SRI International

CSL Internal Note • October 28, 2003

# The Needham-Schroeder Protocol in SAL

John Rushby
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

This research was supported by NSA under contract MDA904-02-C-1196.

# Abstract

The Needham-Schroeder authentication protocol is specified in SAL and its model checker is used to detect the flaw discovered by Gavin Lowe. The SAL simulator is used to further explore the model of the protocol. This provides a simple illustration in the use of SAL for this domain.

# Contents

# 1 Introduction

The Needham-Schroeder authentication protocol, first published in 1978 [NS78], initiated a large body of work on the design and analysis of cryptographic protocols. In 1995, Gavin Lowe published an attack on the protocol that had apparently been undiscovered for the previous 17 years [Low95]. The following year, Lowe showed how the flaw could be discovered mechanically by model checking [Low96], and this has been followed by many papers on model checking and automated verification of similar protocols. In this note, I describe how SAL can be used for this purpose. My intent is simply to provide a simple illustration of the use of SAL in this domain.

I assume readers have some familiarity with the concepts and notation for cryptographic protocols and their analysis. The Needham-Schroeder protocol is specified as follows.

Message 1. $A \to B$: $A.B.\{A, N_A\}_{PK(B)}$
Message 2. $B \to A$: $B.A.\{N_A, N_B\}_{PK(A)}$
Message 3. $A \to B$: $A.B.\{N_B\}_{PK(B)}$.

The intent of the protocol is to establish mutual authentication between the principals $A$ and $B$ in the presence of an intruder who can intercept, delay, read, copy, and generate messages, but who does not know the secret keys of the principals.

Principal $A$ begins by sending to $B$ a message containing the its own identity and that of $B$ in the clear, and an encrypted component containing a "nonce" (a previously unused and unpredictable identifier) $N_A$ and its own identity. The encryption uses $B$'s public key and can be decoded by $B$ using its secret key, but is indecipherable to all other participants.

$B$ responds with a similar message to $A$, including the nonce received from $A$ and a new nonce of its own in the encrypted portion, which is readable only by $A$.

$A$ examines the second message and concludes that it really is from $B$ (since only $B$ could have discovered the nonce $N_A$) and that it is not a replay (because the nonce $N_A$ is current). It then returns the nonce $N_B$ to $B$ under $B$'s public key. On receipt of this message, $B$ uses similar reasoning to determine that it really is from $A$, and is current, and so both principals have authenticated the other.

The flaw discovered by Lowe uses an interleaving of two runs of the protocol, as shown below. $A$ initiates a run with principal $I$, who is corrupt (i.e., an intruder). $I$ then initiates a run with $B$, purporting to be $A$ and using the nonce provided by $A$. $B$ replies with a message encrypted for $A$ that $I$ uses unchanged in a message of its own to $A$. $A$ decrypts this to discover $B$'s nonce (thinking it is $I$'s) and sends it to $I$ under $I$'s public key. $I$ now knows $B$'s nonce and can complete its run with $B$. At this point, $B$ believes it has authenticated $A$, but it is actually talking to $I$.

Message 1a. $A \rightarrow I$: $A.I.\{A, N_A\}_{PK(I)}$
Message 1b. $I_A \rightarrow B$: $A.B.\{A, N_A\}_{PK(B)}$
Message 2b. $B \rightarrow I_A$: $B.A.\{N_A, N_B\}_{PK(A)}$
Message 2a. $I \rightarrow A$: $I.A.\{N_A, N_B\}_{PK(A)}$
Message 3a. $A \rightarrow I$: $A.I.\{N_B\}_{PK(I)}$
Message 3b. $I_A \rightarrow B$: $A.B.\{N_B\}_{PK(B)}$.

Here, $I_A$ indicates $I$ masquerading as $A$, and the suffices a, and b on the message numbers indicate which run of the protocol they belong to.

The protocol is easily fixed by including the identity of the responder in the encrypted portion of the second message (this prevents the replay of the encrypted portion of 2b in 2a).

Message $2'$. $B \rightarrow A$: $B.A.\{B, N_A, N_B\}_{PK(A)}$

Our task now is to specify the protocol in SAL and to use its model checkers to discover the bug and provide some assurance for the correction.

