfied by the communication channel created after the handshake protocol completes. Based on our interpretation of the informal discussion in Sections 1 and 5.5 of the Internet draft, we believe that the resulting channel can be considered "secure" if and only if the handshake protocol completes successfully (i.e., both parties reach the state in which they are ready to communicate using the negotiated cipher) and the following properties must hold after protocol completion:

- Let $Secret_C$ be the number that C considers the shared secret, and $Secret_S$ the number that S considers the shared secret. Then $Secret_C$ and $Secret_S$ must be identical.

- The secret shared between C and S is not in intruder I's database of known message components.

- The parties agree on each other's identity and protocol completion status. More precisely, C reaches the state in which it is ready to start communicating with S using the negotiated cipher iff S reaches the corresponding state in which it is ready to start communicating with C.

- The cryptographic algorithms selected by the parties for encryption and authentication of handshake messages are the strongest ones that are supported by both C and S.
- The parties have a consistent opinion about each other's version, i.e., it is never the case that an SSL 3.0 client and a 3.0 server are communicating using the SSL 2.0 protocol.

We propose that any violation of the foregoing invariants that goes undetected by the legitimate participants constitutes a successful attack on the protocol.

SSL 3.0 supports *protocol resumption*. In the initial run of the protocol, C and S establish a shared secret by going through the full protocol and computing secret keys that protect subsequent communication. SSL 3.0 allows the parties to resume their connection at a later time without repeating the full protocol. If the client hello message sent by C to S includes the identifier of an SSL session that is still active according to S's internal state, the server assumes that C wants to resume a previous session. No public keys or secrets are exchanged in this case, but the master secret and the keys derived from it are recomputed using new nonces. (See Section 4.8 for an explanation of how nonces are used in the protocol to prevent replay attacks, and the Appendix to see how the master secret is computed from the nonces and shared secret.) Our Murφ model supports protocol resumption.

Finally, it should be noted that whenever one of the parties detects an inconsistency in the messages it receives, or any of the protocol steps fails in transmission, the protocol is aborted and the parties revert to their initial state. This implies that SSL is susceptible by design to some forms of "denial of service" attacks: an intruder can simply send an arbitrary message to a client or server engaged in the handshake protocol, forcing protocol failure.

4 Modeling SSL 3.0

We start our incremental analysis with the simplest and most intuitive version of the protocol and give an attack found by Murφ. We then add a little piece of SSL 3.0 that foils the attack, and let Murφ discover an attack on the augmented protocol. We continue this iterative process until no more attacks can be found. The final protocol turns out to be a stripped-down version of SSL 3.0.

4.1 Notation

The following notation will be used throughout the paper.

Ver_i	SSL version number of party i
$Suite_i$	Cryptographic preferences of party i
N_i	Random nonce generated by party i
$Secret_i$	Random secret generated by party i
K_i^+	Public encryption key of party i
V_i	Public verification key of party i
$\text{sign}_i\{\ldots\}$	Signed by party i
$\{\ldots\}_{K_i^+}$	Encrypted by public key K_i^+
$Messages$	All messages up to this point
$\langle I \rangle$	Message is intercepted by the intruder

4.2 Assumptions about cryptography

In general, our model assumes perfect cryptography. The following list explains what this assumption implies for all cryptographic functions used in SSL.

Opaque encryption. Encryption is assumed to be opaque. If a message has the form $\{x\}_{K_i^+}$, only party i can learn x. (This is only true iff the private key K_i^- is not available to any party except i. This is a safe assumption, given that no participants in the SSL handshake protocol are ever required to send their private key over the network.) The intruder may, however, store the entire encrypted message and replay it later without learning x. The structure of the encrypted message is inaccessible to the intruder, i.e., it cannot split the encrypted message into parts and insert them into other encrypted messages.

Unforgeable signatures. Signatures are assumed to be unforgeable. Messages of the form $\text{sign}_i\{x\}$ can only be generated by the party i. Anyone who possesses i's verification key V_i is able to verify that the message was indeed signed by i. We assume that signatures do not encrypt. Therefore, x can be learned by anyone.

