CSL Internal Note • October 28, 2003

The Needham-Schroeder Public-Key Protocol in Higher-Order Logic

Tom Melham
University of Cambridge Computer Laboratory
William Gates Building, JJ Thomson Avenue
Cambridge CB3 0FD, UK

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

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

network: MODULE =
BEGIN
  INPUT  act: action, inms: msg
```

the network and the message the action is applied to (relevant only for

that is addressed to `i` and encrypted with its key. In this case, it extracts the nonce f7om the message, constructs the second meF471age of the p7otocol, and

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 concernedTfA421(Te)-1the

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 stepn. 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
```

```
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, p0(p9999999999999999999999mm8010:-----
Step 1: a sends msg 1 to e
omsg.from = a;  omsg.to = e;  omsg.em = enc(e, (a, a, p0(p9999999999999999999999mm8016--remembe
responder[a] = e;omsgbfrom = a;  omsg.to = e;  omsg.em = enc(e, (a, a, p0(p999999999999999999
```

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

References