Three Tools for Model-Checking Security Protocols

Nelson Arruda Ali Juma

January 5, 2004

Abstract

We look at three tools for model-checking security protocols. These tools are OFMC, Hermes, and Maude. We also model and verify the SSL Handshake Protocol using OFMC.

1 Introduction

While the Internet becomes a more prominent tool in our society, there is a corresponding increase in the need for tools to exchange sensitive data. Security protocols have garnered much attention in their role in fulfilling this need. It is important to verify that these protocols do in fact provide the required security. Analyzing these protocols by hand was time-consuming, error-prone and, at times, incomplete. As a result, model-checking tools have been developed to analyze security protocols. However, it should be noted that protocol security is undecidable [DLMS99], which limits the power of any automated tool. In this paper, we give a brief introduction to security protocols and model-checking for security protocols. We present three recently-developed model-checkers that can be used to check protocol security - OFMC, Hermes, and Maude. Finally, we use OFMC to check the security of the SSL Handshake Protocol.

2 Security protocols

Security protocols are used extensively for both data storage and communication. Their purpose in data storage is to deny access to any party for which the data is not "intended". Security protocols are used for communication to allow parties to exchange information over networks with the assurance that the data the parties exchange cannot be read by a third party. Further, security protocols allow a party to verify the identity of the party with which it is communicating. In this paper, we concentrate on communication rather than data storage.

A security protocol defines a set of roles and a sequence of messages that are exchanged between the parties that play each role. A message in the sequence is expressed in the form $A \to B : M$, meaning that the party playing role A sends the message M to the party playing role B. A given party can play any number of roles in a single execution of the protocol. Typically, cryptographic algorithms are required to construct the messages that are exchanged.

Cryptographic algorithms are based on a conjecture that solving certain problems is "hard" (computationally infeasible) for those who don't have a certain secret piece of information (the key), but "easy" for those who do. As a result, outside parties cannot learn anything about information which has been encrypted using these algorithms. When model-checking security protocols, the theoretical concepts behind these algorithms are not considered.

We now define some terms that relate to security protocols:

Definition 1 (Public Key Cryptography) Each user has a pair of keys, K and K', public and private, respectively. A message is encrypted with the receiver's public key, and decrypted with the receiver's private key. However, a message can also be encrypted with the sender's private key, and decrypted with the sender's public key. The latter can be used for digital signatures. Public key cryptography is also known as asymmetric key cryptography.

Definition 2 (Symmetric Key Cryptography) *Symmetric key cryptography* is an encryption system in which the sender and receiver of a message share a single, common key that is used both for encryption and decryption. In contrast, public-key cryptography utilizes two keys.

Definition 3 (Hash Function) A *hash function* H is a function that takes an input m of arbitrary length and returns a string $H(m)$ of a fixed length, called the hash value. For cryptographic purposes, H has the following properties: 1) $H(x)$ is easy to compute for any x; 2) H is one-way; and 3) H is collision-free (see below for definitions of "one-way" and "collision-free").

Definition 4 (One-way Function) A *one-way function* f is a function whose inverse is "hard" to compute. This means that given y, it is "hard" to find x such that $f(x) = y$.

Definition 5 (Collision-Free Function) A function f is *weak collision-free* if given x, it is "hard" to find $y \neq x$ such that $f(x) = f(y)$. A function f is *strong collision-free* if it is "hard" to find x and y, with $x \neq y$, such that $f(x) = f(y)$.

Definition 6 (Authentication) *Authentication* refers to verifying the identity of the party with which one is communicating.

Definition 7 (Certificate) A party's *certificate* states the party's name and its public key. The certificate is signed by a certificate authority (an entity that is trusted by all parties). That is, the certificate is encrypted using the certificate authority's private key. All parties are assumed to possess the certificate authority's public key.

3 Model-checking for security protocols

When model-checking security protocols, we concentrate on looking for logical flaws in the protocol, rather than flaws in the cryptographic algorithms. Typically, the *Dolev-Yao model* [DY83] is used to define the power of a hostile party (also known as an "intruder") trying to break the system. According to the Dolev-Yao model, the limits on the intruder's power are:

- The intruder cannot guess keys or the content of an encrypted string. Therefore, in order to decrypt $\{m\}k$ (meaning message m encrypted with key k), the intruder must know the inverse of k.
- The intruder cannot distinguish a true random number from one generated by a random number generator. That is, we assume that perfect random number generators are used.

On the other hand, according to the Dolev-Yao model, the intruder has complete control of the network. This means that the intruder can listen to all network traffic, intercept any message (preventing it from reaching its destination), and send messages that appear to come from another party (that is, the intruder can *masquerade* as another party). The intruder can construct new messages by composing messages it possesses, encrypting messages it possesses by using keys that it possesses, and decrypting messages it possesses by using keys that it possesses. This means that as long as the intruder possesses at least one message, it is able to construct infinitely many messages.

