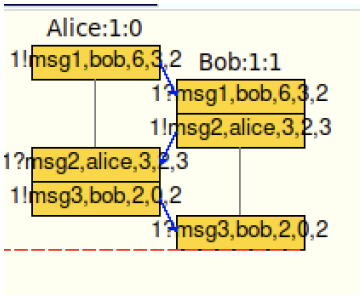
CS4710
Chanpech Hoeng
Tony Garnett
Oct 29, 2023

Phase 1: Needham-Schroeder Protocol

Observation of the simulated model:



On any variation of the seeds, the simulation always yields the above graphical processes interaction where two processes named Alice and Bob start at the same time. The task is for Alice to encrypt her message and her nonce using Bob's public keys and send them to Bob. Bob's process would wait to receive this message and decrypt it using his private key and extract the content of the message. Bob will then use the content to send an acknowledgment back to Alice, the content contains Alice's encrypted nonce (nonceA) along with his nonce (nonceB). Alice upon receiving the acknowledgment, will confirm her nonce are identical to Bob's' and send a final confirmation message back to Bob. After this Alice's process will terminate and Bob will follow after it receives the confirmation message.

LTL property for Alice and Bob's successful termination:

Liveness:

```
//ltl liveness {<>(term == 2)};
```

Eventually, term (process termination) will reach 2 meaning all processes (excluding intruder) will terminate. This is implemented through the incrementation of the term variable, where Alice and Bob process will increment this just before it successfully finishes its instructions.

Safety:

```
ltl safety {[]<>(aliceSendMsg == 1 && bobReceiveMsg == 1 &&
                 aliceReceiveAck == 1 && bobReceiveConfirmation ==1) }
```

For all of the cases, *userA* (Alice) will eventually receive an acknowledgment from *userB* (Bob). Variables aliceSendMsg and aliceReceive ack are implemented in Alice processes. The *aliceSendMsg* will incremented just after Alice sends her message to Bob and the *aliceReceiveAck* will be incremented just after Alice receives the acknowledgment message from

Bob. Similarly, *bobReceiveMsg* and *bobReceiveConfirmation* variables are incremented in the same manner inside of Bob's process with the respective send and receive.

Verification and counterexample:

Passed verification for both of the above LTL properties with no counter-example.

```
unreached in proctype Alice
        NS-Protocol.pml:52, state 19, "printf('Invalid Key for user Alice\n')"
        NS-Protocol.pml:59, state 26, "printf('Invalid   for user Alice\n')"
        (2 of 43 states)
unreached in proctype Bob
        NS-Protocol.pml:94, state 6, "printf('Invalid key for user Bob\n')"
        NS-Protocol.pml:112, state 23, "printf('Invalid key for user Bob\n')"
        NS-Protocol.pml:117, state 30, "printf('Invalid Nonce for user Bob\n')"
        (3 of 37 states)
unreached in claim safety
        _spin_nvr.tmp:10, state 13, "-end-"
        (1 of 13 states)

pan: elapsed time 0 seconds
No errors found -- did you verify all claims?
```

Does weak fairness impact the result of verification?

Verification error: upon enabling weak fairness constraints.

```
No errors found -- did you verify all claims?
spin -a  NS-Protocol.pml
ltl safety: [] (<> (((((aliceSendMsg==1)) && ((bobReceiveMsg==1))) && ((aliceReceiveAck==1))) &&
((bobReceiveConfirmation==1))))
gcc -DMEMLIM=1024 -O2 -DXUSAFE -w -o pan pan.c
./pan -m10000  -a -f -N safety
Pid: 12239
warning: only one claim defined, -N ignored
error: p.o. reduction not compatible with fairness (-f) in models
       with rendezvous operations: recompile with -DNOREDUCE

pan: elapsed time 1.72e+07 seconds
To replay the error-trail, goto Simulate/Replay and select "Run"
```

Trace of

```
nts)
spin: NS-Protocol.pml:0, warning, global, 'int   term' variable is never used (other than in print stmnts)
starting claim 3
spin: couldn't find claim 3 (ignored)
using statement merging
  2: proc 2, no matching stmnt 123
#processes: 2
  2:        proc  1 (Bob:1) NS-Protocol.pml:89 (state 1)
  2:        proc  0 (Alice:1) NS-Protocol.pml:26 (state 5)
2 processes created
Exit-Status 0
```

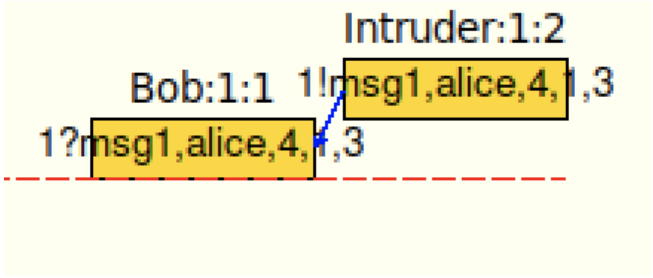
Enabling weak fairness constraints yields the above error upon verification. The error trace does not make sense as it seems like the processes were created and exited right away.