## 2 Specification in SAL

We need to model the behavior of the principals and the intruder. We also need some model of the network over which messages are transmitted. The intruder is generally assumed to have full control of the network, so the network must provide suitable capabilities for the intruder.

We will model the network as a buffer that can hold at most one message and provides four services: `write`, `overwrite`, `read` and `copy`. The `write` service is applicable only when the buffer is empty; it stores the incoming message in the buffer. The other services are applicable only when the buffer is full; `overwrite` replaces the buffer contents with the incoming message, `read` empties the buffer and returns its previous contents, while `copy` returns the contents and leaves the buffer unchanged.

We want to make this network generic with respect to the type of messages, so we enclose the `network` module in a SAL context (also called `network`) that takes the `msg` type as a parameter (see Figure 1). The parameters to SAL contexts comprise a list of types and a list of constants separated by a semicolon and enclosed in braces; here the list of constants is empty, but we still need the semicolon.

Within this context, we define `bufferstate` and `action` as enumerated types, then specify the `network` module. This module has inputs `act` (indicating the action to be performed), and `inms` (indicating the message to be inserted in the network—which is relevant only when `act` is either `write` or `overwrite`). The module has two output variables: `nstate` records whether or not the buffer is full or empty, while `buffer` records the contents of the buffer.

```
network{msg: TYPE;}: CONTEXT =
BEGIN
  bufferstate: TYPE = {empty, full};
  action: TYPE = {read, write, overwrite, copy};

network: MODULE =
BEGIN
  INPUT  act: action, inms: msg
  OUTPUT nstate: bufferstate, buffer: msg
INITIALIZATION
  nstate = empty;
TRANSITION
[
 act' = write AND nstate = empty -->
  buffer' = inms';
  nstate' = full;
[]
 act' = overwrite AND nstate = full -->
  buffer' = inms';
[]
 act' = read AND nstate = full -->
  nstate' = empty;
[]
 act' = copy AND nstate = full -->
  nstate' = nstate;
[]
 ELSE -->
]
END;

END
```

Figure 1: The Network Context

The behavior of the network is specified by guarded commands; the `ELSE` case applies when an `action` is invoked in an inappropriate state (e.g., `act = read` when `nstate = empty`). The bodies of the guarded commands specify the new (primed) values of local and output variables; variables not explicitly mentioned are left unchanged. Observe that it is the new (primed) value of `act` that appears in the guards: this is so that the network responds immediately to the actions requested by its clients and makes it much easier to interpret counterexamples. We place the `network` context in a file called `network.sal` and check it for syntax and wellformedness by the command

```
sal-wfc network
```

to which the response `Ok.` indicates that all is well.

We can now specify the behavior of the principals and intruder within a context `needhamschroeder`. We begin by specifying an enumerated type `ids` that will be used to identify the participants. This type should have sufficient elements to name as many principles and intruders as we wish to examine, plus an extra element `X` that is used as an "error" element in certain constructions. For our purposes, two principals and one intruder are sufficient.

```
needhamschroeder: CONTEXT =
BEGIN
  ids: TYPE = {a, b, e, X};

  participants: TYPE = {x: ids | x /= X};
  intruder(x: participants): BOOLEAN = x=e;

  intruders: TYPE = {x: participants | intruder(x)};
  principals: TYPE = {x: participants | NOT intruder(x)};
```

We then identify `participants` as the subtype of `ids` that are not the "error" value, and provide a predicate `intruder` that indicates the `ids` corresponding to the intruder(s); here only `e` is an intruder. We then specify `intruders` as the subtype of `participants` that are intruders and `principals` as the dual subtype that are not intruders.

Each principal must be able to generate as many nonces as the number of protocol runs that it participates in. For our purposes, we need consider only single runs, so each principal needs only a single nonce. An implementation of the protocol must generate nonces in a way that makes it infeasible for the intruder to guess them, but in modeling we can simply not endow the intruder with guessing ability and then can make the nonces deterministic. We do this by reusing `ids` as nonces: each participant's nonce is its own id.

```
  nonces: TYPE = ids;
  nonce(a: participants): nonces = a;
```

