# Using SPIN to Verify Security Properties of Cryptographic Protocols

Paolo Maggi and Riccardo Sisto

Dip. di Automatica e Informatica - Politecnico di Torino
Corso Duca degli Abruzzi 24, I-10129 Torino, ITALY
maggi@athena.polito.it, sisto@polito.it

**Abstract.** This paper explores the use of Spin for the verification of cryptographic protocol security properties. A general method is proposed to build a Promela model of the protocol and of the intruder capabilities. The method is illustrated showing the modeling of a classical case study, i.e. the Needham-Schroeder Public Key Authentication Protocol. Using the model so built, Spin can find a known attack on the protocol, and it correctly validates the fixed version of the protocol.

## 1 Introduction

All the solutions adopted to ensure security properties in distributed systems are based on some kind of cryptographic protocol which, in turn, uses basic cryptographic operations such as encryption and digital signatures. Despite their apparent simplicity, such protocols have revealed themselves to be very error prone, especially because of the difficulty generally found in foreseeing all the possible attacks. For this reason, researchers have been working on the use of formal verification techniques to analyze the vulnerability of such protocols.

Both the theorem proving and the model checking approaches have been investigated. Some of the researchers who have investigated the model checking approach have developed specific model checkers for cryptographic protocols (e.g. [1]), whereas others have shown how general purpose tools such as FDR [2] and Murphi [3] can be used for the same purpose. In this paper, we follow the latter kind of approach, and we explore the possibility of using Spin, which is one of the most powerful general purpose model checkers, to verify cryptographic protocols. Instead of simply porting the approaches developed by other researchers to the Spin environment, we develop a new approach which makes use of static analysis techniques in order to get simpler models. The main idea is that the protocol configuration to be checked (i.e. the protocol sessions included in the model) can be statically analyzed in order to collect data-flow information, which can be used to simplify the intruder knowledge representation. For example, such a preliminary analysis can identify which data can potentially be learned by the intruder and which cannot, thus avoiding the representation of knowledge elements that will never occur. Similarly, it is possible to foresee which messages that the intruder could build will never be accepted as valid by any protocol agent, and avoid their generation.

The use of Spin for cryptographic protocol verification has already been proposed and discussed in [4], where, however, the author does not give a concrete proposal, but just some general ideas and evaluations of the complexity of the verification task.

This paper is organized as follows. In section 2, we briefly introduce the Needham-Schroeder Public Key Authentication Protocol, which will be used throughout the article to illustrate our modeling approach. In section 3, we present the basic choices and the main underlying principles of our modeling approach, whereas, in section 4, we give a detailed description of the procedure to build a Promela model of an instance of a cryptographic protocol, using the sample protocol as an example. In section 5, we present verification results related to the sample protocol. Section 6 concludes.

## 2 The Needham-Schroeder Public Key Protocol

The Needham-Schroeder Public Key Protocol [5] is a well known authentication protocol that dates back to 1978. It aims to establish mutual authentication between an *initiator* $A$ and a *responder* $B$, after which some session involving the exchange of messages between $A$ and $B$ can take place.

As its name clearly suggests, the protocol uses public key cryptography [6, 7]. Each agent $H$ possesses a *public key*, denoted $PK(H)$, and a *secret key* $SK(H)$, which can be used to decrypt the messages encrypted with $PK(H)$. While $SK(H)$ should be known only by $H$, any other agent can obtain $PK(H)$ from a key server. Any agent can encrypt a message $x$ using $H$'s public key to produce the encrypted message $\{x\}PK(H)$. Only the agents that know $H$'s secret key can decrypt this message in order to obtain $x$. This property should ensure $x$ secrecy. At the same time, any agent $H$ can sign a message $x$ by encrypting it with its own secret key, $\{x\}SK(H)$, in order to ensure its integrity. Any agent can decrypt $\{x\}SK(H)$, using $H$'s public key.

The complete Needham-Schroeder Public Key Protocol [5] involves seven steps and it is described in figure 1, where $A$ is an *initiator* agent who requests to establish a session with a *responder* agent $B$ and $S$ is a trusted key server.

Any run of the protocol opens with $A$ requesting $B$'s public key to the trusted key server $S$ (step 1). S responds sending the message 2. This message, signed by $S$ to ensure its integrity, contains $B$'s public key, $PK(B)$, and $B$'s identity. If $S$ is trusted, this should assure A that $PK(B)$ is really $B$'s public key. It is worth noting that the protocol assumes that $A$ can obtain $PK(A)$, needed to decrypt message 2, in a reliable way. If this assumption is not true, an intruder could try to replace $S$ providing an arbitrary value that $A$ thinks to be $PK(A)$. Note also, that, as pointed out in [8], there is no guarantee that $PK(B)$ is really the current $B$'s public key, rather that a replay of an old and compromised key (however this attack can be easily prevented using timesptamps [8])

