Naval Postgraduate School; 9, 10 Feb 2004

Protocol Exploration

With Modern Model Checkers

(I'll be using SAL)

John Rushby

Computer Science Laboratory

SRI International

Menlo Park CA USA

Introduction

- It is well-known that bugs in cryptographic protocols can often be found using a conventional model checker
  - Build a model of the protocol
  - Build a model of the intruder
  - Specify some numbers of participants, intruders, runs
  - Specify (safety) properties of interest
  - Explore all reachable states (similar for liveness)
  - Relies on raw power to do this
- Cf. specialized crypto protocol analyzers
  - Where some of this is built in
  - Used to optimize the search

What's New?

- Modern model checkers offer a variety of notations
  - Hardware description languages (SMV, BLIF)
  - Programming languages (Java, C)
  - Civilized modeling languages (SAL)
  - Intermediate languages (SAL again)
- They offer several different tools
  - Symbolic, bounded model checkers (nuSMV)
  - Explicit-state, symbolic, bounded, infinite-bounded, witness model checkers (SAL)
- They are modular and scriptable
  - Bogor
  - SAL

So What?

- The modeling can be very direct at higher levels of abstraction
  - And more realistic (e.g., programs) at lower levels
- Easy to model powerful intruders
- Can choose the right tool for the analysis concerned
  - Have the power to examine the vast number of states generated by powerful intruder models
- Can model and analyze other parts/properties than security
- Can do exploration as well as model checking

Example: Needham Schroeder

What did you expect?

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. $A \to B$: $A.B.\{N_B\}_{PK(B)}$.

I'm going to model and analyze this in SAL

Getting Started: the Network

- Want a “network” that is generic wrt. message type
- Acts like a one-place buffer
- Messages can be written (if empty),
- And read, copied, overwritten (if full)

Network

```
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
  ...
```

Network (ctd.)

```
[
 act' = write AND nstate = empty -->
  nstate' = full;  buffer' = inms';
[]
 act' = overwrite AND nstate = full -->
  buffer' = inms';
[]
 act' = read AND nstate = full -->
  nstate' = empty;
[]
 act' = copy AND nstate = full -->
  nstate' = nstate;
[]
 ELSE -->
]
```

The Participants

- Need at least two principals
- And an intruder
- These are subtypes of participants
- Useful to have an extra “error” id for initialization etc.

Participants

```
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)};
```

Nonces

- In practice, need to make sure these are fresh
- In modeling, they can be deterministic
  - Do not endow the intruder with guessing ability

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

Messages

- Messages contain an encrypted component
- When decrypted it's a triple dmsg
- arb is used for the initial value

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

Otherwise the model checker might use a "magic" value

- An encrypted message records the key used

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

- An encrypted message on the network indicates its source and destination

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

- Tuple, datatype, record, just for variety

Decryption

Can successfully decrypt an encrypted message only if you are the participant whose key was used

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

Otherwise, get arb

State of the Principals

- Initially sleeping
- May decide to initiate a dialog and go to waiting
- And then to engaged if the protocol completes

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

- If another initiates the dialog, go to tentative
- And then to responding if the protocol completes
- In either case, responder is the identity of the other

Principals

Initially, each principal is sleeping, and its responder is set to itself

```
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;
```

Principals (ctd. 1)

Waking up and initiating a dialog with j

```
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;
)
```

Principals (ctd. 2)

Waking up and responding to a dialog initiated by j

```
[]
([] (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)))#);
)
```

Principals (ctd. 3)

Initiator accepts the response from j

```
[]
  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))#);
```

Principals (ctd. 4)

Responder accepts second message from initiator j

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

Otherwise do nothing

Intruders

Need to provide the intruder with memory for messages it has seen but not been able to decrypt, and for the contents of messages (i.e., nonces) that it has decrypted

```
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));
```

We provide memory for one of each: nmem and mmem; n1 and n2 are temporaries

Intruders (ctd. 1)

Intruder can read and decrypt messages sent to itself

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

Nondeterministically replaces saved nonce with the new one, and removes the message or copies it

Intruders (ctd. 2)

Can save whole messages not addressed to itself

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

Intruders (ctd. 3)

Can send remembered message to 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 #);
)
```

Intruders (ctd. 4)

And can manufacture messages containing its own nonce or a remembered one

```
[]
([] (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’))#);
)
[]
  ELSE -->
]
END;
```

And that’s all it can do

The Complete System

- Asynchronously compose some collection of principals and intruders
- And synchronously compose that compound with the network
- We'll have two principals a and b, and single intruder e

  No explicit limit on interleaved runs

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

Authentication Property

- The property we wish to examine is correct authentication
- Whenever a principal x reaches the responding state with a principal y, 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)));
```

Model Checking

- Symbolic model checking
  - sal-smc -v 3 needhamschroeder prop

  Builds a transition relation on 150 state bits, 339,917,146 reachable states, and reports a counterexample ten steps long in about 20 secs

- Witness model checking
  - sal-wmc -v 3 needhamschroeder prop

  Also reports the counterexample, in about 40 secs

- Bounded model checking
  - sal-bmc -v 3 -d 10 needhamschroeder prop

  Builds a SAT problem with 40,756 nodes and reports the counterexample in about 40 secs

The Counterexample

```
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;
```

The Counterexample (ctd.)

