Lecture XX

Some Promela examples

Wolf-Sheep-Cabbage problem

```
#define wolf 0
#define sheep 1
#define cabbage 2
#define none 7

chan S1toS2         = [1] of {byte};
chan S2toS1         = [1] of {byte};
```


Wolf-Sheep-Cabbage problem

```
active proctype shore1()
{
    bool S1[3] = 1;
    byte cargo;

    if
    :: S1toS2!none
    :: S1toS2!wolf ; S1[wolf] =0
    :: S1toS2!sheep ; S1[sheep] =0
    :: S1toS2!cabbage ; S1[cabbage] =0
       fi;
end1:   ((!S1[sheep]) || (!S1[wolf] && !S1[cabbage]));
    S2toS1?cargo;
    if
    :: cargo == none
    :: else -> S1[cargo] = 1
    fi;
    if
    :: S1toS2!none
    :: S1[wolf] -> S1toS2!wolf ; S1[wolf] =0
    :: S1[sheep] -> S1toS2!sheep ; S1[sheep] =0
    :: S1[cabbage] -> S1toS2!cabbage ; S1[cabbage] =0
       fi;
    goto end1
}
```

```
active proctype shore2()
{
    bool S2[3] = 0;
    byte cargo;

end2:   ((!S2[sheep]) || (!S2[wolf] &&
    !S2[cabbage]));
    S1toS2?cargo;
    if
    :: cargo == none -> skip
    :: else -> S2[cargo] = 1
    fi;
    assert(!(S2[sheep] && S2[wolf] &&
    S2[cabbage]));
    if
    :: S2toS1!none
    :: S2[wolf] -> S2toS1!wolf ;
    S2[wolf] =0
    :: S2[sheep] -> S2toS1!sheep ;
    S2[sheep] =0
    :: S2[cabbage] -> S2toS1!cabbage ;
    S2[cabbage] =0
        fi;
    goto end2
}
```


Cryptographic protocol modeling

Verification of Security protocols

- One aspect of security analysis is to check that protocols for authentication, transmission, etc. work correctly, assuming that the underlying primitives for encryption, etc. work as intended.

Setup

- A protocol manipulates an unbounded number of strings which can be nonces, encryption keys, etc.
- These strings can not be guessed by an intruder.
- Encrypting message m with key k is denoted k[m]
- In public key cryptography, public key k has inverse k^{-1}
- k^{-1} [k[m]] = m
- Messages can have several components, e.g., [k[<m1,k' [m2]>]]

Dolev-Yao model

Model of how an intruder can operate

- Intruder can intercept any message, and generate any message, using the information it has obtained

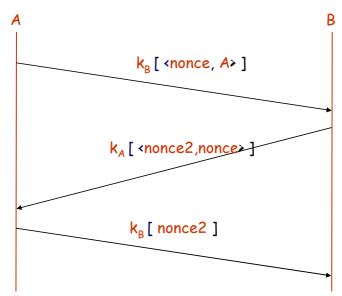
Rules for generating messages (knowledge) from what it knows

- If m1 and m2 are known, then <m1,m2> is known
- If <m1,m2> is known, then m1 and m2 are known
- If m and k are known, then k[m] is known
- If k[m] and k^{-1} are known, then m is known

We would like to represent the knowledge of the intruder by a set of messages, from which all others that it can infer, can be inferred

This is a set of generators

Needham-Schroeder Protocol



Needham-Schroeder Protocol

```
proctype PIni (byte self; byte party; byte nonce)
{
  byte nonce1;
    msg1 ! nonce, self, party;
    msg2 ? eval(nonce), nonce1, eval(self);
    msg3 ! nonce1 , party
}

proctype PRes (byte self; byte nonce)
{
  byte nonce2 , party1;
    msg1 ? nonce2 , party1 , eval(self);
    msg2 ! nonce2, nonce, party1;
    msg3 ? eval(nonce) , eval(self)
}
```


Needham-Schroeder Protocol

```
proctype PIni (byte self; byte party; byte nonce)
{
  byte nonce1;
    ch3 ! nonce, self, party;
    ch3 ? eval(nonce), nonce1, eval(self);
    ch2 ! nonce1 , party ;
    assert (party != B || (! knows1[nonce] && ! knows1[nonce1] ))
}

proctype PRes (byte self; byte nonce)
{
  byte nonce2 , party1;
    ch3 ? nonce2 , party1 , eval(self);
    ch3 ! nonce2, nonce, party1;
    ch2 ? eval(nonce) , eval(self);
    assert (party1 != A || (! knows1[nonce] && ! knows1[nonce2] ))
}
```


Intruder Model

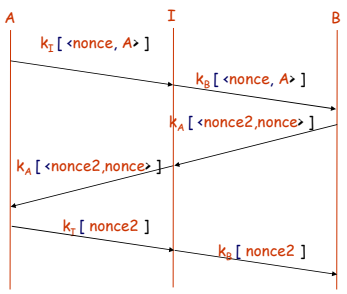
```
proctype PI ()
{
  short i1,i2,i3;
do
:: ch3? i1,i2,i3 ->
    if
    :: i3 == I -> knows1[i1] = true;
                  knows1[i2] = true
    :: i3 != I -> knows3[i1,i2,i3)] =
    true
    fi;
:: ch2? i1,i2 ->
    if
    :: i2 == I ->  knows1[i1] = true
    :: i2 != I -> knows2[i1,i2] = true
    fi
:: do :: i1 < MAX ->
            if :: i1++
               :: knows1[i1] -> break
            fi
      :: i1 == MAX -> break
    od;
(* same for i2, i3 *)
if
:: (i1<MAX && i2<MAX && i3<2) ->
                if
                :: ch2 ! i1 , i3
                :: ch3 ! i1 , i2, i3
                :: skip
                fi
 :: skip
 fi;
 do :: i1 < MAX*2 ->
            if
            :: i1++
            :: knows2[i1] -> break
            fi
      :: i1 == MAX*2 -> break
   od;
   if
   :: (i1 < MAX*2) ->
        if
        :: ch2 ! first2(i1),
second2(i1)
        :: skip
        fi
    :: else -> skip
} ......
```


Configuration

```
init {
        atomic{

 knows1 [A] = true;
 knows1 [B] = true;
 knows1 [I] = true;
                if
                :: run PIni (A,I,Na)
                :: run PIni (A,B,Na)
                fi

                run PRes (B,Nb);
                run PI ()
        }
}
```


Counter-Example