Security II - ProVerif

Stefano Calzavara

Università Ca' Foscari Venezia

April 30, 2020

Università
Ca'Foscari
Venezia

Introduction

ProVerif is a state-of-the-art protocol verification tool

- `prosecco.gforge.inria.fr/personal/bblanche/proverif/`
- accepts protocols expressed in a dialect of the applied pi-calculus
- supports verification of secrecy and authentication properties

Three possible outputs:

- safe: the security property cannot be violated
- unsafe: the security property might be violated, i.e., ProVerif finds a counter-example (which might be a false positive)
- unsure: cannot prove security or find counter-examples

Occasionally, ProVerif might not terminate (!)

Structure of ProVerif Files

A ProVerif files normally includes:

1. declaration of names, constructors, destructors: the symbols that we can use to write processes
2. definition of (parallel) processes: the protocol that we want to verify
3. definition of security queries: the goals of the security analysis

We will use the untyped syntax of ProVerif, which is easier to use

- no need of explicit type annotations
- execute with `proverif -in pi file.pv`

Names and Variables

Names are declared with the syntax:

```
free id1, ..., idn.
```

All names are public by default, unless you prepend the line with the "private" keyword. This is equivalent to using the restriction operator.

No need to declare variables: they are automatically introduced and bound when using inputs and lets.

Constructors and Destructors

Constructors are declared with the syntax:

```
fun const/n.
```

Destructors are defined by their equations:

```
reduc id(M1,...,Mn) = N.
```

Example

```
fun senc/2.
reduc sdec(senc(x,y),y) = x.
```

Processes

It is possible to define process macros by using the "let" keyword:

```
let ident = process-definition.
```

Macros can then be used in the process modeling the protocol to verify:

```
process new a; ...; new k (ident1 | ident2)
```

Example

```
let init = out(c,m); in(d,x).
let relay = in(c,y); out(d,y).
process !init | !relay
```

Queries

Secrecy queries:

```
query attacker: m.
```

Non-injective agreement:

```
query ev:end(x,y,z) ==> ev:begin(x,y,z).
```

Injective agreement:

```
query evinj:end(x,y,z) ==> evinj:begin(x,y,z).
```

Example

```
free a, b, c.
fun senc/2.
reduc sdec(senc(x,y),y) = x.

query attacker: m.
query ev:end(x,y,z) ==> ev:begin(x,y,z).

let sender = event begin(a,b,m); out(c,senc(m,k)).

let receiver =
        in(c,xm);
        let ym = sdec(xm,k) in event end(a,b,ym).

process
        new k; new m; (!sender | !receiver)
```

Demo Time!

Under the Hood

ProVerif works by translating crypto protocols into logical formulas

- sound: if no violation can be proved by the logical formulas, then the protocol is secure
- incomplete: if a violation can be proved by the logical formulas, then the protocol might still be secure (false positive)

More precisely, ProVerif is based on Horn clauses

$$\forall \vec{x}.(p_1(M_1) \wedge \ldots \wedge p_k(M_k) \Rightarrow q(N)).$$

The resolution algorithm takes a set of Horn clauses and a goal $\exists \vec{x}.p(M)$ and checks whether the goal is provable from the clauses.

Checking Secrecy

We sketch here the ProVerif's approach to checking secrecy

$$\begin{array}{rcll} F & ::= & \text{attacker}(M) & \text{the attacker knows } M \\ & | & \text{mess}(L, M) & \text{the message } M \text{ is output on channel } L \end{array}$$

Goal: prove that attacker(N) does not hold for the secret N

Translation

Given a process P and a set of public names C, ProVerif outputs a set of Horn clauses $H(P, C)$:

$$H(P, C) = \mathit{AtkKnows}(C) \cup \mathit{AtkRules} \cup \mathit{ProtRules}(P).$$

Modeling the Attacker

Modeling the initial attacker's knowledge is straightforward:

$$AtkKnows(C) = \{\text{attacker}(n) \mid n \in C\}$$

The attacker's knowledge takes advantage of the equational theory:

- for each constructor f of arity n:

$$\text{attacker}(x_1) \wedge \ldots \wedge \text{attacker}(x_n) \Rightarrow \text{attacker}(f(x_1, \ldots, x_n))$$

- for each destructor g such that $g(M_1, \ldots, M_k) = N$:

$$\text{attacker}(M_1) \wedge \ldots \wedge \text{attacker}(M_k) \Rightarrow \text{attacker}(N)$$

Modeling the Attacker

The attacker's knowledge increases during the protocol run:

- the attacker can read from known channels:

$$\text{mess}(x, y) \wedge \text{attacker}(x) \Rightarrow \text{attacker}(y)$$

- the attacker can write known information on known channels:

$$\text{attacker}(x) \wedge \text{attacker}(y) \Rightarrow \text{mess}(x, y)$$

Modeling the Protocol

Each output statement $\overline{c}\langle N \rangle$ generates a Horn clause of the form:

$$\text{mess}(c_1, M_1) \wedge \ldots \wedge \text{mess}(c_k, M_k) \Rightarrow \text{mess}(c, N),$$

where $M_1, \ldots, M_k$ are the previously received messages.

Example

Consider the process $c(x).c(y).\overline{c}\langle (x, y) \rangle$, ProVerif generates:

$$\text{mess}(c, x) \wedge \text{mess}(c, y) \Rightarrow \text{mess}(c, (x, y))$$

Example: Unsafe Protocol

Protocol P: Assume A sends B the message s over channel net

- initial attacker's knowledge: $\mathit{AtkKnows}(\{net\}) = \{\text{attacker}(net)\}$
- attacker's rules: $\text{attacker}(x) \wedge \text{mess}(x, y) \Rightarrow \text{attacker}(y) \in \mathit{AtkRules}$
- protocol rules: $\mathit{ProtRules}(P) = \{\text{mess}(net, s)\}$

We can show that:

$$H(P, C) \vdash \text{attacker}(s),$$

which correctly suggests that the secrecy of s does not hold for P.

Final Advices

Typos

Undeclared identifiers are assumed to be public names, check your output for warnings!

Modeling Assumptions

Ensure your model is capturing reality! What does the attacker know? Can the same participant play multiple roles in the protocol?

Reachability Queries

You can use the syntax "query ev:end(x,y,z)" to ensure that an end event is actually reachable (this should return false)