Swansea University
Department of Computer Science

U Berger
November 1, 2022, 2022

CSC385/CSCM85 Modelling and Verification Techniques

Lab 3

Modelling and analysing security protocols

The tasks below may be done in groups of up to four students. To obtain full marks, all tasks must be completed. For fewer or incompletely accomplished tasks, partial marks will be awarded. Marks will be awarded only for work that is demonstrated during the lab session. Write your answers in the file `verification-lab3-Oct22-template.csp`.

This lab assignment is worth 50 marks out of 100 lab marks. It needs to be completed and assessed in one of the lab sessions on the 1st, 4th, 15th, or 18th of November.

Introduction In this lab assignment we model and analyse the Needham-Schroeder authentication protocol, NSP. The analysis will consist of checking secrecy, that is, whether a user can get hold of a nonce they are not supposed to know.

We first assume that all users are honest and check that in that case secrecy holds.

Then we add an intruder that composes bogus messages from the nonces he learnt to confuse the honest users.

Finally, we model Lowe's amedment of the protocol and check its safety.

Preliminaries

Users and Nonces. We define the datatypes of users and nonces:

```
datatype User = A | B | I

datatype Nonce = N.User.User
```

We use variables u, v for users and n, m for nonces.

N.u.v is the nonce created by u and intended for v. For example, N.A.B is a nonce created by Alice and intended for Bob.

It is important to note that only the user who created a nonce knows that he or she created it and who it is intended for. All other users cannot read-off this information from the nonce. In general, users can store and copy nonces, include them in a message, and test whether two nonces are equal.

Messages. A message consists of:

* Information about which step the message is for (1 or 2 or 3),
* a plain text consisting of one or two nonces and zero or one user,
* the public key of some user used to encrypt the plain text.

Therefore, the general format of a message is

$$i.ns.us.v$$

where

$i \in \{1, 2, 3\}$,

ns is a sequence of nonces of length one or two,

us is a sequence of users of length zero or one,

v is a user.

We can define the set of all messages in CSP_M as follows:

```
Message =
  { i.ns.us.v | i <- {1,2,3},
                ns <- union({<n>|n<-Nonce}, {<n,m>|n<-Nonce, m<-Nonce}) ,
                us <- union({<>}, {<u>|u<-User}) ,
                v <- User }
```

We use *msg* as a variable for messages.

Encoding the steps of the NSP **as messages.** Messages are used to encode the steps of the NSP as follows:

A (Alice) is replaced by the variable u

B (Bob) is replaced by the variable v

N_A is replaced by the variable n

N_B is replaced by the variable m

We use variables, since the messages should work for arbitrary users, not only for Alice and Bob.

(1) $A \to B : \{N_A, A\}_{pK_B}$	is encoded as	$1.\langle n\rangle.\langle u\rangle.v$
(2) $B \to A : \{N_A, N_B, \}_{pK_A}$	is encoded as	$2.\langle n, m\rangle.\langle\rangle.u$
(3) $A \to B : \{N_B\}_{pK_B}$	is encoded as	$3.\langle m\rangle.\langle\rangle.v$

Access functions. As useful access functions we define a function `nonces` that computes the sequence of nonces contained in a message, and a function `pk` that computes the user whose public key was used to encrypt the plain text of a message:

```
nonces(_.ns._._) = ns

pk(_._._.v) = v
```

Reducing redundancy. The data sets defined above contain some data that will never be used, for example, N.A.A, or 2.$\langle$N.A.$B\rangle$.$\langle\rangle$. We define smaller sets that do not contain such useless elements. This will make our later analysis of the NSP more efficient.

All relevant nonces:

```
RelNonce = { N.u.v | u <- User, v <- diff(User,{u}) }
```

All relevant messages:

```
RelMessage =
  Union( { { 1.<n>.<u>.v | n <- RelNonce, u <- User, v <- User },
           { 2.<n,m>.<>.u | n <- RelNonce, m <- RelNonce, u <- User },
           { 3.<m>.<>.v | m <- RelNonce, v <- User } } )
```

Channels and the Environment. We introduce channels for sending and receiving messages:

```
channel send, receive : Message
```

The users (who will be defined in Task 1) will communicate through an environment which transfers messages from the `send` channel to the `receive` channel and then repeats:

```
ENV = send ? msg : RelMessage -> receive . msg -> ENV
```

Task 1: Modelling honest users.

A user of the NSP may either be active (initiate a communication), or passive (respond to the request for communication).

Task 1.1: Complete the following code fragments of processes:

UA(u) ("User Active") models a user u who plays the role of the initiator,

UP(v) ("User Passive") models a user v who plays the role of the responder,

```
UA(u) = [] v : diff(User,{u}) @
        let n = N.u.v within
        send.1.<n>.<u>.v     ->
        [] m : diff(RelNonce,{n}) @
        receive.2.<n,m>.<>.u ->
        ...                  ->
        STOP

UP(v) = [] u : diff(User,{v}), n : RelNonce @
        ...                  ->
        let m = N.v.u within
        ...                  ->
        ...                  ->
        STOP
```

Task 1.2: Define a process U(u) of a user who has the choice to play an active or passive part in the NSP

```
U(u) = ...
```

In the definitions of UA(u) and UP(v) you may replace STOP by U(u) thus allowing a user to repeatedly engage in the protocol.

Task 1.3: Define all users running in parallel without direct communication

```
USERS = ...
```

10 marks

Task 2: Modelling and analysing the NSP with honest users only

Task 2.1: Model the system of all users interacting with the environment via the channels `send` and `receive`. This is our model of the NSP.

```
System = ...
```

Secrecy. Can a user get hold of a nonce they are not supposed to know?