4 Needham-Schroeder Public Key Protocol (NSPK)

The Needham-Schroeder Public Key Protocol [NS78] is a classical example that is used in model-checking of security protocols.

Two parties, A and B have public keys, ka and kb respectively. It is assumed that the two parties know each other's public key. The goal is mutual authentication. In the following, Na and Nb are random values chosen by A and B respectively, while $\{X\}k$ denotes X encrypted with key k.

The protocol is as follows:

1. $A \rightarrow B : \{Na, A\}kb$
2. $B \rightarrow A : \{Na, Nb\}ka$
3. $A \rightarrow B : \{Nb\}kb$

When the protocol was introduced in 1978, it was thought to have provided mutual authentication. However, in 1996, Lowe [Low96] discovered a "Man in the Middle" attack. Say the intruder I has a public key ki. The attack is as follows:

1. $A \rightarrow I : \{Na, A\}ki$

2. $I \rightarrow B : \{Na, A\}kb$

3. $B \rightarrow I : \{Na, Nb\}ka$

4. $I \rightarrow A : \{Na, Nb\}ka$

5. $A \rightarrow I : \{Nb\}ki$

6. $I \rightarrow B : \{Nb\}ka$

At this point the problem is that B thinks I is in fact A, which violates the goal of the protocol.

5 OFMC

OFMC (On-the-Fly Model-Checker) [Möd03, BMV03, ABB+03] is a model-checker for security protocols, developed by the Information Security Group at ETH Zurich. OFMC has been tested on several protocols, including 46 of the 51 protocols in the Clark/Jacob [CJ97] library (the other five protocols cannot be expressed in OFMC's specification language, HLPSL). Out of these 46 protocols, 32 are known to have an attack, and OFMC finds attacks for each one [ABB+03].

OFMC uses two languages for protocol specification: HLPSL (High Level Protocol Specification Language) and IF(intermediate format).

HLPSL allows security protocols to be specified in a concise manner. An HLPSL specification must state:

- the initial knowledge required of each agent participating in the protocol
- the initial knowledge of the intruder
- the sequence of messages required by the protocol
- an initial set of sessions (or instantiations) of the protocol
- the intruder's abilities
- the goals that we want the protocol to achieve

The initial set of sessions is used to specify which agents are willing to communicate with each other and in what roles (with respect to the protocol) they are willing to do so. The intruder's abilities can include impersonating other agents (sending messages that appear to be from another agent), intercepting messages, and eavesdropping (reading messages without preventing them from reaching their destination). The goal corresponds to the property that we wish to verify. Goals can include authentication of agents and the secrecy of information. To make these ideas more concrete, an HLPSL specification of the NSPK protocol is provided in Figure 1.

```
PROTOCOL NSPK;
Identifiers
   A, B: user;
   Na, Nb:  number;
   Ka, Kb:  public_key;
Knowledge
   A: B,Ka,Ka',Kb;
   B: A,Ka,Kb,Kb';
Messages
   1.  A -> B: {Na,A}Kb
   2.  B -> A: {Na,Nb}Ka
   3.  A -> B: {Nb}Kb
Session_instances
   [ A:a, B:b, Ka:ka, Kb:kb ]
   [ A:a, B:I, Ka:ka, Kb:ki ];
Intruder divert, impersonate;
Intruder_knowledge I, b, ka, kb, ki;
Goal A authenticate B on Nb;
Goal B authenticate A on Na;
```

Figure 1: HLPSL specification (from [Möd03]) of the NSPK protocol

HLPSL can be used to model a variety of security protocols, including those that involve key generation, key distribution, and key revocation. However, HLPSL also has some limitations. Exponentiation cannot be expressed in HLPSL. As a result, the Diffie-Hellman key exchange cannot be modeled. Furthermore, HLPSL cannot be used to specify protocols whose messages include timestamps. There are also limitations on the goals that can be specified in HLPSL. For example, it is not possible to specify that agents should only sign messages whose contents they know.

OFMC includes a translator (HLPSL2IF) that converts an HLPSL specification to an IF specification. The OFMC model checker works with IF specifications.

IF is used to specify infinite state transition systems. The specification includes a goal predicate that determines if a state is an attack (on the protocol). In other words, checking for an attack is like checking a safety property. IF uses an optimization technique called *step-compression* to make sets of transitions atomic. For example, we can assume that an honest agent immediately replies (according to the protocol) to a message it receives. We don't need to consider the case where the honest agent waits for other messages from other sessions before replying, since this won't change the agent's reply - since the honest agent follows the protocol, the additional information will not change its reply. We can assume, without loss of generality, that the intruder intercepts every message sent by an honest agent, and that every message that is received by an honest agent is sent by the intruder. With this in mind, step-compression combines the following steps into a single atomic step: the intruder sends a message to an honest agent, the honest agent responds to the message, the intruder intercepts the honest agent's response. It can be shown that an attack state (i.e. a state where the safety property fails) exists when using step-compression if and only if an attack state exists without step compression [BMV03].