Messages contain an encrypted component; when decrypted, this component is a tuple, which we call a `dmsg` (for "decrypted message"), containing an id and two nonces. The constant `arb` is a `dmsg` whose elements are all `X`; it is used to represent an "uninitialized" `dmsg` (otherwise the model checker may "guess" a magical value). When encrypted, a `dmsg` yields an `emsg` (for "encrypted message"), which we specify as a datatype with constructor `enc` (for "encrypt") and accessors `key` (the id of the participant whose public key is used) and `payload` (the `dmsg` that is encrypted). Messages that are sent to the network must contain the id of the sender and intended recipient as well as the actual `emsg` to be communicated; we represent these kinds of messages by the record type `msg`. Notice that we have used a tuple, a datatype, and a record to represent the similar types `dmsg`, `emsg`, and `msg`, respectively; this is done purely to illustrate use of these three constructors.

```
dmsg: TYPE = [ids, nonces, nonces];
arb: dmsg = (X,X,X);

emsg: TYPE = DATATYPE
  enc(key: ids, payload: dmsg)
END;

msg: TYPE = [# from: participants, to: participants, em: emsg #];
```

Now that we have the type `msg` of messages that are used, we can instantiate the context `network` appropriately and give the instantiation the abbreviated name `net`.

```
net: CONTEXT = network{msg;};
```

A participant can decrypt an `emsg` only if the key matches its own id, otherwise it gets the `arb` value. We specify this in the function `dec`.

```
dec(k: participants, m:emsg): dmsg =
 IF key(m)=k THEN payload(m) ELSE arb ENDIF;
```

During the protocol, principals progress through some of five possible states, which we identify in the enumerated type `states`.

```
states: TYPE = {sleeping, waiting, tentative, engaged, responding};
```

The behavior of the `i`'th `principal` is specified in a module called `principal`, parameterized by `i`. This module takes as inputs the state of the network, and a message `imsg` from the network (which may be invalid if the state is `empty`). Observe how an exclamation point is used to extract the type `bufferstate` from the network instance `net`. The outputs of the module are the action it performs on the network and the message the action is applied to (relevant only for `write` and `overwrite`). However, in SAL, output variables may be written by only a single module, whereas we will have a module for each principal and each of these will be writing to these variables, so they must be declared as global rather than output. The module has two local state variables: `pc` (the “program counter”) records where it is in the protocol, and `responder` records the principal which it is authenticating.

```
principal[i: principals]: MODULE =
BEGIN
  INPUT nstate: net!bufferstate, imsg: msg
  GLOBAL act: net!action, omsg: msg
  LOCAL pc: states, responder: participants
INITIALIZATION
  pc = sleeping;
  responder = i;
```

Initially, the principal is `sleeping`, and its `responder` is set to `itself`.

The protocol is specified as a series of guarded commands in the transition section of the module. The first set of rules applies when a principal is `sleeping` and decides to initiate a run of the protocol with another participant `j` different to itself. Notice the “quantification” over `j`, and notice that it is over `participants`, not just `principals`. This is because we must allow a principal to initiate the protocol with an intruder. The command can be used only when the `net` is `empty`, in which case the principal constructs the first message of the protocol and writes it to the network. Observe that `(i, nonce(i), X)` constructs the tuple consisting of `i`’s id, its nonce, and the “empty” nonce `X`. The construction `enc(j, (i, nonce(i), X))` then encrypts this `dmsg` with `j`’s key and `(# from := i, to := j, em := enc(j, (i, nonce(i), X)) #)` then constructs the `msg` to be written to the network: the `(#...#)` construction is a record constructor containing “assignments” to its fields. The principal records `j` as its `responder`, and enters the `waiting` state.

```
TRANSITION
[
([] (j: participants): i /= j AND
  pc = sleeping AND nstate = net!empty -->
    pc’ = waiting;
    responder’ = j;
    omsg’ = (# from := i, to := j, em := enc(j, (i, nonce(i), X)) #);
    act’ = net!write;
)
```

The next set of commands describe the action of a principal that receives a message produced as described above. The command applies only if the principal is `sleeping`, the `net` is `full`, and contains a message from a principal `j` different to `i` that is addressed to `i` and encrypted with its key. In this case, it extracts the nonce from the message, constructs the second message of the protocol, and `overwrites` the message on the network. The principal records `i` as its `responder`, and enters the `tentative` state.