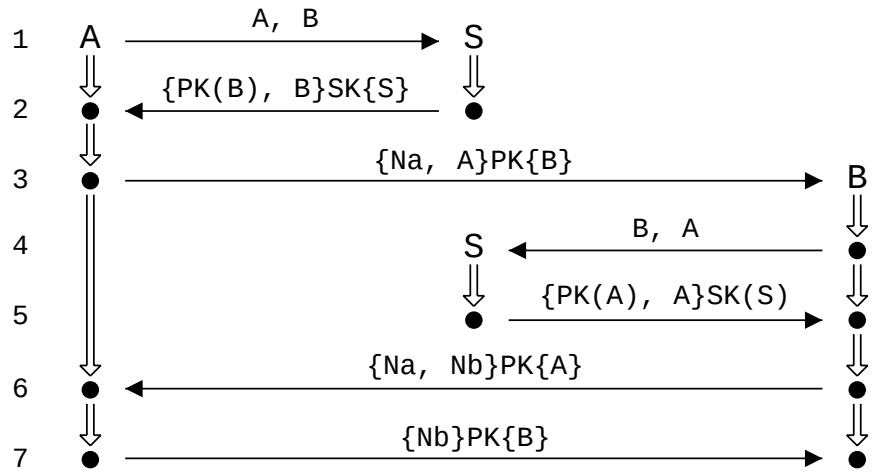


**Fig. 1.** The complete Needham-Schroeder Public Key Protocol.

Once obtained $B$'s public key, $A$ selects a *nonce*[1] $Na$ and sends message 3 to $B$. This message can only be understood by $B$, beeing encrypted with its public key, and indicates that someone, supposed to be $A$, whishes to autheticate himself to $B$.

After having received message 3, $B$ decrypts it using its secret key to obtain the nonce $Na$ and then requests $A$'s public key to $S$ (steps 4 and 5).

At this point, $B$ sends the nonce $Na$ to $A$, along with a new nonce $Nb$, encrypted with $A$'a public key (message 6). With this message, $B$ authenticates itself to $A$, since, receiving it, $A$ is sure that it is communicating with $B$, being $B$ the only agent that should be able to obtain $Na$ decrypting message 3.

To finish the protocol run, $A$ returns the nonce $Nb$ to $B$ in order to authenticate itself to $B$ (message 7).

Looking at the protocol, it is easy to observe as four of the seven steps can be removed if we assume that $A$ and $B$ already know each other's public keys. Indeed, messages 1, 2, 4 and 5 make up a protocol that aims to obtain $A$'s and $B$'s public keys from a trusted key server $S$, whereas messages 3, 6 and 7 make up the real authentication protocol.

In the following of this paper, we will assume that all the agents already know each other's public keys and so we focus our attention on the reduced protocol obtained removing messages 1, 2, 4 and 5 and described in figure 2.

[1] A *nonce* is a random number generated with the purpose to be used in a sinlge run of the protocol.

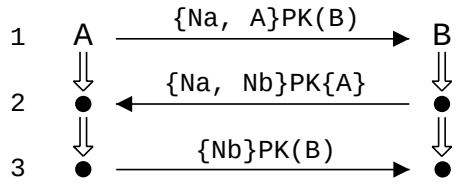


**Fig. 2.** The reduced Needham-Schroeder Public Key Protocol.

## 3 Modeling cryptographic protocols with PROMELA

Formal models of cryptographic protocols are typically composed of a set of principals which send messages to each other according to the protocol rules, and an intruder, representing the activity of possible attackers.

Since such models are meant to reveal possible security flaws in the protocols and not flaws in the cryptosystems used by the protocols, cryptography is modeled in a very abstract way and it is assumed to be "perfect". This means that:

- the only way to decrypt an encrypted message is to know the corresponding key;
- an encrypted message does not reveal the key that was used to encrypt it;
- there is sufficient redundancy in messages so that the decryption algorithm can detect whether a ciphertext was encrypted with the expected key.

Although such assumptions are obviously not completely true for real cryptosystems, they represent the properties of an ideal cryptosystem, so they are useful to isolate the flaws of the protocol itself. In other words, any flaw found with this model is a real protocol flaw, but it is possible that the model does not reveal other weaknesses due to the used cryptosystems.

Both the principals and the intruder are represented in our models by means of Promela processes that communicate with each other through shared channels. More precisely, we have a different process definition for each protocol role, and a process definition for the intruder. Principals do not communicate with each other directly but all the messages they send are intercepted by the intruder which eventually will forward them to the right addressee. This approach has been followed by other researchers too (e.g. [1]) and avoids redundant execution paths.

The intruder can interact with the protocol in any way we would expect a real-word attacker to be able to do, but at the same time it is also able to behave like a normal user of the computer network. For this reason other principals may initiate protocol runs with it. It is even possible that the intruder behaves as a set of different cooperating malicious users.

At any instant, the behavior of the intruder depends on the knowledge it has acquired. Before a protocol run starts, it is assumed that the intruder knows only a given set of data. For example, such data normally include the intruder

identity(ies), its public and private keys, the identity of the other principals, their public keys, and, possibly, any secret keys the intruder shares with other principals.

