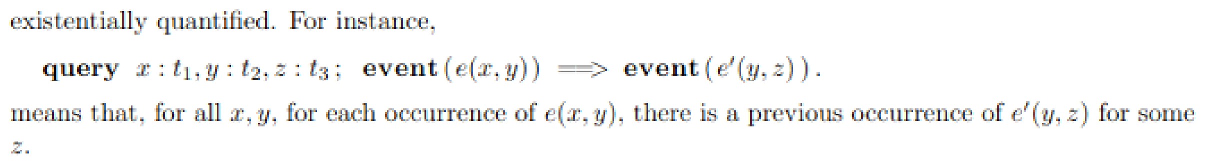
Introduction to ProVerif

Name
Casper Kristiansson, casperkr
Jonas Liley, liley
Nicole Wijkman, nwijkman

Date
2024-03-01

Q1: What can we conclude from that these two queries evaluate to true? Consult the manual to understand queries of this form if necessary
In Proverif we use queries to check the reachability of events. In this case, if we query event(evReachI) we can check if the event can be reached.

Q2: Explain how this query can still evaluate to true even though the Idummy-process does nothing. Use the ProVerif manual if needed. *HINT: consider at which point the evICommplete event is emitted in traces from this updated model.*
The query evaluates whether the right expression is always executed before the left expression. Because the left expression will never be reached (Process I is never started) the right expression will always run beforehand.


existentially quantified. For instance,

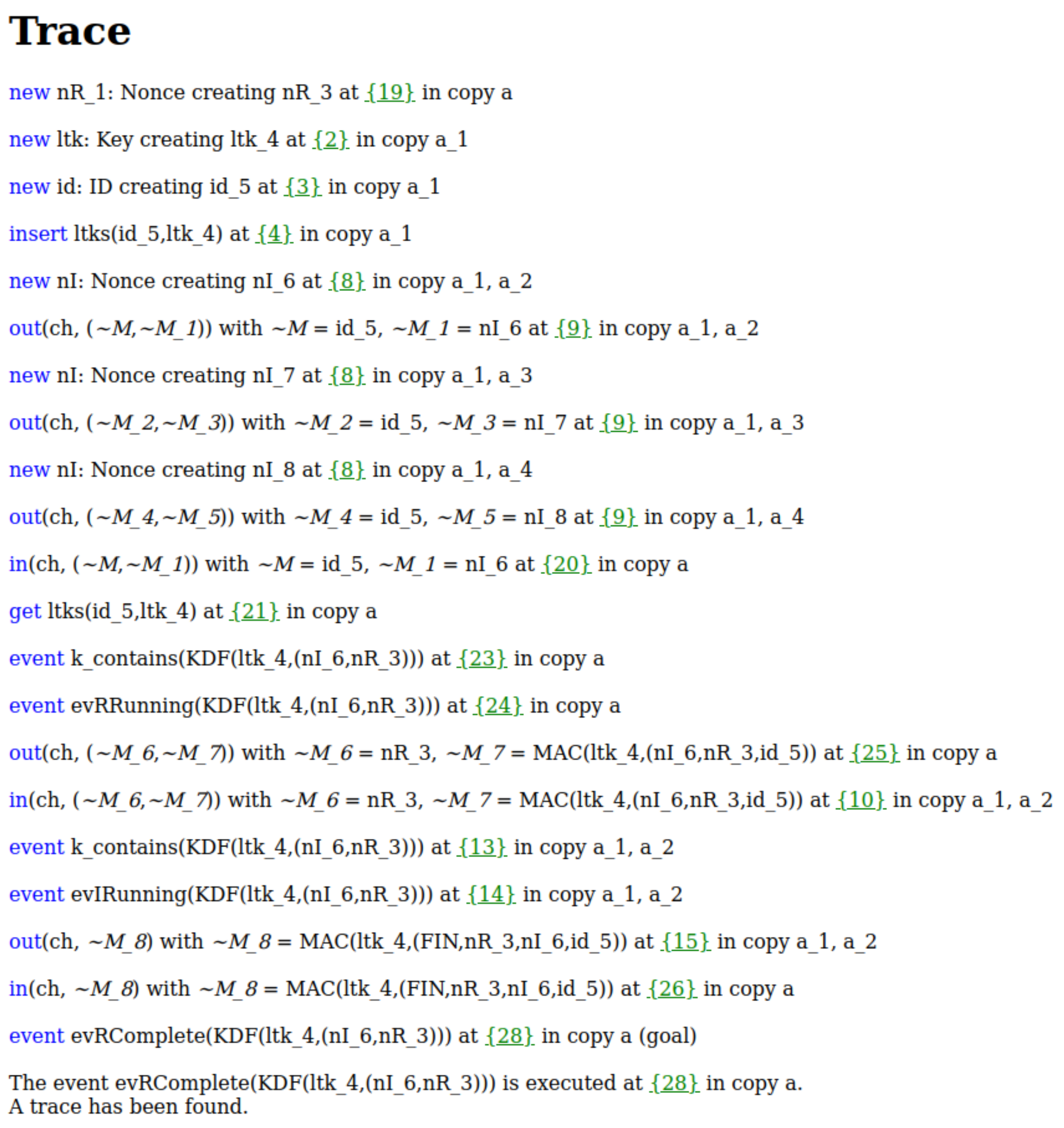
query $x : t_1, y : t_2, z : t_3$; **event**$(e(x, y)) \implies$ **event**$(e'(y, z))$.

