Symbolic Verification of Cryptographic Protocols

Unbounded Process Verification with Proverif

David Baelde

LSV, ENS Paris-Saclay

2019–2020
Introduction

Proverif

Protocol verifier developed by Bruno Blanchet at Inria Paris since 2000
- Analysis in formal model: secrecy, correspondences, equivalences, etc.
- Based on applied pi-calculus, Horn-clause abstraction and resolution
- The method is approximate but supports unbounded processes

Highly successful, works for most protocols including industrial ones: certified email, secure filesystem, Signal messaging, TLS draft, avionic protocols, etc.

These lectures
- Theory and practice of Proverif
- Secrecy, correspondences, equivalences
Terms

As usual in the formal model, messages are represented by terms

- built using constructor symbols from \( f \in \Sigma_c \)
- quotiented by an equational theory \( E \);
- notation: \( M \in \mathcal{M} = \mathcal{T}(\Sigma_c, \mathcal{N}) \).

Additionally, computations are also modeled explicitly

- terms may also feature destructor symbols \( g \in \Sigma_d \);
- semantics given by reduction rules \( g(M_1, \ldots, M_n) \rightarrow M \);
- yields partial computation relation \( \Downarrow \) over \( \mathcal{T}(\Sigma, \mathcal{N}) \times \mathcal{M} \).

Intuition:

- use constructors for total functions,
- destructors when failure is possible/observable.
Example primitives

**Symmetric encryption**

type key.
fun enc(bitstring,key):bitstring.
reduc forall m:bitstring, k:key;
  dec(enc(m,k),k) = m.

**Block cipher**

type key.
fun enc(bitstring,key):bitstring.
fun dec(bitstring,key):bitstring.
equation forall m:bitstring, k:key; dec(enc(m,k),k) = m.
equation forall m:bitstring, k:key; enc(dec(m,k),k) = m.

**Exercise:** how would you model signatures?
Processes

Similar to the one(s) seen before, with a few key differences:
- variables are typed (more on that later);
- private channels, phases, tables, events, etc.

Concrete syntax

\[ P, Q ::= 0 \mid (P|Q) \mid !P \mid \text{new } n:t;P \]
\[
\mid \text{in}(c,x:t);P \mid \text{out}(c,u);P
\]
\[
\mid \text{if } u = v \text{ then } P \text{ else } Q
\]
\[
\mid \text{let } x = g(u_1,\ldots,u_N) \text{ in } P \text{ else } Q
\]

where \( u, v \) stand for constructor terms.

More details in reference manual:

First examples

File structure

- **Declarations**: types, constructors, destructors, public and private data, processes...
- **Queries**, for now only secrecy: query attacker(s).
- **System specification**: the process/scenario to be analyzed.

**Demo**: hello.pv (basic file structure and use).

**Demo**: types.pv (on the role of types).
Correspondences

Roughly, express that if X happens then Y must have happened.

- If B thinks he’s completed the protocol with A, then A thinks he’s completed the protocol with B.

Events

Add events to the syntax of protocols:

(* Declaration *)

event evName(type1,..,typeN).

(* Use inside processes *)

P ::= ... | event evName(u1,..,uN); P

Semantics extended as follows:

\[(\text{event } E. \ P \ | \ Q, \Phi) \xrightarrow{\tau} (P \ | \ Q, \Phi)\]
Queries

Definition

The query

\[
\text{query } x_1:t_1, \ldots, x_N:t_K; \\
\text{event}(E(u_1,\ldots,u_N)) \implies \text{event}(E'(v_1,\ldots,v_M))
\]

holds if for all traces of the system

- if the trace ends with an event rule for an event of the form \( E(u_i) \),
- there is a prior execution of the rule for an event of the form \( E'(v_j) \).

Note that variables of \( u_i \) are \textit{universally} quantified
while those only occurring in \( v_j \) are \textit{existentially} quantified.

Example

\[
\text{query } \text{na:bitstring, nb:bitstring}; \\
\text{event}(\text{endR}(pka,pkb,na,nb)) \implies \text{event}(\text{endI}(pka,pkb,na,nb)).
\]
Model the Needham-Schroeder public key protocol from the first lecture by completing the nspk.pv file.

In that file, declare a system that allows for the man-in-the-middle attack, and ask Proverif to check the secrecy of $n_b$. It should find the attack.

Finally, fix the protocol as proposed during the first lecture, check that secrecy holds. You may then try to check authentication using correspondences.
Exercise: injectivity

Proverif also allows to check injective correspondences:

```
query x1:t1, .., xN:tK;
    inj-event(E(u1,..,uN)) => inj-event(E'(v1,..,vM))
```

holds if for all traces of the system there is an injective \( \phi \) such that

- if an event of the form \( E(u_i) \) is emitted at step \( \tau \),
- an event of the form \( E'(v_j) \) is emitted at step \( \phi(\tau) < \tau \).

Exercise:

1. Check that NSL satisfies mutual authentication in its injective form, which is the proper form.
2. Give a protocol that satisfies mutual authentication only in its non-injective form.