1

```
[]
([] (j: participants): i /= j AND
     pc = sleeping AND nstate = net!full
     AND imsg.from = j AND imsg.to = i AND dec(i, imsg.em).1=j  -->
   responder' = j;
   pc' = tentative;
   act' = net!overwrite;
   omsg' = (# from := i, to := j,
              em := enc(j, (X, dec(i,imsg.em).2, nonce(i))) #);
)
```

A principal in the `waiting` state that finds the network `full`, containing a message from its `responder` addressed to itself, encrypted under its own key (if it were not, the value of `dec(i,imsg.em).2` would be `X`), and bearing its own nonce, constructs the third message of the protocol, and enters the `engaged` state.

2

```
[]
  pc = waiting AND nstate = net!full
     AND imsg.from = responder AND imsg.to = i
     AND dec(i,imsg.em).2 = nonce(i) -->
   pc' = engaged;
   act' = net!overwrite;
   omsg' = (# from := i, to := responder,
              em := enc(responder, (X, dec(i,imsg.em).3, X)) #);
```

A principal in the `tentative` state that finds the network `full`, containing a message from its `responder` addressed to itself, encrypted under its own key, and bearing its own nonce, enters the `responding` state.

```
[]
  pc = tentative AND nstate = net!full
     AND imsg.from = responder AND imsg.to = i
     AND dec(i,imsg.em).3 = nonce(i) -->
   pc' = responding;
   act' = net!read;
```

In all other cases, the principal does nothing.

```
[]
 ELSE -->
]
END;
```

Next, we specify the behavior of the intruder(s); we allow more than one so the module `intruder` is parameterized by the id `x` of the intruder concerned. The global and input variables of an intruder are the same as those of a principal. Intruders operate by remembering messages that they have seen (but may not be able to decrypt), together with any nonces they have recovered from messages that they can decrypt. We need to allocate local variables for however many messages and nonces we allow an intruder to remember. For our purposes, one of each is sufficient: `nmem` is the memory for a nonce, and `mmem` the memory for an `emsg`; we also need `n1` and `n2` as temporaries for holding nonces. We initialize the memories to harmless values.

```
intruder[x:intruders]: MODULE =
BEGIN
  GLOBAL act: net!action, omsg: msg
  INPUT nstate: net!bufferstate, imsg: msg
  LOCAL  nmem, n1, n2: nonces, mmem: emsg
INITIALIZATION
  nmem = nonce(e);
  mmem = enc(X,(X,X,X));
```

The first thing an intruder can do is read messages addressed to itself. This guarded command applies when the network is `full`, the message is addressed to `x`, and uses its key. The intruder extracts the nonce in the second position of the decrypted message: this is the value `dec(x,imsg.em).2` (which will be `X` if the message did not use x's key). It then nondeterministically chooses either to store this value in `nmem`, or to leave that value unchanged: the `IN` construction specifies this nondeterministic choice. The intruder similarly makes a nondeterministic choice whether to `read` or `copy` the message.

```
TRANSITION
[
 nstate = net!full AND imsg.to = x  -->
   nmem' IN {dec(x,imsg.em).2, nmem};
   act' IN {net!read, net!copy};
```

When a message is not addressed to the intruder, it cannot decrypt it, but it can choose to save the whole encrypted portion of the message.

```
[]
 nstate = net!full AND imsg.to /= x -->
   mmem' IN {imsg.em, mmem};
   act' IN {net!read, net!copy};
```

The intruder can nondeterministically choose to send the encrypted message it has remembered to a principal `j`, while masquerading as `i`.

```
[]
([] (i: participants, j: principals): TRUE -->
   act' = IF nstate = net!empty THEN net!write ELSE net!overwrite ENDIF;
   omsg' = (# from := i, to := j, em := mmem #);
)
```

Similarly, the intruder can nondeterministically construct a message containing its own nonce and the one it has remembered.

```
[]
([] (i: participants, j: principals): TRUE -->
   act' = IF nstate = net!empty THEN net!write ELSE net!overwrite ENDIF;
   n1' IN {nmem, nonce(x)};
   n2' IN {nmem, nonce(x)};
   omsg' = (# from := i, to := j, em := enc(j, (i, n1', n2')) #);
)
```

In all other circumstances, it does nothing.

```
[]
  ELSE -->
]
END;
```

We now construct a complete system by asynchronously composing some collection of principals and intruders, and synchronously composing that compound with the network. We rename the `buffer` and `inms` of the network so that they connect up to the `imsg` and `omsg` of the principals and intruder.

```
 system: MODULE = (([] (id: principals): principal[id]) [] intruder[e])
   || (RENAME buffer TO imsg, inms TO omsg IN net!network);
```

Here, our principals are `a` and `b`, and there is but a single intruder `e`.