Since OFMC uses the Dolev-Yao model of the intruder, each node in the transition system has infinitely many children. Since there can be an unlimited number of protocol sessions being run simultaneously, the computation tree that corresponds to the transition system has infinite depth.

To deal with the infinite depth of the computation tree, OFMC uses the *lazy infinite-state* approach - an infinite-depth tree can be represented finitely using a lazy data-type. When representing a tree using lazy data-type, each node explains how to create its children, and the information needed to create a child is not evaluated until we actually want to visit this child. In other words, nodes are constructed as they are needed. This allows infinite-depth trees to be represented.

To deal with infinite branching in the computation tree, OFMC uses the *lazy intruder* approach to model the actions of the intruder. There are always infinitely many messages that the Dolev-Yao intruder can produce. As a result, OFMC can't deal with each such possible message individually. Instead, OFMC symbolically represents the (infinite) set of messages that the intruder is able to produce at any given state. That is, rather than exploring just a single branch of the computation tree at any given time, OFMC explores infinitely many branches simultaneously. Additional constraints are imposed on the set of possible messages as needed. For example, an honest agent may expect (based on the protocol) that part of a particular message is identical to part of a previous message, and an honest agent will not accept the message if this condition is not satisfied. In this case, OFMC adds a constraint to ensure that the intruder's message is one that will be accepted by an honest agent.

OFMC uses iterative deepening to search the computation tree for safety property violations. At each point in its exploration of the tree, OFMC checks if any safety property has been violated. To do this, OFMC tries to select concrete values for messages so that the messages satisfy the constraints (discussed in the previous paragraph) but result in a violation of the property. This iterative-deepening search is guaranteed to terminate if an attack on the protocol (that is, a safety property violation) exists. This search is also guaranteed to terminate if the number of protocol sessions is finite (since, in this case, the depth of computation tree is also finite).

When OFMC finds an attack, it outputs the goal that the attack violates and a trace of the attack.

6 Hermes

Hermes [BLP03a, BLP03b, BLP02, JM01] is a model checker for security protocols, developed by the VERIMAG lab at IMAG. Currently, Hermes is only able to check secrecy properties of protocols. Hermes has been tested on 15 of the protocols in the Clark/Jacob library. Eight of these protocols are known to have attacks (we are only considering violations of secrecy properties), and Hermes is able to find attacks for six of them [BLP03a]. The attacks on the other two flawed protocols involve the use of non-atomic keys, which Hermes is currently unable to handle.

Hermes uses the language EVA for protocol specification. EVA is similar to HLPSL (OFMC's specification language). As with HLPSL, an EVA specification must state:

- the initial knowledge of each agent participating in the protocol
- the initial knowledge of the intruder
- the sequence of messages required by the protocol
- an initial set of sessions (or instantiations) of the protocol
- the goals that we wish the protocol to achieve

To specify the initial knowledge of the intruder in EVA, we state what the intruder does *not* know (rather than stating what the intruder does know, as in HLPSL). When specifying the initial set of sessions, we can allow for a finite or infinite number of sessions running in parallel. As in HLPSL, the goals of the protocol can include the authentication of agents and the secrecy of information. Unlike HLPSL, the intruder's abilities are not stated in an EVA specification - the intruder is always able to eavesdrop, intercept messages, and impersonate other agents. An EVA specification of the NSPK protocol is provided in Figure 2.

Hermes is able to handle only a subset of the protocols that can be specified in EVA. In particular, Hermes cannot be used with protocols that involve XOR encryption or exponentiation, even though these protocols can be expressed in EVA. Furthermore, Hermes can only check secrecy properties - it cannot check other goals (such as authentication) that can be expressed in EVA. EVA also has its own limitations. For example, the use of non-atomic keys (keys that are constructed from two or more "regular" keys) cannot be specified in EVA.

Like OFMC, Hermes uses the Dolev-Yao model of the intruder. This results in a transition system that has an infinite branching factor. Furthermore, since there can be infinitely many sessions running in parallel, the computation tree that corresponds to the transition system has infinite depth. Hermes uses abstraction to deal with this. First of all, it assumes that there is a single honest agent (say agent A) and a single intruder (called I). It can be shown that an attack exists under this assumption if and only if an attack exists when there are multiple honest agents and intruders [BLP02]. Suppose the protocol being verified involves two roles. Then, Hermes considers three types of sessions - sessions where A plays both roles, sessions where A plays the first role and I plays the second role, and sessions where I plays the first role and A plays the second role.