Is essentially the classic one

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

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

Repairing The Protocol

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

Repairing The Protocol (ctd.)

- In SAL, we need to change the final assignment on slide 17 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 on slide 18 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) -->
```

Model Checking Again

- Symbolic model checking
  - sal-smc -v 3 needhamschroeder prop

  This time there are 339,954,654 reachable states, and the property is “verified” in 25 seconds

  - Verification is relative to the intruder model and dimensions used

- Witness model checking
  - sal-wmc -v 3 needhamschroeder prop

  Verifies the property in 30 seconds, and (internally) constructs a witness

- Bounded model checking
  - sal-bmc -v 3 -d 10 needhamschroeder prop

  Finds no counterexamples to depth 10 in 85 seconds

Exploration

- Finding bugs and verifying are extreme examples of exploration
- In general, want to see runs that take us to interesting states or through interesting scenarios as a way of increasing our understanding and confidence in the operation of the system
- Can do this in a simulator, but have to think of all the inputs and interactions ourselves
- Supposing we had a simulator built on a model checker
- Then we could tell the model checker to find a path to an interesting state, then take over and explore in detail, and so on
- The SAL simulator does this

Exploration with the SAL Simulator

- Suppose we are interested in the scenario where the intruder spoofs both principals into thinking they are responding to the other
- We start the simulator and tell it to take us to a state where both principals are in tentative state

```
tulip:sal> sal-sim
 SAL Simulator (Version 2.2). Copyright (c) 2003, 2004 SRI
sal > (import! "needhamschroeder")
sal > (start-simulation! "system")
sal > (run! "(and (= pc[a] tentative)(= pc[b] tentative))")
#t
```

- The `#t` means it succeeded
- Notice we're using LSAL syntax here

Exploration with the SAL Simulator (ctd. 1)

- Now we would like to see the intruder continue and bring both principals to the responding state

```
sal > (run! "(and (= pc[a] responding) (= pc[b] responding))")
#f
```

- The #f means it failed
- Perplexed, we see if it can bring either to completion

```
sal > (run! "(or (= pc[a] responding) (= pc[b] responding))")
#f
```

- Hmm! Let's restart and find a path where a is responding to the intruder

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

Exploration with the SAL Simulator (ctd. 1)

- So let's get to a state where a is in the tentative state (which we already know is possible)

```
sal > (run! "(and (= pc[a] tentative) (= responder[a] e))")
#t
```

- Now we know that the intruder should be able to construct the message to take a to the responding state provided it knows a's nonce (which is also a)

- So let's see if the intruder does know this nonce in the current state

```
sal > (filter-curr-states! "(= nmem a)")
sal > (display-curr-states)
#t
```

Evidently not—the `#t` means the filtered set is empty (we also could have just looked at the current state)

Exploration with the SAL Simulator (ctd. 2)

- So now let's get back to where we were

```
sal > (backtrack!)
sal > (run! "(and (= pc[a] tentative) (= responder[a] e))")
#t
```

- And look for a state where the intruder knows the nonce

```
sal > (run! "(= nmem a)")
#f
```

- Hmm! Let's go back to the beginning and look for any state where it knows this nonce

```
sal > (start-simulation! "system")
sal > (run! "(= nmem a)")
#t
sal > (display-curr-states)
```

Exploration with the SAL Simulator (ctd. 3)

- It turns out the simulator has found a run where a initiated the dialog
- It seems that the intruder can learn a's nonce when a is the initiator, but not when it is the responder
- The difference between these cases is that a's 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}
```

Exploration with the SAL Simulator (ctd. 3)

- After making this change, we exit the simulator and restart it, and again check the properties of interest

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

This time, the simulator is able to find suitable paths

- We also check that the authentication property is still true using sal-smc

Summary

- Modern model checkers are attractive and effective tools for exploring protocols
- Can add GUIs and also raise modeling level
- For example, Guido Wimmel at TUM models e-commerce protocols in the Focus graphical environment
    - Messages are specified as secret, authentic, etc.
    - A suitable intruder model is synthesized
    - Properties are selected
    - And the whole lot sent to a model checker

Summary (ctd.)

- A simulation environment built on a model checker provides a powerful environment for exploring protocols
- The sal-sim environment is scriptable in Scheme
- In fact, the SAL model checkers are just Scheme scripts on the underlying API
- Can develop novel capabilities as scripts on this API
  - Example: test case generation
- See `fm.csl.sri.com` for our tools
- `http://www.csl.sri.com/users/rushby/abstracts/needham03` for this example

Putting More Things Together

- It is now fairly routine to have model checkers as backends to theorem provers (e.g., PVS), or proof assistants as front ends to model checkers (e.g., Cadence SMV)
- But we envisage a larger collection of symbolic computational procedures
  - Decision procs, ITPs, abstractors, inv generators, model checkers, static analyzers, test generators, ITPs, rewriters
- Interacting through a scriptable tool bus
- The bus manages symbolic and concrete artifacts
  - Test cases, abstractions, theorems, invariants

  Over which it performs evidence management
- Focus shifts from verification to symbolic analysis
  - Iterative application of analysis to artifacts to yield new artifacts, insight and evidence

Integrated, Iterated Analysis