For example, Bob (B) doesn't learn a forbidden secret from any of the messages

$1.\langle \mathrm{N}.A.I\rangle.\langle A\rangle.I,$

$2.\langle \mathrm{N}.A.B, \mathrm{N}.B.A\rangle.\langle\rangle.B$

In the first case, because Bob cannot read the plain text, in the second case, because the Bob is allowed to know all the nonce contained in that message.

On the other hand, the message

$1.\langle \mathrm{N}.A.I\rangle.\langle A\rangle.B$

reveals to Bob the nonce $\mathrm{N}.A.I$ which he is not supposed to know. Therefore, if Bob received such a message, it would constitute a breach of secrecy.

To answer the question whether a breach of secrecy may occur, we first define the set of nonces a user is allowed to know:

```
noncesFrom(u) = { N.u.v | v <- diff(User,{u}) }
noncesFor(u) = { N.v.u | v <- diff(User,{u}) }

noncesAllowed(u) = union(noncesFrom(u), noncesFor(u))
```

Next, we define a boolean function that tests whether all nonces in a list of nonces are allowed to be known by a user:

```
allAllowed(ns,u) =
  if ns == <>
  then true
  else member(head(ns),noncesAllowed(u)) and allAllowed(tail(ns),u)
```

We call a message *innocent* if the only user who can see the nonces in the message is allowed to see them.

```
Innocent(msg) = allAllowed(nonces(msg),pk(msg))
```

Given a set S of users, the function SECRET(S) below computes the set of all messages *msg* such that no user in S can learn a secret nonce from *msg*. In other words, either no user in S can decrypt *msg*, or *msg* is innocent.

```
SECRET(S) =
  { msg | msg <- RelMessage, not(member(pk(msg),S)) or Innocent(msg) }
```

If in a trace of `System` all actions of the form receive.*msg* are such that *msg* $\in$ SECRET(S), we know that no user in S has learnt a nonce they should not know. In other words, no user in S can breach secrecy.

Task 2.2: Define a specification process SECRECY(S) which can perform exactly the actions of the form `receive.msg`, where `msg` is a message in SECRET(S).

```
SECRECY(S) = ...
```

Task 2.3: Check that `System` is safe. That is, when running `System`, no user can get hold of a nonce they are not supposed to know.

```
assert ...
```

Our analysis so far aimed at showing that nothing bad can happen. But do the things we want to happen actually happen?

Task 2.4 (Optional): Think of ways to see that the intended runs of the NSP are modelled by the process `System`, that is, occur as traces of System.

Task 2.5 (Optional): Check whether System is deadlock free. Analyse the result and think of ways to prevent deadlock.

10 marks

Task 3: Modelling the intruder

User I has so far been honest. Now we change the behaviour of I to act as an intruder, that is, a user who tries to break the protocol. The intruder repeatedly intercepts any message that has been sent, tries to extract nonces from it, composes from these nonces new messages and sends them to confuse the other users.

We define several functions that compute the information the intruder can extract or synthesise from a message.

The sequence of nonces the intruder can read off from a message:

```
learnI(msg) = if pk(msg) == I then nonces(msg) else <>
```

The set of nonces the intruder can learn from a set of messages, or generate by himself:

```
genNoncesI(Msg) = union(Union({set(learnI(msg)) | msg <- Msg}), noncesFrom(I))
```

Generating all unsuspected messages from a given set Ns of nonces

```
allMessages(Ns) =
 Union(
  {
   {1.<n>.<u>.v | v <- User, n <- Ns, u <- diff(User,{v}) },
   {2.<n,m>.<>.u | u <- User,
                   n <- { N.u.v | v <- diff(User,{u}) },
                   member(n,Ns),
                   m <- diff(Ns,{n}) },
   {3.<n>.<>.v | v <- User,
                 n <- { N.v.u | u <- diff(User,{v}) },
                 member(n,Ns)}
  }
      )
```

The set of messages the intruder can generate from the messages in a set of messages `Msg`, or generate by himself:

```
genMessagesI(Msg) = union(Msg,allMessages(genNoncesI(Msg)))
```

More precisely: `genMessagesI(Msg)` is the set of messages the intruder can generate from the messages in `Msg`, either by replicating messages in `Msg` or generating new messages from nonces that are either extracted from messages in Msg or generated by the intruder.

Here is the task: Model the intruder by completing the following template:

```
Intruder = [] msg : RelMessage @
           receive.msg ->
           [] msg' : ... @
           ... ->
           Intruder
```

10 marks

Task 4: Modelling and analysing the NSP with intruder

We let user I behave as an intruder:

```
UI(u) = if u != I then U(u) else Intruder
```

Task 4.1: Let the users and the intruder run in parallel, without direct communication:

```
USERSI = ...
```

Task 4.2: Define the analogue to `System`, but now with the intruder:

```
SystemI = ...
```

Task 4.3: Check whether the NSP is safe, that is, whether the can learn a secret nonce.

```
assert ...
```

Check that the attack described by Lowe has been found.

10 marks

Task 5: Modelling and analysing the Needham-Schroeder-Lowe protocol (NSLP) with intruder

Change the code above so that it models the NSLP, Lowe's amendment of the NSP. Analyse the NSLP. Try to increase the number of users. You may either make a copy of the file and then make changes or directly edit the file, keeping the old lines as comments. Alternatively, you may create new processes with new names (add "L" at the end).

10 marks