# 3 Analysis in SAL

The property we wish to examine is correct authentication: whenever a principal `x` reaches the `responding` state with a principal `y`, then it must be that `y` initiated the protocol with `x`—that is, `y` must be in the `waiting` or `engaged` states and have `x` as its responder. We specify this as the property `prop`.

```
 prop: THEOREM system |- G((FORALL (x,y: principals):
    (pc[x]=responding AND responder[x]=y) =>
         ((pc[y]=waiting OR pc[y]=engaged) AND responder[y]=x)));

END
```

We place the `needhamschroeder` context in a file of the same name and invoke the SAL symbolic model checker as follows.

```
sal-smc needhamschroeder prop
```

Within a few seconds, this reports that `prop` is invalid and produces a counterexample comprising 10 steps. The counterexample trace is rather verbose, so I provide an abbreviated version with commentary on each step in Figure 2. A similar counterexample can be found using the bounded model checker.

```
sal-bmc needhamschroeder prop
```

The counterexamples found by SAL are equivalent the one described in the introduction.

The weakness in the protocol that is exploited in the counterexamples is the lack of any indication of the sender in the encrypted part of message 2. This can be corrected by changing the final assignment in $\boxed{1}$ on page 7 to the following (the `X` is changed to `i`).

```
 omsg' = (# from := i, to := j,
            em := enc(j, (i, dec(i,imsg.em).2, nonce(i))) #);
```

The guard in $\boxed{2}$ must then be changed (by addition of the third line below) to check that the message really does come from the expected responder.

```
pc = waiting AND nstate = net!full
   AND imsg.from = responder AND imsg.to = i
   AND dec(i,imsg.em).1 = responder
   AND dec(i,imsg.em).2 = nonce(i) -->
```

With these changes, the SAL symbolic model checker proves `prop`. Of course, "prove" has a rather limited meaning in model checking: we have proved that the property is not violated by any run of the protocol within the limits established by this specific model. To prove that it is sound in general, we need to use more classical theorem proving within an environment such as PVS, where unlimited numbers of runs and participants can be considered: see, for example [MR00]. The state of the art in fully automated cryptographic protocol analysis uses constraint satisfaction procedures to solve the problem for arbitrary, but bounded, numbers of runs [MS01].

## 3.1 Exploration in SAL

In addition to finding bugs and proving (limited) correctness, model checking also can be used to explore behaviors in either a random or directed manner. This can be very useful as a way to gain an understanding of the possibilities contained

```
Step 0: Initialization
-------------------------------------------------------------------------
Step 1: a sends message 1 to e

pc[a] = waiting;  pc[b] = sleeping;
responder[a] = e;
omsg.from = a;  omsg.to = e;  omsg.em = enc(e, (a, a, X));
-------------------------------------------------------------------------
Step 2: e remembers a’s nonce

nmem = a;
-------------------------------------------------------------------------
Step 3: e (masquerading as a) send message 1 to b

omsg.from = a;  omsg.to = b;  omsg.em = enc(b, (a, a, a));
-------------------------------------------------------------------------
Step 4: b sends message 2 to a, but it is intercepted by e

pc[a] = waiting;  pc[b] = tentative;
responder[b] = a;
omsg.from = b;  omsg.to = a;  omsg.em = enc(a, (X, a, b));
-------------------------------------------------------------------------
Step 5: e remembers encrypted part of b’s message

mmem = enc(a, (X, a, b));
-------------------------------------------------------------------------
Step 6: e sends message 2 (using remembered part) to a

omsg.from = e;  omsg.to = a;  omsg.em = enc(a, (X, a, b));
-------------------------------------------------------------------------
Step 7: a sends message 3 to e

pc[a] = engaged;  pc[b] = tentative;
omsg.from = a;  omsg.to = e;  omsg.em = enc(e, (X, b, X));
-------------------------------------------------------------------------
Step 8: e remembers b’s nonce from a’s message 3

nmem = b;
-------------------------------------------------------------------------
Step 9: e (masquerading as a) sends message 3 to b (with remembered nonce)

omsg.from = a;  omsg.to = b;  omsg.em = enc(b, (a, e, b));
-------------------------------------------------------------------------
Step 10: b falsely believes it has authenticated a

pc[b] = responding;
```