Hermes constructs a set of rules that describe the transitions corresponding to each type of session. That is, the rules describe the message that can be produced during a run of each type of session. Hermes uses the rules to find a set of "good" message patterns and a set of "bad" message patterns with respect to a set of secrets S. While finding these message patterns, Hermes augments S if necessary (in order for elements of S to remain secret there may be other pieces of information that must also remain secret). Intuitively, good message patterns represent messages that protect the secrets, and bad message patterns represent

```
Needham_Schroeder_Public_Key

   alg :  asym_algo
   everybody knows alg

   A, B: principal

   Na, Nb :  number

   keypair^alg PK, SK (principal)
   everybody knows PK

   A knows A, B, SK(A)
   B knows B, SK(B)

   {
     1.  A -> B : {A, Na}_(PK(B))^alg
     2.  B -> A : {Na, Nb}_(PK(A))^alg
     3.  A -> B : {Nb}_(PK(B))^alg
   }

   s1.  session *{A,B} A=A, B=B

   assume
     secret (SK(A)@s1.A), secret (SK(B)@s1.B),
     secret (SK(B@s1.A)), secret (SK(A@s1.B))

   claim
     *A*G secret (SK(A)@s1.A),
     *A*G secret (SK(B)@s1.B),
     *A*G secret (Na@s1.A),
     *A*G secret (Nb@s1.B)
```

Figure 2: EVA specification of the NSPK protocol

messages from which the intruder is able to find a secret. A given message that contains a secret is good if and only if it unifies with a good message pattern and does not unify with any bad message pattern.

Fixpoint iteration is used to find the set of all bad message patterns and the augmented set of secrets. The main idea here is that a message is bad, if using one of the rules, it can be used to produce a message that is already known to be bad. The fixpoint iteration stops when no more bad message patterns and no more secrets can be added. The protocol satisfies the specified secrecy properties if no secret (from the augmented set of secrets) appears in a bad message in the specified initial set of sessions. However, it is possible that the fixpoint iteration does not terminate. That is, the set of bad message patterns grows without bound. Hermes is able to detect the existence of rules that may lead to an infinite set of bad message patterns. So, rather than simply running fixpoint iteration, Hermes uses an algorithm that produces an overapproximation to the set of bad message patterns when there exist rules that may cause fixpoint iteration to not terminate. Note that overapproximating the set of bad message patterns does not eliminate any attacks on the protocol, but may lead to spurious attacks. However, when Hermes was tested with protocols from the Clark/Jacob library, it did not produce any spurious attacks.

When Hermes finds an attack, it outputs a trace of the attack. Whether or not an attack is found, Hermes outputs the sets of good and bad message patterns, along with the augmented set of secrets.

7 Maude

Maude [DMT98, BD01, CDE+03] is the result of a collaboration of researchers from SRI International, Universidad de Málaga(Spain), Universidad Complutense de Madrid(Spain), and the University of Illinois. Maude is not exclusively a security protocol model-checker, but instead is an executable specification language which is based on rewriting logic [Mes92], which makes Maude much more expressive then OFMC and Hermes. Maude uses both an equation style of functional specification and an object-oriented specification style for objects in the system. These objects can execute concurrently in both a synchronous and asynchronous way and can also be non-deterministic in nature. The modules in Maude are rewrite theories whose axioms are rewrite rules (rewrite rules can be thought of as the transition rules of the system being modeled), which define how Maude is suppose to go from one state of the modeled system to another.

Maude can be used to find Lowe's "Man in the Middle" attack on the NSPK protocol, but we do not provide the Maude code for this due to its length. Instead, we will discuss some of the Maude concepts which are used to create such a model.

Like many other object-oriented languages, Maude uses classes to define objects. A class is defined by "`class C | att1: s1 ... attn: sn`" which defines class C with attributes $att1, ..., attn$ of corresponding types $s1, ..., sn$. Objects of this class are of the form "`<N; C | att1: v1 ... attn:vn>`" where N is the name of the object and $v1, ..., vn$ are the current values the object has for its attributes.

Objects can interact with each other through messages. Messages are defined using the syntax "`msg M: m1 ... mn -> Message`" where the name of the message is M, and the message has n parameters whose types are $m1, ..., mn$.

The state of a modeled system in Maude is called a *configuration*. The state contains the current values of both the objects present in the system, and the current value of the messages. Transitions between states are defined by rewrite rules.

Each rewrite rule maps a given configuration C to another configuration C'. These rules can be based on some condition *cond*. Conditional rules have the syntax "`crl [label] : C => C' if cond`" and unconditional rules have the syntax "`rl [label] : C => C'`". These rewrite rules also play the role of a Kripke structure which allows temporal logic properties to be asserted in LTL.

Maude provides many verification tools, and although none of them are specifically for security protocols, they are very useful for verification in general. The tools include:

- linear temporal logic (LTL) property checker for finite-state system modules

- inductive theorem prover (ITP) to verify properties of functional modules
- Church-Rosser checker
- Knuth-Bendix completion tool and termination checker
- coherence checker for system modules.

Maude is a very powerful specification language with a high level of expressibility. For general model checking, Maude is useful because of its ability to express large and complex systems. However, since Maude is not specifically designed for security protocols, we find that specifying a security protocol in Maude is very difficult relative to the other tools we have examined. In particular, unlike the other tools, Maude requires the user to define the search strategy when a security protocol is being verified.