means that, for all x, y, for each occurrence of $e(x, y)$, there is a previous occurrence of $e'(y, z)$ for some z.

Q3: What is the syntactical difference between the two definitions? If you were to model a protocol for KTH students logging in to Canvas and I models the student and R the server, which of the two process definitions would you use? Motivate your answer and consider limitations of the definitions
The syntactical difference between the two definitions is the “!” before the I. When using “!” before the “I” we specify that we want to run in parallel. This means that we create infinite processes that can handle an infinite number of processes. The two situations mean that if we want the login system, each user should be able to create infinitely many different login sessions simultaneously. While the other system will allow for one login.

MILESTONE M1:

Q4: Why does the second query evaluate to false? Hint: it has to do with the fact that queries must hold for all traces and that events are scheduled and appear in the trace like any other action such as , e.g., in, out or get.
The event evRComplete(k) will in some cases be reached before evIComplete(k). That's why it evaluates to false.



```
Trace

new nR_1: Nonce creating nR_3 at {19} in copy a

new ltk: Key creating ltk_4 at {2} in copy a_1

new id: ID creating id_5 at {3} in copy a_1

insert ltks(id_5,ltk_4) at {4} in copy a_1

new nI: Nonce creating nI_6 at {8} in copy a_1, a_2

out(ch, (~M,~M_1)) with ~M = id_5, ~M_1 = nI_6 at {9} in copy a_1, a_2

new nI: Nonce creating nI_7 at {8} in copy a_1, a_3

out(ch, (~M_2,~M_3)) with ~M_2 = id_5, ~M_3 = nI_7 at {9} in copy a_1, a_3

new nI: Nonce creating nI_8 at {8} in copy a_1, a_4

out(ch, (~M_4,~M_5)) with ~M_4 = id_5, ~M_5 = nI_8 at {9} in copy a_1, a_4

in(ch, (~M,~M_1)) with ~M = id_5, ~M_1 = nI_6 at {20} in copy a

get ltks(id_5,ltk_4) at {21} in copy a

event k_contains(KDF(ltk_4,(nI_6,nR_3))) at {23} in copy a

event evRRunning(KDF(ltk_4,(nI_6,nR_3))) at {24} in copy a

out(ch, (~M_6,~M_7)) with ~M_6 = nR_3, ~M_7 = MAC(ltk_4,(nI_6,nR_3,id_5)) at {25} in copy a

in(ch, (~M_6,~M_7)) with ~M_6 = nR_3, ~M_7 = MAC(ltk_4,(nI_6,nR_3,id_5)) at {10} in copy a_1, a_2

event k_contains(KDF(ltk_4,(nI_6,nR_3))) at {13} in copy a_1, a_2

event evIRunning(KDF(ltk_4,(nI_6,nR_3))) at {14} in copy a_1, a_2

out(ch, ~M_8) with ~M_8 = MAC(ltk_4,(FIN,nR_3,nI_6,id_5)) at {15} in copy a_1, a_2

in(ch, ~M_8) with ~M_8 = MAC(ltk_4,(FIN,nR_3,nI_6,id_5)) at {26} in copy a

event evRComplete(KDF(ltk_4,(nI_6,nR_3))) at {28} in copy a (goal)

The event evRComplete(KDF(ltk_4,(nI_6,nR_3))) is executed at {28} in copy a.
A trace has been found.
```

Q5: Correct the second query using a more appropriate event and verify that it now evaluates to true. Explain why your change works better

event(evRComplete(k)) ⇒ event (evIRunning(k))

OLD ONE: event(evRComplete(k)) ==> event(evIComplete(k)).