Figure 2: Counterexample Trace Found by SAL Model Checker

within a specification. The crudest form of exploration is random simulation. This can be accomplished in SAL with `sal-pathfinder`, which uses the bounded model checker to construct an example path of a given length. For example, the following command constructs a path of length 20.

```
sal-path-finder -v 1 -d 20 needhamschroeder system
```

A rather more directed approach uses the model checkers to find a path to some state of interest. Here, we characterize the states of interest by some predicate `p`, then model check `G(NOT p)`. For example, suppose we are interested in the states where the intruder has spoofed both principals and brought them to the `tentative` state. The property here is

```
  pc[a]=tentative AND pc[b]=tentative
```

so we check the following property.

```
test: THEOREM system |-  G( NOT (pc[a]=tentative AND pc[b]=tentative));
```

We can use either the symbolic model checker

```
sal-smc -v 1 needhamschroeder test
```

or the bounded model checker

```
sal-bmc -v 1 -d 20 -it needhamschroeder test
```

to construct the path to the state of interest. The argument `-d 20` tells the bounded model checker to search to a maximum path length of 20, while the argument `-it` tells it to do so by iterative deepening (i.e., starting at 1 and incrementing by 1 each time) so that it is sure to find the shortest path.

The most controlled method of exploration uses the SAL Simulator. This builds on the machinery of the symbolic model checker to allow the statespace to be explored under interactive guidance. In the example below, we start `sal-sim`, then instruct it to import the `needhamschroeder` context, and start the simulation (the `#unspecified` string is an artifact of the simulation environment).

```
> sal-sim
SAL Simulator (Version 2.0). Copyright (c) 2003 SRI International.
Build date: Fri Oct 24 17:07:50 PDT 2003
Type ‘(exit)’ to exit.
Type ‘(help)’ for help.
sal > (import! "needhamschroeder")
#unspecified
sal > (start-simulation! "system")
#unspecified
```

At each point, the state of the simulation is a set of states. The command `(step!)` picks one of these arbitrarily, and then computes *all* its successors, which then become the new current state of the simulation. The command `(display-curr-states)` prints the current states, up to some maximum number (ten by default). The tenth state of this particular simulation (following the step) is shown below.

```
State 10
--- System Variables (assignments) ---
(= act write);
(= imsg.em (enc e (mk-tuple b b X)));
(= imsg.from b);
(= imsg.to e);
(= mmem (enc X (mk-tuple X X X)));
(= n1 b);
(= n2 e);
(= nmem e);
(= nstate full);
(= omsg.em (enc e (mk-tuple b b X)));
(= omsg.from b);
(= omsg.to e);
(= pc[a] sleeping);
(= pc[b] waiting);
(= responder[a] a);
(= responder[b] e);
```

Notice that it uses the `lsal` syntax (this is an alternative, Lisp-like syntax for SAL). The command `(display-curr-trace)` picks one of the current states arbitrarily, then prints the path to reach it.

If we are interested in exploring the trace found earlier by model-checking the `test` property, we can see from that trace that we need to select a state where the intruder sends a message with the `em` field equal to `enc(b, (a, e, e))`. We can prune the current set of states to just those having this value by the following command.

```
(filter-curr-states! "(= omsg.em (enc b (mk-tuple a e e)))")
```

It is easy to construct the lsal expressions needed by looking at those that are printed by the `(display-curr-states)` command. We can now take another step, prune again, and iterate this process to reach a state in which both principals are in the `tentative` state.

```
(step!)
(filter-curr-states! "(= pc[b] tentative)")
(step!)
(filter-curr-states! "(= omsg.em (enc a (mk-tuple e e e)))")
(step!)
(filter-curr-states! "(= pc[a] tentative)")
```