Collision-free hashes. Hashes are assumed to be collision-free. Given a message of the form $\text{Hash}\{x\}$, it is not computationally feasible to find any x' such that $\text{Hash}\{x'\} = \text{Hash}\{x\}$. In protocols, it is therefore assumed that a participant can determine whether $x = x'$ by comparing $\text{Hash}\{x\}$ to $\text{Hash}\{x'\}$.

Trusted certificate authority. There exists a trusted certificate authority (CA). All parties are assumed to possess CA's verification key V_{CA}, and are thus able to verify messages signed by CA. Every party i is assumed to possess CA-signed certificates for its own public keys: $\text{sign}_{CA}\{i, K_i^+\}$ (certifying that public encryption key K_i^+ indeed belongs to i) and $\text{sign}_{CA}\{i, V_i\}$ (certifying that public verification key V_i indeed belongs to i).

4.3 Protocol A

Basic protocol (A)

The first step of the basic protocol consists of C sending information about its identity, SSL version number, and cryptographic preferences (aka *cryptosuite*) to S. Upon receipt of C's hello message, S sends back its version, cryptosuite (S selects one set of algorithms from the preference list submitted by C), and its public encryption key. C then generates a random secret and sends it to S, encrypted by S's public key.

Notice that the first hello message (that from C to S) contains the identity of C. There is no way for S to know who initiated the protocol unless this information is contained in the message itself (perhaps implicitly in the network packet header).

$C \to S$	$C,\ Ver_C,\ Suite_C$
$S \to C$	$Ver_S,\ Suite_S,\ K_S^+$
$C \to S$	$\{Secret_C\}_{K_S^+}$

⟨Change to negotiated cipher⟩

Attack on A

Protocol A does not explicitly (and securely) associate the server's name with its public encryption key. This allows the intruder to insert its own key into the server's hello message. The client then encrypts the generated secret with the intruder's key, enabling the intruder to read the message and learn the secret.

$C \to S$	$C,\ Ver_C,\ Suite_C$
$S \to C\langle I\rangle$	$Ver_S,\ Suite_S,\ K_S^+$
$I \to C$	$Ver_S,\ Suite_S,\ \underline{K_I^+}$
$C \to S\langle I\rangle$	$\{Secret_C\}_{K_I^+}$
$I \to S$	$\{Secret_C\}_{K_S^+}$

⟨Change to negotiated cipher⟩

4.4 Protocol B

A + server authentication

To fix the bug in Protocol A, we add verification of the public key. The server now sends its public key K_S^+ in a certificate signed by the certificate authority. As described before, the certificate has the following form: $\text{sign}_{CA}\{S, K_S^+\}$.

We assume that signatures are unforgeable. Therefore, the intruder will not be able to generate $\text{sign}_{CA}\{S, K_I^+\}$. The intruder may send the certificate for its own public key $\text{sign}_{CA}\{I, K_I^+\}$, but the client will reject it since it expects S's name in the certificate. Finally, the intruder may generate $\text{sign}_I\{S, K_I^+\}$, but the client expects a message signed by CA, and will try to verify it using CA's verification key. Verification will fail since the message is not signed by CA, and the client will abort the protocol. Notice that SSL's usage of certificates to verify the server's public key depends on the trusted certificate authority assumption (see Section 4.2 above).

$C \to S$	$C,\ Ver_C,\ Suite_C$
$S \to C$	$Ver_S,\ Suite_S,\ \mathrm{sign}_{CA}\{S, K_S^+\}$
$C \to S$	$\{Secret_C\}_{K_S^+}$

⟨Change to negotiated cipher⟩

Attack on B

Protocol B includes no verification of the client's identity. This allows the intruder to impersonate the client by generating protocol messages and pretending they originate from C. In particular, the intruder is able to send its own secret to the server, which the latter will use to compute the master secret and the derived keys.