Every time the intruder intercepts a message, it can increase its knowledge. Indeed, if the intercepted message or part of it is encrypted and the intruder knows the decryption key, it can decrypt it and learn its contents. Otherwise, if the intruder is not able to decrypt the intercepted message or parts of it, it can remember the encrypted components even if it cannot understand them. Since we are interested in modeling the most powerful intruder, we assume it always learns as much as possible from the intercepted messages.

Of course, besides intercepting messages the intruder can also forge and send new ones into the system. These ones are created using all the items it currently knows. Since it is normally assumed that the intruder can also create new data items from scratch (e.g. nonces), and use them to forge new messages, this capability can be represented including one or more generic distinguished data items in the initial intruder knowledge. Such data items represent the ones the intruder will generate from scratch during the protocol runs.

Note that, even if a message can be forged by the intruder, it could be that it cannot be accepted by the receiving principal. This fact can be exploited to safely restrict the messages the intruder can generate, excluding the ones that will not be accepted by the receiving processes.

In order to have finite models, we follow the common practice of putting some restrictions on the modeled behaviors. First of all, the model can represent only a finite number of parallel protocol sessions. Each principal can be engaged in a finite number of runs of the protocol, each run being modeled by a different instance of the corresponding process definition. Nonces created during a protocol run are represented as process parameters, and are assigned different actual values at each process instantiation.

Another possible source of infinite behaviors is the intruder. In fact, it can in principle forge and send infinite different messages. A typical solution to this problem is to restrict the way the intruder can generate messages, e.g. by restricting the maximum complexity of the generated messages [2, 1] and by limiting the number of different data items it can generate by scratch. More recent works [9] have shown that such restrictions can be avoided by representing the messages that can be generated by the intruder symbolically, and by specializing them as soon as the receiving process performs some checks on them. In this paper we follow a simplified symbolic approach inspired by the one reported in [9], which consists of symbolically representing all the data items the intruder can generate from scratch as well as all the other complex data items it can generate from its knowledge by means of a single distinguished symbolic identifier. Such an identifier is always part of the intruder knowledge. However, instead of saying that the intruder always sends a generic message which is then specialized, we statically determine all the possible specializations it can take, which are finite, and we use the symbolic identifier only for the leaves of the possible message structures.

## 4 Building the Promela model

In this section we describe a procedure to build a Promela model of a cryptographic protocol instance, to be used for security property verification. Such a procedure is illustrated using the reduced Needham-Schroeder Public Key Authentication Protocol as an example. The Promela model we build can be divided into two parts:

- the description of the protocol rules and of the protocol instance
- the description of the intruder behavior

The first part is quite simple and should be written manually, whereas the second part can be generated automatically.

The instance of the sample protocol we deal with is fairly small. It includes three principals: $A$, $B$ and $I$. $A$ plays the role of the *initiator*, whereas principal $B$ plays the role of the *responder*. $I$ is one of the possible identities of the intruder, so principal $I$ can play any role.

### 4.1 The protocol instance model

The first step in the construction of the model is the definition of the finite set of *names*. With the term name we mean any distinguished identifier, key, nonce or data used in the protocol. As already explained, we also use a special name which symbolically represents nonces and other generic data items generated by the intruder. For our specific sample case, we need a name for each identity and a name for each nonce. The resulting set of names we will use is then defined as follows:

```
mtype = {A, B, I, Na, Nb, gD, R};
```

where `A`, `B` and `I` are the identities we consider, `Na` is the nonce generated by principal $A$, `Nb` the nonce generated by $B$ and `gD` the symbolic representation of a generic data item used by the intruder. `R` is a service costant we will use in the intruder process definition, as explained later on.

The second step in the construction of the model is the definition of the channels used by the principals to communicate with each other. A different global channel is defined for each different message structure used by the protocol. Of course, each protocol uses a finite number of different message structures, which can be easily identified by a simple inspection of the protocol messages. For example, the reduced version of the Needham-Schroeder Public Key Authentication Protocol uses only two message structures, i.e $\{x_1, x_2\}PK(x_3)$ (a pair of elements encoded with a public key) and $\{x_1\}PK(x_2)$ (a single element encoded with a public key). Since we want to enforce the fact that messages are always exchanged between the intruder and one of the other principals, we always specify the identity of the principal involved in the communication with the intruder as the first data exchanged on the channel, while we do not specify the identifier of the other party, because it is always the intruder. The subsequent data exchanged on the channel are the data components of the message

in order of occurrence in the message. So for example, the two global channels for the sample protocol are defined as follows:

```
chan ca = [0] of {mtype, mtype, mtype, mtype};
chan cb = [0] of {mtype, mtype, mtype};
```

Channel `ca` is used to transfer messages of type $\{x_1, x_2\}PK(x_3)$, whereas channel `cb` is used to transfer messages of type $\{x_1\}PK(x_2)$. If, for example, the process representing principal $A$ has to send the message $\{Na, A\}PK(B)$, the following statement has to be used:

```
ca ! A, Na, A, B
```

where the first `A` represents the sender of the message, and the other three items represent the message components (in this case, `B` is considered as a component as well, since it uniquely identifies $PK(B)$). The reception of a message by a principal is expressed similarly. Here it is possible to express any requirements on the received message as well. For example, to express the fact that A must receive a message of type $\{x_1, x_2\}PK(x_3)$, and that it will accept as valid only messages with $x_1 = Na$ and $x_3 = A$, we can use the following statement:

```
ca ? eval(A), eval(Na), x2, eval(A);
```

where $x2$ is a local variable. In general, since all the messages sent by the intruder have the indentity of the receiver as their left most item, when a *normal* host has to receive a message, it must also require that the first data item is its name, which means checking that the message is really addressed to it.