We have now reached the situation of interest to us and can explore forwards and backwards using the capabilities of the simulator. Because the simulator is built on the machinery of a model checker, these capabilities go far beyond those of an ordinary simulator. In particular, the SAL Simulator can search from the current set of states forward to find states satisfying a given property, or report that the property is unreachable from the current set of states. For example, we may be interested to check that the intruder can complete its spoofing of the participants and bring either of them to the `responding` state.

```
(run! "(or (= pc[a] responding) (= pc[b] responding))")
```

Here, the simulator reports `#f`, meaning that it could not find a path to a state satisfying this property. Perplexed, we restart the system and search from the beginning for a state in which principal `a` is responding to the intruder.

```
(start-simulation! "system")
(run! "(and (= pc[a] responding) (= responder[a] e))")
```

Again, we learn there is no such path. So now we look for a state in which the intruder has brought `a` to the `tentative` state (which we already know is possible).

```
(run! "(and (= pc[a] tentative) (= responder[a] e))")
```

We can display the current states or trace to reinforce our understanding at this point, but we know that the intruder should be able to deliver the message that takes `a` to the `responding` state provided it knows `a`'s nonce (which is also `a`). We look to see if the intruder knows this nonce in any of the current states.

```
(filter-curr-states! "(= nmem a)")
(display-curr-states)
```

The simulator responds `#t`, which means the current set of states is empty. So now we backtrack, return to our previous state (the first `run!` command below), then search forward to a state in which the intruder knows the nonce (the second `run!` command).

```
(backtrack!)
(run! "(and (= pc[a] tentative) (= responder[a] e))")
(run! "(= nmem a)")
```

Again, we find there is no path to this property once `a` is in the `tentative` state and responding to `e`. So now we search for *any* path to a state in which the intruder knows `a`'s nonce.

```
(backtrack!)
(run! "(= nmem a)")
```

The simulator finds suitable paths, but we see that they all lead to states in which `a` has initiated the protocol (i.e., it is in the `waiting` state). We have now collected enough information to reexamine our model of the intruder: it seems that it can discover nonces from the first message of a protocol run, but not the second. The difference between these two cases is that the "other" nonce is in the second position of the `emsg` tuple in the former case, and the third in the latter. Sure enough, the command in the relevant step of the intruder is the following.

```
    nmem' IN {dec(x,imsg.em).2, nmem};
```

It should, of course, be changed as follows.

```
    nmem' IN {dec(x,imsg.em).2, dec(x,imsg.em).3, nmem};
```

After making this change, we exit the simulator and restart it, and once again check the property of interest.

```
(import! "needhamschroeder")
(start-simulation! "system")
(run! "(and (= pc[a] responding) (= responder[a] e))")
```

This time, the simulator is able to find a suitable path.

The SAL Simulator is implemented as a set of scripts in the Scheme language. The scripts that interpret most of the high-level commands are just two or three lines long: all the real work is done by calling the functions on the BDD interface used by the symbolic model checker. It is easy for users to write their own Scheme scripts to extend the functionality of the simulator interface.

## 4 Conclusion

This example has introduced some aspects of the SAL language not used in the other tutorial examples (for example, parameterized contexts and multiple files). It should also provide a start to those who wish to use SAL to analyze cryptographic protocols. More complex protocols may require a more complex intruder model (for example one that remembers multiple messages and nonces) and may require more participants and more interleaved runs, but the basic approach should extend quite easily. You can download the SAL files developed in this note from `http://www.csl.sri.com/users/rushby/abstracts/needham03`.

## References

[Low95] Gavin Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131–133, February 1995. 1

[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 '96*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166, Passau, Germany, March 1996. Springer-Verlag. 1

[MR00] Jonathan Millen and Harald Rueß. Protocol-independent secrecy. In Michael Reiter and Roger Needham, editors, *Proceedings of the Symposium on Security and Privacy*, pages 110–119, Oakland, CA, May 2000. IEEE Computer Society. 10

[MS01] Jonathan Millen and Vitaly Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis. In *8th ACM Conference on Computer and Communications Security (CCS '01)*, pages 166–175, Philadelphia, PA, November 2001. Association for Computing Machinery. 10

[NS78] Roger Needham and Michael Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, December 1978. 1