$\underline{I} \to S$	$C,\ Ver_C,\ Suite_C$
$S \to C\langle I\rangle$	$Ver_S,\ Suite_S,\ \mathrm{sign}_{CA}\{S, K_S^+\}$
$I \to S$	$\{\underline{Secret_I}\}_{K_S^+}$

⟨Change to negotiated cipher⟩

4.5 Protocol C

B + client authentication

To fix the bug in Protocol B, the server has to verify that the secret it received was indeed generated by the party whose identity was specified in the first hello message. For this purpose, SSL employs client signatures.

The client sends to the server its verification key in the CA-signed certificate $\mathrm{sign}_{CA}\{C, V_C\}$. In addition, immediately after sending its secret encrypted with the server's public key, the client signs the hash of the secret $\mathrm{sign}_C\{\mathrm{Hash}\,(Secret_C)\}$ and sends it to the server. Hashing the secret is necessary so that the intruder will not be able to learn the secret even if it intercepts the message. Since the server can learn the secret by decrypting the client key exchange message, it is able to compute the hash of the secret and compare it with the one sent by the client.

Notice that the server can be assured that V_C is indeed C's verification key since the intruder cannot insert its own key in the CA-signed certificate $\mathrm{sign}_{CA}\{C, V_C\}$ assuming that signatures are unforgeable. Therefore, the server will always use V_C to verify messages ostensibly signed by the client, and all messages of the form $\mathrm{sign}_I\{\ldots\}$ will be rejected. Even if the intruder were able to generate the message $\mathrm{sign}_C\{\mathrm{Hash}\,(Secret_I)\}$, the attack will be detected when the server computes $\mathrm{Hash}\,(Secret_C)$ and discovers that it is different from $\mathrm{Hash}\,(Secret_I)$.

Instead of signing the hashed secret, the client can sign the secret directly and send it to the server encrypted by the server's public key. The SSL definition, however, does not include encryption in this step [5, Section 5.6.8]. We used hashing instead of encryption as well since we intend our incremental reconstruction of SSL to follow the definition as closely as possible. The exact reason why the designers of the protocol chose to rely on hashing rather than encryption to protect the secret is unclear to us.

It is interesting to note that SSL 2.0 has several problems in the client authentication stage. It uses the same key for verification and encryption, while the approach outlined above allows the verification key to be longer (and thus, hopefully, more secure) than the encryption key. Also, the strength of the verification key in SSL 2.0 is independent of the cryptographic preferences.

$C \to S$	$C,\ Ver_C,\ Suite_C$
$S \to C$	$Ver_S,\ Suite_S,\ \mathrm{sign}_{CA}\{S, K_S^+\}$
$C \to S$	$\mathrm{sign}_{CA}\{C, V_C\},\ \{Secret_C\}_{K_S^+},$ $\mathrm{sign}_C\{\mathrm{Hash}\,(Secret_C)\}$

⟨Change to negotiated cipher⟩

Attack on C

Even though the intruder cannot modify keys and shared secrets in Protocol C, it is able to attack the plaintext information transmitted in the hello messages. This includes the parties' version numbers and cryptographic preferences.

By modifying version numbers, the intruder can convince an SSL 3.0 client that it is communicating with a 2.0 server, and a 3.0 server that it is communicating with a 2.0 client. This will cause the parties to communicate using SSL 2.0, giving the intruder an opportunity to exploit any of the known weaknesses in SSL 2.0.

By modifying the parties' cryptographic preferences, the intruder can force them into selecting a weaker encryption and/or signing algorithm than they normally would. This may make it easier for the intruder to decrypt the client's secret exchange message, or to forge the client's signature. It is interesting to note that the SSL 2.0 protocol is susceptible to this "dumbing down" attack.