The next step in the model construction is the definition of the processes representing the various roles of the protocol. Such processes must be parameterized with the data that may change from session to session and from instance to instance. Their definition is the Promela representation of the sequence of message exchanges prescribed by the protocol. The definition must include also the recording of particular conditions that are useful in the expression of the security properties to be checked. For our sample protocol, we define proctype `PIni` which describes the behavior of the *initiator* as follows:

```
proctype PIni (mtype self; mtype party; mtype nonce)
{
  mtype g1;

        atomic {
        IniRunning(self,party);
        ca ! self, nonce, self, party;
      }

      atomic {
        ca ? eval (self), eval (nonce), g1, eval (self);
        IniCommit(self,party);
```

```
        cb ! self, g1, party;
      }
}
```

Parameter `self` represents the identity of the host where the initiator process is running, whereas `party` is the identity of the host with which the `self` host wants to run a protocol session. Finally, `nonce` is the nonce that the initiator process will use during the protocol run.

The `atomic` sequences have been used to reduce the amount of allowed interleavings and so to reduce the complexity of the model from the verification point of view.

`IniRunning` and `IniCommit` are two macros used to update the values of the variables recording the atomic predicates that are used to express the authentication properties. In order to explain this part, we have to explain the technique used for property specifications, which is similar to the one presented in [2]. We say that a protocol agent X takes part in a protocol run with agent Y if X has initiated a protocol session with Y. Similarly, we say that a protocol agent X commits to a session with agent Y if X has correctly concluded a protocol session with Y. The fact that a responder with identity B correctly authenticates to an initiator with identity A can be expressed by the following proposition: A commits to a session with B only if B has indeed taken part in a run of the protocol with A. A similar proposition expresses the reciprocal property, i.e. the fact that an initiator with identity A correctly authenticates to a responder with identity B.

Each one of the basic propositions involved in the above properties can be represented in Promela by means of a global boolean variable which becomes true at a particular stage of a protocol run. In the protocol configuration we want to analyze, we have to express that initiator A and responder B correctly authenticate each other, so we need 4 variables, that we define as follows:

```
bit IniRunningAB = 0;
bit IniCommitAB  = 0;
bit ResRunningAB = 0;
bit ResCommitAB  = 0;
```

`IniRunningAB` is true iff initiator A takes part in a session of the protocol with B. `ResRunningAB` is true iff responder B takes part in a session of the protocol with A. `IniCommitAB` is true iff initiator A commits to a session with B. `ResCommitAB` is true iff responder B commits to a session with A.

Authentication of B to A can thus be expressed saying that `ResRunningAB` must become true before `IniCommitAB`, whereas the converse authentication property corresponds to saying that `IniRunningAB` becomes true before `ResCommitAB`. In the LTL formalism, such precedence properties can be expressed as:

- `[] ( ([] !IniCommitAB) || (!IniCommitAB U ResRunningAB) )`
- `[] ( ([] !ResCommitAB) || (!ResCommitAB U IniRunningAB) )`

So, the macros `IniRunning` and `IniCommit` used in the protocol definition update the values of the global variables `IniRunningAB` and `IniCommitAB` and are defined in Promela as follows:

```
#define IniRunning(x,y) if                                  \
                        :: ((x==A)&&(y==B))-> IniRunningAB=1 \
                        :: else skip                        \
                        fi

#define IniCommit(x,y)  if                                  \
                        :: ((x==A)&&(y==B))-> IniCommitAB=1  \
                        :: else skip                        \
                        fi
```

The proctype `PRes` of the *responder* processes is defined according to the same principles as follows:

```
proctype PRes (mtype self; mtype nonce)
{
        mtype g2, g3;

        atomic {
          ca ? eval (self), g2, g3, eval (self);
          ResRunning(g3,self);

          ca ! self, g2, nonce, g3;
        }

        atomic {
          cb ? eval (self), eval (nonce), eval (self);
          ResCommit(g3,self);
        }
}
```

where parameter `self` represents the identity of the host where the responder process is running and `nonce` is the nonce it will use during the protocol run.