8 Comparing the model-checkers

Like SMV and Promela, Maude is general specification language, and hence its expressive power is greater than that of HLPSL and EVA. However, HLPSL and EVA are specifically designed for modeling security protocols, and as a result, security protocols can be expressed far more concisely in these languages than in Maude. This can be seen in Table 1 - the Maude specification of the NSPK protocol is over 30 times longer than the HLPSL and EVA specifications. Nevertheless, all three model checkers do find Lowe's attack.

Model-checker	**Lines of code**	**Running time**	**Machine Used**
OFMC	19	0.0 seconds	1.4 GHz Pentium III
Hermes	23	0.04 seconds	600 MHz Pentium III
Maude	727	96 minutes	300 MHz, processor type not given

Table 1: Performance of model-checkers on NSPK protocol [BMV03, BLP02, DMT98, BD01]

Hermes appears to be in the early stages of development. It is missing some desirable features that are in fact available in OFMC (some of these are discussed in Section 6). As another sign of the infancy of Hermes, we found that there was less documentation available for EVA syntax than desired.

Of the three model-checkers, we find OFMC to be the best choice for modeling security protocols. OFMC allows many different protocols to be concisely specified, and allows a variety of goals to be efficiently verified. As a result, we decided to use OFMC to verify the SSL Handshake Protocol.

9 Modeling the SSL Handshake Protocol

9.1 The SSL Protocol

Netscape Communications developed SSL (Secure Sockets Layer) [FKK96] to provide a secure way for clients and servers to communicate. We examine SSL version 3.0. The Internet draft of this version was released in 1996.

The security provided by SSL has the following three properties:

1. *A private connection*: During the handshake, the client and server agree on the cryptographic algorithms and keys to use. Subsequently, all communication between the client and the server is encrypted using these algorithms and keys.

2. *Identities of the parties are authenticated*: The parties authenticate each other using certificate verification.

3. *Reliable connection*: The integrity of each message is verified, using secure hash functions.

In this paper we examine the SSL Handshake protocol.

9.2 The SSL Handshake Protocol

We give an overview of the SSL Handshake Protocol under the assumption that the client is initiating a new session and that the server possesses a certificate. For a detailed description of the protocol, see [FKK96]. The purpose of the SSL Handshake Protocol is to allow the server and the client to authenticate each other and to agree on both an encryption algorithm and the cryptographic keys used to transmit data.

The protocol works as follows:

The client initiates a new handshake by sending a *hello* message to the server, which responds with its own *hello* message:

- The client *hello* includes the protocol version supported by the client, a new random number generated by the client, and the lists of the cryptographic suites and compression methods which the client is able to use.

- The server *hello* includes the protocol version supported by the server, a new random number generated by the server, the session id for this new session, and selections from the cryptographic suites and compression methods lists provided by the client.

The server now sends its *certificate* to the client, along with a *certificate request*, if the server wishes to authenticate the client, and finally a *server hello done* message.

If the server has requested a certificate, the client must send its certificate or a *no certificate* message to the server. The client then sends a *client key exchange* message to the server. The client key exchange message includes information that can be use to construct a key of the type agreed upon during the *hello* messages exchange. The client key exchange message is followed by a *certificate verify* message if the client sent a certificate.

Finally, the client sends a *change cipher spec* message followed by a *finished* message to the server and now uses the agreed-upon algorithms and keys. The server sends a *change cipher spec* message followed by a *finished* message and now uses the agreed-upon algorithms and keys. Note that the client's and server's finished messages are encrypted by the agreed-upon algorithms and keys. At this point, the handshake is complete.

We modeled the SSL Handshake Protocol with RSA used for key exchange. We considered the following two settings for the Handshake Protocol:

- In the first setting, the client and server both have certificates and the server does request a certificate from the client. In this setting, the client and server want to authenticate each other and agree on a shared secret (known as the *premaster secret*). We refer to this setting as `SSL_Handshake_With_Client_Authentication`. Our HLPSL specification for this setting can be found in Appendix A.1.

- In the second setting, the server has a certificate but the server does not request a certificate from the client. In this setting, the client wants to authenticate the server, and the client and the server want to agree on a shared secret (known as the *premaster secret*). We refer to this setting as `SSL_Handshake_Without_Client_Authentication`. Our HLPSL specification for this setting can be found in Appendix A.2.

9.3 Environmental assumptions

We assume that there exist two honest agents, the client and the server, who are willing to communicate with each other. In addition, we assume that there exists a single dishonest agent, known as the intruder. As is standard when model-checking security protocols, we assume that the cryptography and hashing used by the protocol were perfect. That is, an encrypted message cannot be decrypted without having the appropriate key, and the output of hash functions cannot be inverted. Also, as is standard, we assume that the intruder has full control of the network. To maximize the intruder's ability to get information, we assume that the client is willing to connect to the intruder and the server is willing to accept connections from the intruder, though neither the client nor the server know that the intruder is a dishonest agent.