$C \to S\langle I\rangle$	$C,\ Ver_C,\ Suite_C$
$I \to S$	$C,\ \underline{Ver_I},\ \underline{Suite_I}$
$S \to C\langle I\rangle$	$Ver_S,\ Suite_S,\ \mathrm{sign}_{CA}\{S, K_S^+\}$
$I \to C$	$\underline{Ver_I},\ \underline{Suite_I},\ \mathrm{sign}_{CA}\{S, K_S^+\}$
$C \to S$	$\mathrm{sign}_{CA}\{C, V_C\},\ \{Secret_C\}_{K_S^+},$ $\mathrm{sign}_C\{\mathrm{Hash}\,(Secret_C)\}$

⟨Change to negotiated cipher⟩

4.6 Protocol D

C + post-handshake verification of plaintext

The parties can prevent attacks on plaintext by repeating the exchange of versions and cryptographic preferences once the handshake protocol is complete; the additional messages will be called *verification messages*. Since the intruder cannot learn the shared secret, it cannot compute the master secret and the derived keys and thus cannot interfere with the parties' communication after they switch to the negotiated cipher.

Suppose the intruder altered the version number or cryptographic preferences in the client's hello message. When the client sends its version and cryptosuite to the server under the just negotiated encryption, the intruder cannot change them. The server will detect the discrepancy and abort the protocol. This is also true for the server's version and cryptosuite.

$C \to S$	$C,\ Ver_C,\ Suite_C$
$S \to C$	$Ver_S,\ Suite_S,\ \mathrm{sign}_{CA}\{S, K_S^+\}$
$C \to S$	$\mathrm{sign}_{CA}\{C, V_C\},\ \{Secret_C\}_{K_S^+},$ $\mathrm{sign}_C\{\mathrm{Hash}\,(Secret_C)\}$

⟨Change to negotiated cipher⟩

$S \to C$	$\{\mathrm{Hash}\,(Ver_C, Suite_C, Ver_S, Suite_S)\}_{\mathrm{Master}(Secret_C)}$
$C \to S$	$\{\mathrm{Hash}\,(Ver_C, Suite_C, Ver_S, Suite_S)\}_{\mathrm{Master}(Secret_C)}$

Attack on D

In Protocol D, the parties verify only plaintext information after the handshake negotiation is complete. Since the intruder cannot forge signatures, invert hash functions, or break encryption without the correct private key, it can neither learn the client's secret, nor substitute its own. It may appear that D provides complete security for the communication channel between C and S. However, Murφ discovered an attack on client's identity that succeeds even if all cryptographic algorithms are perfect.

Intruder I intercepts C's hello message to server S, and initiates the handshake protocol with S *under its own name.* All messages sent by S are re-transmitted to C, while most of C's messages, including the post-handshake verification messages, are re-transmitted to S. (See the protocol run below for details. Re-transmission of C's verification message is required to change the sender identifier, which is not shown explicitly below.) As a result, both C and S will complete the handshake protocol successfully, but C will be convinced that it is talking to S, while S will be convinced that it is talking to I.

Notice that I does not have access to the secret shared between C and S. Therefore, it will not be able to generate or decrypt encrypted messages after the protocol is complete, and will only be able to re-transmit C's messages. However, the server will believe that the messages are coming from I, whereas in fact they were sent by C.

This kind of attack, while somewhat unusual in that it explicitly reveals the intruder's identity, may prove harmful for a number of reasons. For example, it deprives C of the possibility to claim later that it communicated with S, since S will not be able to support C's claims (S may not even know about C's existence). If S is a pay server providing some kind of online service in exchange for anonymous "electronic coins" such as eCash [4], I may be able to receive service from S using C's cash. Recall, however, that I can only receive the service if it is not encrypted, which might be the case for large volumes of data.