`ResRunning` and `ResCommit` are macros used to update the values of the global variables `ResRunningAB` and `ResCommitAB` and are defined as follows:

```
#define ResRunning(x,y) if                                  \
                        :: ((x==A)&&(y==B))-> ResRunningAB=1 \
                        :: else skip                        \
                        fi

#define ResCommit(x,y)  if                                  \
                        :: ((x==A)&&(y==B))-> ResCommitAB=1  \
                        :: else skip                        \
                        fi
```

At this stage it is also possible to define the protocol instance to be modeled. It is simply specified introducing a process instantiation statement in the `init` process for each instance of the initiator and for each instance of the responder. The `init` process must include also the instantiation of a process `PI` representing the intruder activity. `PI` is a process definition without parameters. Its construction is illustrated in the following section.

For our sample instance, we have the following `init` definition:

```
init
{
    atomic {
        if
        :: run PIni (A, I, Na)
        :: run PIni (A, B, Na)
        fi;

        run PRes (B, Nb);

        run PI ();
    }
}
```

The if statement specifies that principal A may initiate a protocol run with any other principal, i.e. either with B or with I. A similar statement is not needed for the responder, because it cannot decide the party with which it must communicate.

### 4.2 The intruder model

The automatic construction of the intruder process definition requires a preliminary static analysis in order to collect all the needed information.

First of all, we need to determine the sets of possible values taken by the free variables occurring in each protocol process. For our sample protocol, such variables are `g1`, `g2` and `g3`. This operation can be performed using a simple data-flow analysis. For example, since initiator processes never check the value of `g1`, but they simply pass it on, such an analysis will yield that in principle `g1` can assume any possible value, i.e. `Na`, `Nb`, `gD`, `A`, `B` and `I`. The same argument is valid for variable `g2`, while, since `g3` is used as an host identifier, the analysis will yield that it can assume only the values `A`, `B` and `I`.

A second static analysis is then needed to restrict the potential intruder knowledge representation to the minimum actually needed. First of all we have to define the intruder's initial knowledge, which, in our sample, is made up of the identities and the public keys of all the principals of the system, i.e. `A`, `B` and `I`, the intruder own private key, and the generic data `gD`. Such a knowledge can increase when the intruder intercepts messages. If we compute all the possible messages the intruder can intercept during the protocol run we can deduce also which are

| Received message | Learned item |
|---|---|
| $\{Na, A\}PK(I)$ | $Na$ |
| $\{Na, A\}PK(B)$ | $\{Na, A\}PK(B)$ |
| $\{Na\}PK(I)$ | $Na$ |
| $\{Nb\}PK(I)$ | $Nb$ |
| $\{gD\}PK(I)$ | - |
| $\{A\}PK(I)$ | - |
| $\{B\}PK(I)$ | - |
| $\{I\}PK(I)$ | - |
| $\{Na\}PK(B)$ | $\{Na\}PK(B)$ |
| $\{Nb\}PK(B)$ | $\{Nb\}PK(B)$ |
| $\{gD\}PK(B)$ | $\{gD\}PK(B)$ |
| $\{A\}PK(B)$ | $\{A\}PK(B)$ |
| $\{B\}PK(B)$ | $\{B\}PK(B)$ |
| $\{I\}PK(B)$ | $\{I\}PK(B)$ |
| $\{Na, Nb\}PK(I)$ | $Na$, $Nb$ |
| $\{Nb, Nb\}PK(I)$ | $Nb$ |
| $\{gD, Nb\}PK(I)$ | $Nb$ |
| $\{Na, Nb\}PK(A)$ | $\{Na, Nb\}PK(A)$ |
| $\{Nb, Nb\}PK(A)$ | $\{Nb, Nb\}PK(A)$ |
| $\{gD, Nb\}PK(A)$ | $\{gD, Nb\}PK(A)$ |
| $\{A, Nb\}PK(I)$ | $Nb$ |
| $\{B, Nb\}PK(I)$ | $Nb$ |
| $\{I, Nb\}PK(I)$ | $Nb$ |
| $\{A, Nb\}PK(A)$ | $\{A, Nb\}PK(A)$ |
| $\{B, Nb\}PK(A)$ | $\{B, Nb\}PK(A)$ |
| $\{I, Nb\}PK(A)$ | $\{I, Nb\}PK(A)$ |

**Table 1.** Knowledge elements that the intruder can eventually acquire

the possible messages the intruder can add to its knowledge. For example, if the intruder intercepts the message $\{Na, A\}PK(I)$ it can learn the nonce $Na$. In order to avoid the representation of redundant knowledge elements, we assume that the intruder always records the learned items in their most elementary forms. For example, if message $\{Na, A\}PK(I)$ is intercepted, we assume that the intruder records only $Na$, and not the whole message $\{Na, A\}PK(I)$, since this message can be built from $Na$ and from $A$, which is always known to the intruder. In other words, the intruder records a complex message in its knowledge only if it cannot decrypt it. For example, if the intruder intercepts the message $\{Na, A\}PK(B)$, it can only remember the message as a whole.