9.4 Modeling decisions

When modeling the Handshake Protocol, we omitted messages whose content did not change from one run of the protocol to the next. For example, we omitted the *server hello done* message. In practice, the main purpose of such messages is simply to let an agent know that it is their turn to communicate. However, even without such messages, the syntax of HLPSL makes it clear when it is each agent's turn to communicate. Furthermore, by their very nature, such messages have no effect on the security of the protocol.

Similarly, we omitted fields of messages when the content of such fields did not change from one run of the protocol to the next. For example, we omitted the *pad1* and *pad2* fields from the *finished* messages. In practice, the main purpose of such fields is to act as a delimiter between other fields or to ensure that a message has a particular size. However, in the syntax of HLPSL, we simply use commas as a delimiter between fields, and there is no notion of message size. Furthermore, as in the case of fixed-content messages, fixed-content fields have no effect on the security of the protocol.

According to the Handshake Protocol, two hash functions (MD5 and SHA) are used in the *certificate verify* and *finished* messages. These two hash functions are used in an identical way, and it appears that this duplication is used as a form of redundancy. Since we are assuming that hash functions are perfect, there is nothing to be gained (or lost) from such duplication. As a result, we decided to omit this duplication.

One of the goals of the Handshake Protocol is to establish a shared secret (the *premaster secret*) that can be used to produced a key used for symmetric encryption. In practice, a *master secret* is computed from the premaster secret, and the key is computed from the master secret (in a way that depends on the symmetric encryption algorithm that is agreed upon). Both parties perform these computations independently once the premaster secret has been established. However, in HLPSL, any number can be used as a key for symmetric encryption. So, to simplify our model, we decided to use the premaster secret as both the master secret and as the key used for symmetric encryption.

9.5 Verified properties

For the `SSL_Handshake_With_Client_Authentication` model, we verified the following properties:

1. The server sends its public key to the client, and the client receives this key intact (that is, the key that the client receives is identical to the key that is sent by the server). In other words, the client authenticates the server's public key.

2. The client sends its public key to the server, and the server receives this key intact (that is, the key that the server receives is identical to the key that is sent by the client). In other words, the server authenticates the client's public key.

3. The premaster secret is known only by the client and the server.

For the `SSL_Handshake_Without_Client_Authentication` model, we verified properties 1 and 3 from above but not property 2 (since the server does not receive the client's public key in this setting).

9.6 Results of verification

For the `SSL_Handshake_With_Client_Authentication` model, all properties were successfully verified. The verification took 25.3 seconds and used 506 MB of memory.

For the `SSL_Handshake_Without_Client_Authentication` model, OFMC terminated after 161 minutes without providing a result. To deal with this, we modified the model so that the server was no longer willing to accept connections from the intruder (though the intruder could still connect to the server by pretending to be the client). That is, we removed the line `[C:I;S:s;Ks:ks;Kca:kca;H:h]` in the Session_instances section of the HLPSL specification.

Then, all properties were successfully verified. The verification took 0.82 seconds and used 3.5 MB of memory.

10 Conclusion

Model-checking is a powerful technique for verifying protocol security. The Lowe attack on the NSPK protocol was discovered 18 years after the protocol's introduction, but the three model checking tools we have presented can find this attack in minutes, or even seconds. This suggests that more extensive use of model-checking can lead to greater confidence in the security of protocols, such as SSL, that are widely used today. Furthermore, in the future, protocols should be verified using model-checkers before they are put into use.

References

[ABB+03] Alessandro Armando, David Basin, Mehdi Bouallagui, Yannick Chevalier, Luca Compagna, Sebastian Mödersheim, Michael Rusinowitch, Mathieu Turuani, Luca Viganò, and Laurent Vigneron. AVISS final project report (draft). World Wide Web, `http://www.inf.ethz.ch/~moederss/research/ofmc-1.2.tgz`, 2003.

[BD01] David Basin and Grit Denker. Maude versus Haskell: an experimental comparison in security protocol analysis. In Kokichi Futatsugi, editor, *Electronic Notes in Theoretical Computer Science*, volume 36, Amsterdam, 2001. Elsevier Science Publishers.

[BLP02] Liana Bozga, Yassine Lakhnech, and Michaël Périn. Abstract interpretation for secrecy using patterns. Technical Report 5, EVA, 2002.

[BLP03a] Liana Bozga, Yassine Lakhnech, and Michaël Périn. HERMES: An automatic tool for verification of secrecy in security protocols. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Computer Aided Verification, 15th International Conference, CAV 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*. Springer, 2003.

[BLP03b] Liana Bozga, Yassine Lakhnech, and Michaël Périn. Pattern-based abstraction for verifying secrecy in protocols. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Proceedings*, volume 2619 of *Lecture Notes in Computer Science*. Springer, 2003.

[BMV03] David Basin, Sebastian Mödersheim, and Luca Viganò. An on-the-fly model-checker for security protocol analysis. Technical Report 404, Swiss Federal Institute of Technology Zurich, 2003.