$C \to S\langle I\rangle$	$C,\ Ver_C,\ Suite_C$
$I \to S$	$\underline{I},\ Ver_C,\ Suite_C$
$S \to I$	$Ver_S,\ Suite_S,\ \mathrm{sign}_{CA}\{S, K_S^+\}$
$I \to C$	$Ver_S,\ Suite_S,\ \mathrm{sign}_{CA}\{S, K_S^+\}$
$C \to S\langle I\rangle$	$\mathrm{sign}_{CA}\{C, V_C\},\ \{Secret_C\}_{K_S^+},$ $\mathrm{sign}_C\{\mathrm{Hash}\,(Secret_C)\}$
$I \to S$	$\underline{\mathrm{sign}_{CA}\{I, V_I\}},\ \{Secret_C\}_{K_S^+},$ $\underline{\mathrm{sign}_I}\{\mathrm{Hash}\,(Secret_C)\}$
$\langle$Change to negotiated cipher$\rangle$	
$S \to I$	$\{\mathrm{Hash}\,(Ver_C, Suite_C, Ver_S, Suite_S)\}_{\mathrm{Master}(Secret_C)}$
$I \to C$	$\{\mathrm{Hash}\,(Ver_C, Suite_C, Ver_S, Suite_S)\}_{\mathrm{Master}(Secret_C)}$
$C \to S\langle I\rangle$	$\{\mathrm{Hash}\,(Ver_C, Suite_C, Ver_S, Suite_S)\}_{\mathrm{Master}(Secret_C)}$
$I \to S$	$\{\mathrm{Hash}\,(Ver_C, Suite_C, Ver_S, Suite_S)\}_{\mathrm{Master}(Secret_C)}$

4.7 Protocol E

D + post-handshake verification of all messages

To fix the bug in Protocol D, the parties verify *all* of their communication after the handshake is complete. Now the intruder may not re-transmit C's messages to S, because C's hello message contained C, while the hello message received by the server contained I. The discrepancy will be detected in post-handshake verification.

$C \rightarrow S$	$C,\ \mathit{Ver}_C,\ \mathit{Suite}_C$
$S \rightarrow C$	$\mathit{Ver}_S,\ \mathit{Suite}_S,\ \mathrm{sign}_{CA}\{S, K_S^+\}$
$C \rightarrow S$	$\mathrm{sign}_{CA}\{C, V_C\},\ \{\mathit{Secret}_C\}_{K_S^+},$ $\mathrm{sign}_C\{\mathrm{Hash}\,(\mathit{Secret}_C)\}$
⟨Change to negotiated cipher⟩	
$S \rightarrow C$	$\{\mathrm{Hash}\,(\mathit{Messages})\}_{\mathrm{Master}(\mathit{Secret}_C)}$
$C \rightarrow S$	$\{\mathrm{Hash}\,(\mathit{Messages})\}_{\mathrm{Master}(\mathit{Secret}_C)}$

Attack on E

I observes a run of the protocol and records all of C's messages. Some time later, I initiates a new run of the protocol, ostensibly from C to S, and replays recorded C's messages in response to messages from S. Even though I is unable to read the recorded messages, it manages to convince S that the latter is talking to C, even though C did not initiate the protocol.

$C \rightarrow S$	$C,\ \mathit{Ver}_C,\ \mathit{Suite}_C$
$S \rightarrow C$	$\mathit{Ver}_S,\ \mathit{Suite}_S,\ \mathrm{sign}_{CA}\{S, K_S^+\}$
$C \rightarrow S$	$\mathrm{sign}_{CA}\{C, V_C\},\ \{\mathit{Secret}_C\}_{K_S^+},$ $\mathrm{sign}_C\{\mathrm{Hash}\,(\mathit{Secret}_C)\}$
⟨Change to negotiated cipher⟩	
$S \rightarrow C$	$\{\mathrm{Hash}\,(\mathit{Messages})\}_{\mathrm{Master}(\mathit{Secret}_C)}$
$C \rightarrow S$	$\{\mathrm{Hash}\,(\mathit{Messages})\}_{\mathrm{Master}(\mathit{Secret}_C)}$

Next run of the protocol ...