Knowledge items are represented in the intruder process `PI` by means of local boolean variables. For atomic data items (names) such variables have the same name of the data item itself, with the 'k' prefix. For example, variable `kNa` represents the knowledge of $Na$. Similarly, the knowledge of structured data items can be represented by means of bit variables whose names have the 'k'

| Message | Needed knowledge (besides initial knowledge) |
|---|---|
| $\{Na, A\}PK(B)$ | $Na$ or $\{Na, A\}PK(B)$ |
| $\{Na, B\}PK(B)$ | $Na$ or $\{Na, B\}PK(B)$ |
| $\{Na, I\}PK(B)$ | $Na$ or $\{Na, I\}PK(B)$ |
| $\{Nb, A\}PK(B)$ | $Nb$ or $\{Nb, A\}PK(B)$ |
| $\{Nb, B\}PK(B)$ | $Nb$ or $\{Nb, B\}PK(B)$ |
| $\{Nb, I\}PK(B)$ | $Nb$ or $\{Nb, I\}PK(B)$ |
| $\{gD, A\}PK(B)$ | - |
| $\{gD, B\}PK(B)$ | - |
| $\{gD, I\}PK(B)$ | - |
| $\{Na, A\}PK(A)$ | $Na$ or $\{Na, A\}PK(A)$ |
| $\{Na, B\}PK(A)$ | $Na$ or $\{Na, B\}PK(A)$ |
| $\{Na, I\}PK(A)$ | $Na$ or $\{Na, I\}PK(A)$ |
| $\{A, A\}PK(B)$ | - |
| $\{A, B\}PK(B)$ | - |
| $\{A, I\}PK(B)$ | - |
| $\{B, A\}PK(B)$ | - |
| $\{B, B\}PK(B)$ | - |
| $\{B, I\}PK(B)$ | - |
| $\{I, A\}PK(B)$ | - |
| $\{I, B\}PK(B)$ | - |
| $\{I, I\}PK(B)$ | - |
| $\{Nb\}PK(B)$ | $Nb$ or $\{Nb\}PK(B)$ |
| $\{Na, Na\}PK(A)$ | $Na$ or $\{Na, Na\}PK(A)$ |
| $\{Na, Nb\}PK(A)$ | ($Na$ and $Nb$) or $\{Na, Nb\}PK(A)$ |
| $\{Na, gD\}PK(A)$ | $Na$ or $\{Na, gD\}PK(A)$ |

**Table 2.** Knowledge elements potentially needed by the intruder

prefix and include the various data fields in order of occurrence. For example, variable `k_Na_A__B` will represent the knowledge of message $\{Na, A\}PK(B)$.

Since the messages the intruder can intercept are finite, we will also have a finite number of corresponding possible knowledge elements. Table 1 lists all the possible messages that the intruder can intercept in our sample protocol instance and, for each of them, the data items the intruder can learn from it.

A further restriction of the intruder knowledge items to be recorded is possible if we exclude the ones which can never be used by the intruder to generate valid messages. For example, if the intruder knows message $\{Na\}PK(B)$ as a whole, it can potentially send it to $B$, but this one will not accept it, so it is not useful for the intruder to know that message. The set of data items potentially useful to the intruder can be computed listing all the valid messages that the intruder could eventually send to the other principals and determining, for each of them, which data items the intruder could use to build it. The result of this analysis is showed in Table 2 for our sample protocol instance.

The actual set of knowledge elements to be stored in the intruder is the intersection of the two sets just computed (the right columns of Tables 1 and 2):

- Nonces: $Na$ and $Nb$;
- Messages as a whole: $\{Na, Nb\}PK(A)$, $\{Na, A\}PK(B)$ and $\{Nb\}PK(B)$.

Preformed these preliminary analyses, the `PI` proctype definition can be written as follows:

```
proctype PI ()
{
        /* The intruder always knows
           A, B, I, PK(A), PK(B), PK(I), SK(I) and gD
        */

        bit kNa = 0;        /* Intruder knows Na */
        bit kNb = 0;        /* Intruder knows Nb */
        bit k_Na_Nb__A = 0; /*     "       "  {Na, Nb}{PK(A)} */
        bit k_Na_A__B = 0;  /*     "       "  {Na, A}{PK(B)}  */
        bit k_Nb__B = 0;    /*     "       "  {Nb}{PK(B)}     */

        mtype x1 = 0, x2 = 0, x3 = 0;

        do
        ::   ca ! B, gD, A, B
        ::   ca ! B, gD, B, B
        ::   ca ! B, gD, I, B
        ::   ca ! B, A, A, B
        ::   ca ! B, A, B, B
        ::   ca ! B, A, I, B
        ::   ca ! B, B, A, B
        ::   ca ! B, B, B, B
        ::   ca ! B, B, I, B
        ::   ca ! B, I, A, B
        ::   ca ! B, I, B, B
        ::   ca ! B, I, I, B
        ::   ca ! (kNa -> A : R), Na, Na, A
        ::   ca ! (((kNa && kNb) || k_Na_Nb__A) -> A : R),
                  Na, Nb, A
        ::   ca ! (kNa -> A : R), Na, gD, A
        ::   ca ! (kNa -> A : R), Na, A, A
        ::   ca ! (kNa -> A : R), Na, B, A
        ::   ca ! (kNa -> A : R), Na, I, A
        ::   ca ! ((kNa || kNa_A__B) -> B : R), Na, A, B
        ::   ca ! (kNa -> B : R), Na, B, B
        ::   ca ! (kNa -> B : R), Na, I, B
        ::   ca ! (kNb -> B : R), Nb, A, B
```

```
        ::   ca ! (kNb -> B : R), Nb, B, B
        ::   ca ! (kNb -> B : R), Nb, I, B
        ::   cb ! ((k_Nb__B || k_Nb) -> B : R), Nb, B

        :: d_step {
             ca ? _, x1, x2, x3;        if
                                        :: (x3 == I)-> k(x1);
                                                       k(x2)
                                        :: else k3(x1,x2,x3)
                                        fi;
                                        x1 = 0;
                                        x2 = 0;
                                        x3 = 0;
           }

        :: d_step {
             cb ? _, x1, x2;            if
                                        :: (x2 == I)-> k(x1)
                                        :: else k2(x1,x2)
                                        fi;
                                        x1 = 0;
                                        x2 = 0;
            }
        od
}
```