MILESTONE M2:

4.1: Implementation
Write a let-expression in the I-process that assigns the encryption (enc) of aMessage using the derived session key k to a variable ct. Make sure that you place the expression after all inputs to enc are accessible, but before I emits the evReachI event (we still want to verify that the process can complete).

Following creation of ct, add a line that sends a tuple containing ct and a MAC of ct keyed by k. Send the tuple using the out function on channel ch. If you are unsure how to form the MAC, look at the previous MAC computed by I.

Correspondingly, in the R-process, write code to receive the message. After R considers the key establishment phase completed, and has emitted the evRComplete(k) event, add a line to the effect of R reading input from the network. More precisely, add an in expression that reads from channel ch a tuple of two bitstrings, the first called ct and the second called mac2.

Once in has assigned values to ct and mac2 add an if-statement testing whether mac2 equals the function symbol MAC applied to k and ct. If the test is successful, R should emit the event evRecvAMessage(dec(k, ct)) to the trace.

Events must be explicitly declared, so add a new event evRecvAMessage(bitstring) at the top of the model.

4.2: Verification
You will verify two properties, that the message aMessage is still secret after being transmitted, and that if R has received a message and considers its integrity valid, then that message must be aMessage.

The latter means that the attacker cannot fool R into accepting any other message.
Look up the attacker query in the ProVerif manual and add such a query to verify that aMessage is still secret.

Run the tool and verify that this is the case.

```
type FINMarker.
type ID.
type Nonce.
type Key.

table ltks(ID, Key).

free psk:Key [private].
free ch:channel.
free aMessage:bitstring [private].

fun KDF(Key, bitstring):Key.
fun MAC(Key, bitstring):bitstring.
fun enc(Key, bitstring):bitstring.
reduc forall k:Key, b:bitstring;  dec(k, enc(k, b)) = b.

free FIN:FINMarker.

event evIRunning(Key).
event evRRunning(Key).
event evIComplete(Key).
event evRComplete(Key).
event evReachI.
event evReachR.
event evRecvAMessage(bitstring).

(* client *)
let Idummy(Itk:Key, id:ID) = 0.

let I(Itk:Key, id:ID) =
   new nI:Nonce;
   out(ch, (id, nI));
   in(ch, (nR:Nonce, mac:bitstring));
   if mac = MAC(Itk, (nI, nR, id)) then
      let k = KDF(Itk, (nI, nR)) in
      event evIRunning(k);
      out(ch, (MAC(Itk, (FIN, nR, nI, id))));
      event evIComplete(k);
      (* Section 4: Addition of transmission of an encrypted and MACed message.
       * Three parts are needed: this sending, the receiving code in R,
       * and the new properties attacker (aMessage) and that R received
       * the aMessage.
       * In all three places the relevant code is encapsulated in
       * BEGIN/END comments.
       *)
      (* BEGIN *)
      let ct = enc(k, aMessage) in
      event evSendAMessage(aMessage);
      out(ch, (ct, MAC(k, ct)));
      (* END *)
      event evReachI.

(* server *)
let R() =
   new nR:Nonce;
   in(ch, (id:ID, nI:Nonce));
   get Itks(=id, ItkPeer:Key) in
   let k = KDF(ItkPeer, (nI, nR)) in
   event evRRunning(k);
   out(ch, (nR, MAC(ItkPeer, (nI, nR, id))));
   in(ch, mac:bitstring);
   if mac = MAC(ltkPeer, (FIN, nR, nI, id)) then
      event evRComplete(k);
      (* Section 4: Addition of transmission of an encrypted and MACed
       * message.
       *)
      (* BEGIN *)
      in(ch, (ct:bitstring, mac2:bitstring));
      if mac2 = MAC(k, ct) then
         event evRecvAMessage(dec(k, ct));
      (* END *)
      event evReachR.

(* Sanity *)
query event(evReachI).
query event(evReachR).

(* Session key secrecy *)
query secret k.

(* Key Authentication
 * The idea is that I and R shold agree on the output key from the KDF
 *)
query k:Key; event(evIComplete(k)) ==> event(evRRunning(k)).
query k:Key; event(evRComplete(k)) ==> event(evIRunning(k)).

(* Section 4: Addition of transmission of an encrypted and MACed message.
 * The "attacker query" is used to check secrecy of built up terms.
 * The "secret query" is used to check secrecy of bound names or variables.
 *)
(* BEGIN *)
query attacker(aMessage).
query message:bitstring; event(evRecvAMessage(message))  ==> message = aMessage;
(* END *)

process
   !(new ltk:Key; new id:ID; insert ltks(id, ltk); !I(ltk, id)) | !(R)
```