$I \to S$	$C,\ \mathit{Ver}_C,\ \mathit{Suite}_C$
$S \to C\langle I\rangle$	$\mathit{Ver}_S,\ \mathit{Suite}_S,\ \mathrm{sign}_{CA}\{S, K_S^+\}$
$I \to S$	$\mathrm{sign}_{CA}\{C, V_C\},\ \{\mathit{Secret}_C\}_{K_S^+},$ $\mathrm{sign}_C\{\mathrm{Hash}\,(\mathit{Secret}_C)\}$
$\langle$Change to negotiated cipher$\rangle$	
$S \to C\langle I\rangle$	$\{\mathrm{Hash}\,(\mathit{Messages})\}_{\mathrm{Master}(\mathit{Secret}_C)}$
$I \to S$	$\{\mathrm{Hash}\,(\mathit{Messages})\}_{\mathrm{Master}(\mathit{Secret}_C)}$

4.8 Protocol F

E + nonces

By adding random nonces to each run of the protocol, we make sure that there are always some differences between independent runs of the protocol. The intruder is thus unable to replay verification messages from one run in another run.

$C \to S$	$C,\ \mathit{Ver}_C,\ \mathit{Suite}_C,\ N_C$
$S \to C$	$\mathit{Ver}_S,\ \mathit{Suite}_S,\ N_S,\ \mathrm{sign}_{CA}\{S, K_S^+\}$
$C \to S$	$\mathrm{sign}_{CA}\{C, V_C\},\ \{\mathit{Secret}_C\}_{K_S^+},$ $\mathrm{sign}_C\{\mathrm{Hash}\,(\mathit{Secret}_C)\}$
$\langle$Change to negotiated cipher$\rangle$	
$S \to C$	$\{\mathrm{Hash}\,(\mathit{Messages})\}_{\mathrm{Master}(\mathit{Secret}_C)}$
$C \to S$	$\{\mathrm{Hash}\,(\mathit{Messages})\}_{\mathrm{Master}(\mathit{Secret}_C)}$

Attack on F

The exact semantics of the verification messages exchanged after switching to the negotiated cipher (i.e., *Finished* messages in the SSL terminology) is somewhat unclear. Section 5.6.9 of [5] states: "No acknowledgment of the finished message is required; parties may begin sending encrypted data immediately after sending the finished message. Recipients of finished messages must verify that the contents are correct." The straightforward implementation of this definition led Murφ to discover the following attack on Protocol F:

1. I modifies the hello messages, changing the legitimate parties' cryptosuites so as to force them into choosing a weak encryption algorithm for the secret exchange.

2. I records the weakly encrypted $Secret_C$ as it is being transmitted from C to S.

3. After C and S switch to the negotiated cipher, I delays their verification messages indefinitely, preventing them from discovering the attack on cryptosuites and gaining extra time to crack the encryption algorithm and learn $Secret_C$.

4. Once the secret is learned, I is able to compute the keys and forge verification messages.

Since we did not model weak encryption that can be broken by the intruder, we also did not model the last step of the attack explicitly. Instead, if the model reached the state after the 3rd step, the attack was considered successful.

4.9 Protocol G

To prevent the attack on Protocol F, it is sufficient to require that the parties consider the protocol incomplete until checking verification messages they received from each other. Murφ did not discover any bugs in the model implemented according to this semantics.

Alternatively, yet another piece of SSL can be added to Protocol F. If the client sends the server a hash of all messages *before* switching to the negotiated cipher, the server will be able to detect an attack on its cryptosuite earlier.

$C \rightarrow S$	$C,\ Ver_C,\ Suite_C,\ N_C$
$S \rightarrow C$	$Ver_S,\ Suite_S,\ N_S,\ \mathrm{sign}_{CA}\{S, K_S^+\}$
$C \rightarrow S$	$\mathrm{sign}_{CA}\{C, V_C\},\ \{Secret_C\}_{K_S^+},$ $\mathrm{sign}_C\{\mathrm{Hash}\,(Messages)\}$
⟨Change to negotiated cipher⟩	
$S \rightarrow C$	$\{\mathrm{Hash}\,(Messages)\}_{\mathrm{Master}(Secret_C)}$
$C \rightarrow S$	$\{\mathrm{Hash}\,(Messages)\}_{\mathrm{Master}(Secret_C)}$

Murφ was used to model Protocol G with 2 clients, 1 intruder, 1 server, no more than 2 simultaneous open sessions per server, and no more than 1 resumption per session. No bugs were discovered. We are currently investigating several protocol resumption anomalies related to the version rollback attack.