[CDE+03] Manuel Clavel, Francisco Duranàn, Steven Eker, Patrick Lincoln, Narciso Martì-Oliet, Josè Meseguer, and Carolyn Talcott. Maude 2.0 manual. World Wide Web, `http://maude.cs.uiuc.edu`, 2003.

[CJ97] John A. Clark and Jeremy L. Jacob. A survey of authentication protocol literature, version 1.0. World Wide Web, `http://www.cs.york.ac.uk/~jac/papers/drareview.ps.gz`, 1997.

[DLMS99] N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. Undecidability of bounded security protocols. In N. Heintze and E. Clarke, editors, *Proceedings of the Workshop on Formal Methods and Security Protocols — FMSP, Trento, Italy*, July 1999.

[DMT98] G. Denker, J. Meseguer, and C. Talcott. Protocol specification and analysis in Maude, 1998.

[DY83] D. Dolev and A.C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, March 1983.

[FKK96] Alan O. Freier, Philip Karlton, and Paul C. Kocher. SSL 3.0 specification. World Wide Web, `http://wp.netscape.com/eng/ssl3`, 1996.

[JM01] Florent Jacquemard and Daniel Le Métayer. Langage de spécification de protocoles cryptographiques de EVA : syntaxe concrète. Technical Report 1, EVA, 2001.

[Low96] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1055, pages 147–166. Springer-Verlag, 1996.

[Mes92] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[Möd03] Sebastian Mödersheim. OFMC. World Wide Web, `http://www.inf.ethz.ch/~moederss/research`, 2003.

[NS78] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

A HLPSL specifications

A.1 SSL_Handshake_With_Client_Authentication

```
PROTOCOL SSL_Handshake_With_Client_Authentication;

Identifiers

  % The client and the server.
  C,S: user;

  % Session I.D., generated by the server.
  SID: number;

  % A random number generated by the client.
  Rc: number;

  % A random number generated by the server.
  Rs: number;

  % The pre-master secret - a random number generated by the client.
  Pm: symmetric_key;

  % The client's public key.
  Kc: public_key;

  % The server's public key.
  Ks: public_key;

  % The public key of the certificate authority.
  Kca: public_key;

  % Hash functions.
  H: function;


Knowledge

  % The client's knowledge. Note that {C,Kc}Kca' is the client's certificate.
  C: C,S,{C,Kc}Kca',Kc,Kc',Kca,H;

  % The server's knowledge. Note that {S,Ks}Kca' is the server's certificate.
  S: S,{S,Ks}Kca',Ks,Ks',Kca,H;
```

```
Messages

 1. C->S:

   % The client's hello message. Since we assume that the client is using SSL
   % version 3.0 and does not wish to reuse security parameters from another
   % session, we omit the client_version and session_id fields. Furthermore,
   % since we assume that the CipherSuite is fixed (in particular, RSA is used
   % for key exchange) and that no compression is used, we omit the
   % cipher_suites and compression_methods fields.
   Rc


 2. S->C:

   % The server's hello message. Since we assume that the server is using SSL
   % version 3.0, we omit the server_version field. For the reasons described
   % previously, we omit the cipher_suites and compression_methods fields.
   (Rs, SID),

   % The server certificate.
   {S,Ks}Kca'


 3. C->S:

   % The client certificate.
   {C,Kc}Kca',

   % The client key exchange message. We omit the gmt_unix_time field since
   % timestamps are not supported by OFMC.
   {Pm}Ks,

   % The "certificate verify" message. Since we are assuming that our hash
   % functions are perfect, we hash using a single hash function (H) rather
   % than using both the MD5 and SHA hash functions.
   {H(Pm,
      H(Rc, (Rs, SID) ,{S,Ks}Kca',{C,Kc}Kca',{Pm}Ks,
        Pm))}Kc',

   % The client's "finished" message. Since we are assuming that our hash
   % functions are perfect, we hash using a single hash function (H) rather
   % than using both the MD5 and SHA hash functions.
```

```
  {H(Pm,
     H(Rc, (Rs, SID),{S,Ks}Kca',{C,Kc}Kca',{Pm}Ks,
       {H(Pm,
          H(Rc, (Rs, SID) ,{S,Ks}Kca',{C,Kc}Kca',{Pm}Ks,
            Pm))}Kc',
       C,Pm))
  }Pm


4. S->C:

  % The server's "finished" message. Since we are assuming that our hash
  % functions are perfect, we hash using a single hash function (H) rather
  % than using both the MD5 and SHA hash functions.
  {H(Pm,
     H(Rc, (Rs, SID),{S,Ks}Kca',{C,Kc}Kca',{Pm}Ks,
       {H(Pm,
          H(Rc, (Rs, SID) ,{S,Ks}Kca',{C,Kc}Kca',{Pm}Ks,
            Pm))}Kc',
       {H(Pm,
          H(Rc, (Rs, SID),{S,Ks}Kca',{C,Kc}Kca',{Pm}Ks,
            {H(Pm,
               H(Rc, (Rs, SID) ,{S,Ks}Kca',{C,Kc}Kca',{Pm}Ks,
                 Pm))}Kc',
            C,Pm))
       }Pm,
       S,Pm))
  }Pm


Session_instances

  % The client (agent c) is willing to connect to the server (agent s) and the
  % intruder (although the client doesn't know that the intruder is a dishonest
  % agent).
  % The server is willing to accept connections from the client and the
  % intruder (although the server doesn't know that the intruder is a dishonest
  % agent).
  [C:c;S:s;Kc:kc;Ks:ks;Kca:kca;H:h]
  [C:c;S:I;Kc:kc;Ks:ki;Kca:kca;H:h]
  [C:I;S:s;Kc:ki;Ks:ks;Kca:kca;H:h];


% The intruder has complete control of the network.
```

```
Intruder divert, impersonate, eaves_dropping;


Intruder_knowledge I,c,s,kc,ks,ki,ki',kca,h;


% The properties we are verifying.

% S exists, S sent Ks to C, and C received Ks as it was sent by S (that is,
% the value of Ks that was received by C is identical to the value of Ks that
% was sent by S).
Goal C authenticate S on Ks;

% C exists, C sent Kc to S, and S received Kc as it was sent by C (that is,
% the value of Kc that was received by S is identical to the value of Kc that
% was sent by C).
Goal S authenticate C on Kc;

% Only C and S know the value of Pm.
Goal secrecy_of Pm;
```