Let us first comment the variable declaration part. We have a bit variable for each knowledge item to be represented in the intruder. These variables are initially set to `0`, because initially the corresponding data items are not known to the intruder. They will be set to `1` as soon as the intruder learns the corresponding item. Besides the variables used to represent the knowledge of the intruder (the ones whose name begins with `k`), three services variables `x1`, `x2` and `x3` are declared. The use of these ones will be explained later on.

Let us now consider the behavior description. The intruder behaves as a never ending process that spends all its time sending messages to and receiving messages from the protocol channels (`ca` and `cb`). Since each operation is atomic, the state of the intruder is determined only by the current contents of its knowledge variables (service variables are always reset to 0 after each use).

Each branch of the main repetition construct represents an input or output operation on the global channels (`ca` and `cb`).

There is one output branch for each possible message sent by the intruder. Some output branches have a pre-condition which enables them, while other output branches are not conditioned by the intruder knowledge, because the intruder always knows how to build the corresponding messages. For example, the branch:

```
:: ca ! B, gD, A, B
```

representing the intruder sending the message $\{gD, A\}PK(B)$ to $B$, is always executable if a process, in this case the process associated to the principal $B$, is ready to synchronize with the intruder process. In contrast, the branch:

```
:: ca ! (kNa-> A : R), Na, Na, A
```

representing the intruder sending the message $\{Na, Na\}PK(A)$ to $A$, requires also that the intruder knows $Na$, i.e. that the value of the local variable `kNa` is `1`. The conditional expression in the first position evaluates to $A$ only if the intruder knows $Na$, and evaluates to a the non-existing identity $R$ otherwise. This correctly means that the intruder can send message $\{Na, Na\}PK(A)$ to $A$ iff it knows $Na$.

For what concerns input branches, there is one of them for each channel. Each input branch includes the input operation, which records the message components in the service variables, and a series of subsequent decoding operations, which depends directly on the message structure. The setting of knowledge variables is technically obtained by the following macros, which automatically ignore useless knowledge elements:

```
#define k(x1)           if                                  \
                        :: (x1 == Na)-> kNa = 1             \
                        :: (x1 == Nb)-> kNb = 1             \
                        :: else skip                        \
                        fi;

#define k2(x1,x2)       if                                  \
                        :: (x1 == Nb && x2 == B)-> k_Nb__B = 1 \
                        :: else skip                        \
                        fi

#define k3(x1,x2,x3)    if                                  \
                        :: (x1 == Na && x2 == A  && x3 == B)   \
                               -> k_Na_A__B = 1             \
                        :: (x1 == Na && x2 == Nb && x3 == A)   \
                               -> k_Na_Nb__A = 1            \
                        :: else skip                        \
                        fi
```

So, for example, if the intruder receives the message $\{Na, A\}PK(I)$ from the channel `ca`, the variable `kNa` is set to `1`, and, if it receives the message $\{Nb\}PK(B)$ from channel `cb`, the variable `k_Nb__B` is set to `1`.

Note that input operations are included in `d_step` sequences and that variables `x1`, `x2` and `x3` are always set to `0` before the end of the `d_step` sequences. This has been done to reduce the amount of possible states of the intruder process.