Phase 2: Intruder

Observation of the simulated model behavior:

Variation of the intercepts and their graphical process interaction:

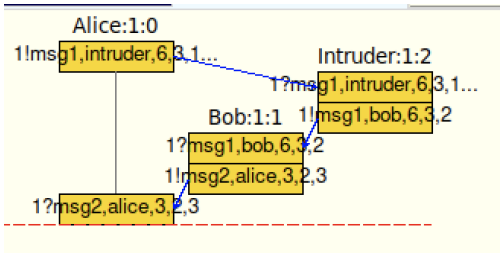
The intruder acts as Alice (verification counter-example):



The intruder intercepts at the very beginning and sends the message to Bob acting as Alice and forwarding Alice's content to Bob. This variation is the counter-example that was shown after verification where Alice's process was stuck in a cycle. This will be discussed further in detail in the LTL section.

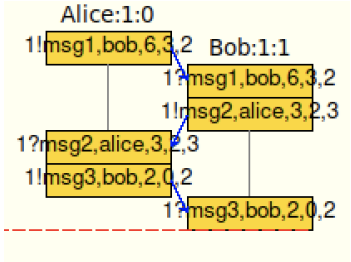
```
23:     proc  0 (Alice:1) NS-Protocol.pml:38 (state 9) [sender.key = 2]
<<<<<START OF CYCLE>>>>>
24:     proc  - (safety:1) _spin_nvr.tmp:8 (state 8) [(!((((aliceSendMsg==1)&&(bobReceiveMsg==1))&&(aliceReceiveAck==
1))&&(bobReceiveConfirmation==1)))]
```

The intruder intercepts Alice's initial message (seed 233)



The intruder intercepts Alice send message and just forwards it as normal to Bob. Bob would then decrypt the message and send back an acknowledgment to Alice. Alice's process upon receiving the message will verify its nonce contents and see that it was not the same thus exited without sending back a confirmation message to Bob. It logically makes sense that since Alice's original nonce content was the intruder but the one it got back was Bob and thus exited.

The intruder didn't intercept, so operates as expected (seed 54)



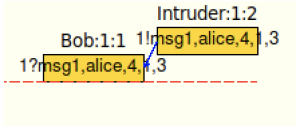
The intruder didn't intercept so the communication behavior was as expected (shown in phase 1)

The same LTL Properties:

```
//ltl liveness {<>(term == 2)};
```

```
ltl safety {[]<>(aliceSendMsg == 1 && bobReceiveMsg == 1 &&
                 aliceReceiveAck == 1 && bobReceiveConfirmation ==1) }
```

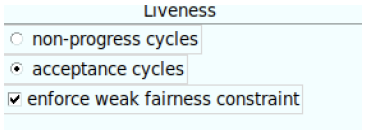
Verification produced the following counter-example:



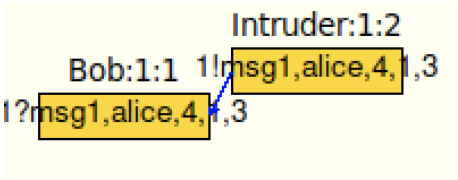

```
25:     proc  0 (Alice:1) NS-Protocol.pml:38 (state 10) [sender.key = 2]
<<<<<START OF CYCLE>>>>>
26:     proc  - (safety:1) _spin_nvr.tmp:8 (state 8) [(!(((((aliceSendMsg==1)&&(bobRecei
```

The counter-example states that Alice's process will eventually end up getting stuck in a cycle of trying to send its message to Bob. This cycle logically makes sense as the intruder as shown on the graph intercepts and grabs a hold of the Alice send signal in the communication channel. The intruder then arbitrarily picked out message types. For this trace, it picked msg1 and acted as Alice, and forwarded it to userB (Bob). At this point, Bob will try to decrypt the message and send back an acknowledgment to the sender. However, by this point, one of the LTL properties from the liveness called *aliceSendMsg* will never be reached (thus will never be 1).

Does fairness impact the result of the verification?



Result in:



Fairness didn't change the counter-example nor the trace of error. Alice's process still ends up going into cycles. This makes sense in this man-in-the-middle attack scenario as the intruder process is non-deterministically intercepting the message. So it can always be the case that if the intruder intercepts a message between Alice and Bob and one of the process never receive back a confirmation message then it will eventually get stuck in cycles of waiting.