4.10 Protocol G vs. SSL 3.0

Following is the definition of the SSL 3.0 handshake protocol according to [5]. When several messages from the same party follow each other in the original definition, they have been collapsed into a single protocol step (e.g., *Certificate*, *ClientKeyExchange*, and *CertificateVerify* were joined into *ClientVerify*). The underlined pieces of SSL 3.0 are not in Protocol G.

`ClientHello` $\quad C \to S \quad C,\ Ver_C,\ Suite_C, N_C$

`ServerHello` $\quad S \to C \quad Ver_S,\ Suite_S,\ N_S,\ \mathrm{sign}_{CA}\{S, K_S^+\}$

`ClientVerify` $\quad C \to S$

$$\begin{aligned}
&\mathrm{sign}_{CA}\{C, V_C\},\\
&\{\underline{Ver_C}, Secret_C\}_{K_S^+},\\
&\mathrm{sign}_C\{\,\mathrm{Hash}\,(\mathrm{Master}(N_C,\ N_S,\ Secret_C) + Pad_2 +\\
&\qquad \mathrm{Hash}\,(Messages + C +\\
&\qquad\qquad \mathrm{Master}(N_C,\ N_S,\ Secret_C) +\\
&\qquad\qquad Pad_1))\,\}
\end{aligned}$$

⟨Change to negotiated cipher⟩

`ServerFinished` $\quad S \to C$

$$\begin{aligned}
&\{\,\mathrm{Hash}\,(\mathrm{Master}(N_C,\ N_S,\ Secret_C) + Pad_2 +\\
&\qquad \mathrm{Hash}\,(Messages\ + S +\\
&\qquad\qquad \mathrm{Master}(N_C,\ N_S,\ Secret_C) +\\
&\qquad\qquad Pad_1))\,\}_{\mathrm{Master}(N_C,\ N_S,\ Secret_C)}
\end{aligned}$$

`ClientFinished` $\quad C \to S$

$$\begin{aligned}
&\{\,\mathrm{Hash}\,(\mathrm{Master}(N_C,\ N_S,\ Secret_C) + Pad_2 +\\
&\qquad \mathrm{Hash}\,(Messages\ + C +\\
&\qquad\qquad \mathrm{Master}(N_C,\ N_S,\ Secret_C) +\\
&\qquad\qquad Pad_1))\,\}_{\mathrm{Master}(N_C,\ N_S,\ Secret_C)}
\end{aligned}$$

5 Using Murφ for protocol verification

While the main goal of our analysis was "rational reconstruction" of the SSL 3.0 handshake protocol, we were also interested in lessons to be learned about using finite-state analysis to verify large protocols. We were particularly concerned about the potentially very large number of reachable states, given that the SSL handshake protocol consists of 7 steps, and each message sent in a particular step includes several components, each of which can be changed by the intruder under certain conditions.

Our model of the intruder is very simple and straightforward. There is no intrinsic knowledge of the protocol embedded in the intruder, nor does the design of the intruder involve any prior knowledge of any form of attack. The intruder may monitor communication between the protocol participants, intercept and generate messages, split intercepted messages into components and recombine them in arbitrary order. No clues are given, however, as to which of these techniques should be used at any given moment. Therefore, the effort involved in implementing the model of the intruder is mostly mechanical.