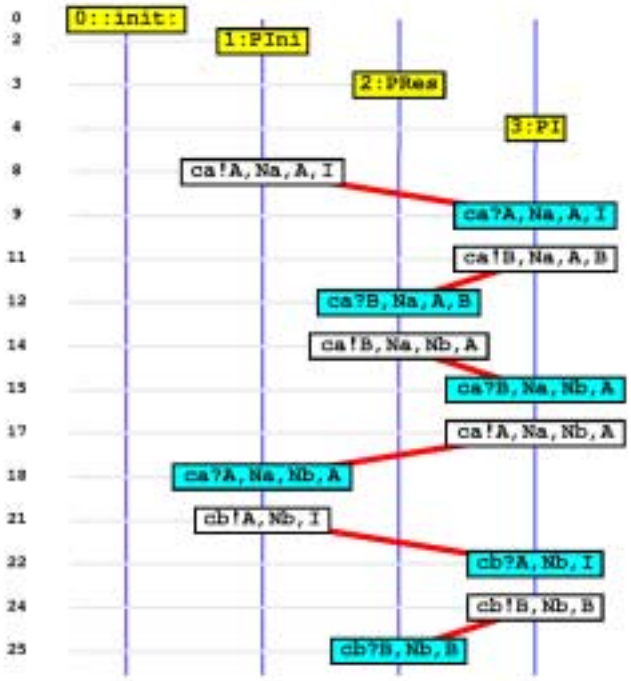


**Fig. 3.** The Lowe's attack on the Needham-Schroeder Public Key Authentication Protocol found by Spin.

## 5 Verification results

Analyzing the previously described model with Spin (we used version 3.4.10), we were able to discover the attack described in [2] by Lowe. The verification of each property took a fraction of second. The first error trail showing the attack is reported in figure 3. As expected, verifying the model of the fixed version of the protocol proposed in [2], we did not find any additional attack on the protocol. The number of reachable states and transitions of the two models (original and fixed protocol) for the verification of the first property (A correctly authenticates to B) are reported in table 3. In both cases, the models are referred to a configuration with one initiator, one responder and one intruder.

To evaluate the efficiency of our modeling approach, we have compared our results with the ones obtained analyzing the same instance of the Needham-Schroeder protocol using the model checker Murhi [3] and the CSP base tools Casper [10] and FDR [11] as described in [12]. All the approaches are able to discover the Lowe's attack but there are some differences in the complexity of the obtained models. The number of reachable states of our model is lower than the Murphi's one, while it is higher than the one obtained using FDR. The lower number of states of the FDR model is probably due to the fact that Casper

| Modeled protocol | States | Transitions |
|---|---|---|
| Original version (property 1) | 381 | 1195 |
| Fixed version (property 1) | 378 | 1275 |

**Table 3.** Verification results

forces strict data typing, and then does not consider possible attacks due to type confusion, which instead are considered by our modeling approach.

## 6 Conclusions

We have presented a way to model cryptographic protocols using Promela. The modeling approach we propose consists of specifying the protocol rules and the configuration to be checked directly in Promela. Instead, the model of the intruder, which is the most difficult part, can be constructed automatically. We described a procedure for the automatic generation of the intruder definition. Such a procedure uses complexity reduction techniques based on a preliminary data-flow analysis to build a simplified model. Following this approach we succeeded in finding the well-known attack on the Needham-Shroeder Public Key Authentication Protocol.

In this paper we have informally illustrated the procedure to construct the intruder model, using a case study. Future work includes a formal definition of the intruder construction procedure and its implementation in an automatic intruder model generator. Another possible future development is the construction of a user-friendly specification interface which makes it possible to describe the protocol rules and configuration more directly and generate the whole Promela model automatically.

## References


1. Clarke, E.M., Jha, S., Marrero, W.: Verifying security protocols with Brutus. ACM Transactions on Software Engineering and Methodology **9** (2000) 443–487
2. Lowe, G.: Breaking and fixing the Needham-Shroeder public-key protocol using FDR. In: Proceeding of TACAS96, LNCS 1055, Springer-Verlag (1996) 147–166
3. Mitchell, J.C., Mitchell, M., Stern, U.: Automated analysis of cryptographic protocols using murphi. In: Proceedings of the 1997 Conference on Security and Privacy (S&P-97), Los Alamitos, IEEE Press (1997) 141–153
4. Josang, A.: Security protocol verification using SPIN. SPIN'95 Workshop (1995)
5. Needham, R.M., Schroeder, M.D.: Using encryption for authentication in large networks of computers. Communications of the ACM **21** (1978) 993–999
6. Diffie, W., Hellman, M.: New directions in cryptography. IEEE Transactions on Information Theory **IT-22** (1976) 644–654
7. Rivest, R.L., Shamir, A., Adleman, L.M.: A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM **21** (1978) 120–126

8. Denning, D.E., Sacco, G.M.: Timestamps in key distribution protocols. Communications of the ACM **24** (1981) 533–536
9. Durante, L., Sisto, R., Valenzano, A.: A state exploration technique for spi-calculus testing equivalence verification. In: Proceedings of FORTE/PSTV 2000, Pisa, Italy, Kluver (2000) 155–170
10. Lowe, G.: Casper: A compiler for the analysis of security protocols. In: PCSFW: Proceedings of The 10th Computer Security Foundations Workshop, IEEE Computer Society Press (1997)
11. Ltd., F.S.E.: Failures-Divergence Refinement. FDR2 User Manual. Available at http://www.formal.demon.co.uk/fdr2manual/index.html (3 May 2000)
12. Lowe, G.: Casper: A compiler for the analysis of security protocols - user manual and tutorial. Available at http://www.mcs.le.ac.uk/~glowe/Security/Casper/manual.ps (1999)