A.2 SSL_Handshake_Without_Client_Authentication

```
PROTOCOL SSL_Handshake_Without_Client_Authentication;

Identifiers

  % The client and the server.
  C,S: user;

  % Session I.D., generated by the server.
  SID: number;

  % A random number generated by the client.
  Rc: number;

  % A random number generated by the server.
  Rs: number;

  % The pre-master secret - a random number generated by the client.
  Pm: symmetric_key;

  % The server's public key.
  Ks: public_key;
```

```
  % The public key of the certificate authority.
  Kca: public_key;

  % Hash functions.
  H: function;


Knowledge

  % The client's knowledge.
  C: C,S,Kca,H;

  % The server's knowledge. Note that {S,Ks}Kca' is the server's certificate.
  S: S,{S,Ks}Kca',Ks,Ks',Kca,H;


Messages

  1. C->S:

    % The client's hello message. Since we assume that the client is using SSL
    % version 3.0 and does not wish to reuse security parameters from another
    % session, we omit the client_version and session_id fields. Furthermore,
    % since we assume that the CipherSuite is fixed (in particular, RSA is used
    % for key exchange) and that no compression is used, we omit the
    % cipher_suites and compression_methods fields.
    Rc


  2. S->C:

    % The server's hello message. Since we assume that the server is using SSL
    % version 3.0, we omit the server_version field. For the reasons described
    % previously, we omit the cipher_suites and compression_methods fields.
    (Rs, SID),

    % The server certificate.
    {S,Ks}Kca'


  3. C->S:

    % The client key exchange message. We omit the gmt_unix_time field since
```

```
    % timestamps are not supported by OFMC.
    {Pm}Ks,

    % The client's "finished" message. Since we are assuming that our hash
    % functions are perfect, we hash using a single hash function (H) rather
    % than using both the MD5 and SHA hash functions.
    {H(Pm,
       H(Rc, (Rs, SID),{S,Ks}Kca',{Pm}Ks,
         C,Pm))
    }Pm


  4. S->C:

    % The server's "finished" message. Since we are assuming that our hash
    % functions are perfect, we hash using a single hash function (H) rather
    % than using both the MD5 and SHA hash functions.
    {H(Pm,
       H(Rc, (Rs, SID),{S,Ks}Kca',{Pm}Ks,
         {H(Pm,
            H(Rc, (Rs, SID),{S,Ks}Kca',{Pm}Ks,
              C,Pm))
         }Pm,
         S,Pm))
    }Pm


Session_instances

  % The client (agent c) is willing to connect to the server (agent s) and the
  % intruder (although the client doesn't know that the intruder is a dishonest
  % agent).
  % The server is willing to accept connections from the client and the
  % intruder (although the server doesn't know that the intruder is a dishonest
  % agent).
  [C:c;S:s;Ks:ks;Kca:kca;H:h]
  [C:c;S:I;Ks:ki;Kca:kca;H:h]
  [C:I;S:s;Ks:ks;Kca:kca;H:h];


% The intruder has complete control of the network.
Intruder divert, impersonate, eaves_dropping;
```

```
Intruder_knowledge I,c,s,ks,ki,ki',kca,h;


% The properties we are verifying.

% S exists, S sent Ks to C, and C received Ks as it was sent by S (that is,
% the value of Ks that was received by C is identical to the value of Ks that
% was sent by S).
Goal C authenticate S on Ks;

% Only C and S know the value of Pm.
Goal secrecy_of Pm;
```