The following simple techniques proved useful in reducing the number of states to be explored:

Full knowledge. We assume that every message sent on the network is intercepted by the intruder. Clearly, this assumption does not weaken the intruder. Without it, however, a high percentage of the states that Murφ explored were identical as far as the state of the legitimate participants was concerned and the only difference was the contents of the intruder's database. By ensuring that the database is always as full as it can possibly be (i.e., it includes all knowledge that can be extracted from the messages transmitted on the network thus far), we achieved an order of magnitude reduction in the number of reachable states (e.g., from approximately 200,000 to 5,000 for a single run of the protocol).

No useless messages. The intruder only generates messages that are expected by the legitimate parties and can be meaningfully interpreted by them in their current state. Since at any point in the protocol sequence, each party is expecting only one particular type of message, the number of message types the intruder generates at each step does not exceed the number of protocol participants. Still, the number of ways in which the intruder could construct individual messages might be large since messages are generated from the message components (keys, nonces, etc.) stored in the intruder's database.

No useless components. If the intercepted message can be completely recreated from the components already in the intruder's database, the message is discarded.

Flags. Every procedure executed by the protocol participants after receiving a message is guarded by a flag. By changing flag values, we can turn on and off pieces of the protocol, enabling incremental and "what-if" modeling (e.g., what happens if the server does not verify the hashed secret it receives from the client).

Running under Linux on a Pentium-120 with 32MB of RAM, the model takes approximately 1.5 minutes to check for the case of 1 client, 1 server, 1 open session, and no resumptions. Less than 5000 states are explored.

The largest instance of our model that we verified included 2 clients, 1 server, no more than 2 simultaneous open sessions per server, and no more than 1 resumption per session. Checking took slightly under 8 hours, with 193,000 states explored.

Appendix: master secret computation

The SSL 3.0 master secret is computed in the following way.

$$\begin{aligned}
&\mathrm{Master}(N_C, N_S, \mathit{Secret}_C) = \\
&\quad \mathrm{MD5}\,(\mathit{Secret}_C + \mathrm{SHA}\,(\text{'}A\text{'} + \mathit{Secret}_C + N_C + N_S)) \; + \\
&\quad \mathrm{MD5}\,(\mathit{Secret}_C + \mathrm{SHA}\,(\text{'}BB\text{'} + \mathit{Secret}_C + N_C + N_S)) \; + \\
&\quad \mathrm{MD5}\,(\mathit{Secret}_C + \mathrm{SHA}\,(\text{'}CCC\text{'} + \mathit{Secret}_C + N_C + N_S))
\end{aligned}$$

References

[1] S. Dietrich. *A Formal Analysis of the Secure Sockets Layer Protocol.* PhD thesis, Dept. Mathematics and Computer Science, Adelphi University, April 1997.

[2] D. L. Dill. The Murφ verification system. In *Computer Aided Verification. 8th International Conference*, pages 390–3, 1996.

[3] D. L. Dill, S. Park, and A. G. Nowatzyk. Formal specification of abstract memory models. In *Symposium on Research on Integrated Systems*, pages 38–52, 1993.

[4] `http://www.digicash.com/ecash/ecash-home.html`.

[5] A. O. Freier, P. Karlton, and P. C. Kocher. The SSL protocol version 3.0. `draft-ietf-tls-ssl-version3-00.txt`, November 18, 1996.

[6] C. N. Ip. *State Reduction Methods for Automatic Formal Verification*. PhD thesis, Stanford University, 1996.

[7] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using CSP and FDR. In *2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*. Springer-Verlag, 1996.

[8] J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Murφ. In *IEEE Symposium on Security and Privacy*, pages 141–51, 1997.

[9] S. Schneider. Security properties and CSP. In *IEEE Symp. Security and Privacy*, 1996.

[10] U. Stern. *Algorithmic Techniques in Verification by Explicit State Enumeration*. PhD thesis, Technical University of Munich, 1997.

[11] U. Stern and D. L. Dill. Automatic verification of the SCI cache coherence protocol. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 21–34, 1995.

[12] L. Yang, D. Gao, J. Mostoufi, R. Joshi, and P. Loewenstein. System design methodology of UltraSPARC™-I. In *32nd Design Automation Conference*, pages 7–12, 1995.