WALTON Institute for Information and Communication Systems Science

SETU Ollscoil Teicneolaíochta an Oirdheiscirt South East Technological University

Formal Verification of Cryptographic Protocols

Ilias Cherkaoui

ProVerif slides are based on Blanchet & Cheval's documentation

Motivation

- Ignored implementation details
- Use of obsolete modes
- Need for a synchronous automatic check
- Handling concurrent and serial processes.

Projects

- ProVerif: Automated tool for verification using a classic symbolic model (Dolev-Yao).
- CryptoVerif: Mechanical verification using computation proofs.
- Tamarin Prover: Heuristic, deduction and equational reasoning.
- F*: Type-checker using SMT (satisfiability modulo theories) and manual proofs (Ocaml..)

- Deepsec: Bounded number of sessions, security properties expressed as trace equivalences.
- Scyther: Inference for proving or disproving a protocol.
- CertiCrypt: Coq-based for mechanised proofs in a computational model.
- EasyCrypt: Uses SMT and axiomatic rules to improve CertiCrypt timing.

Pi-calculus syntax

- N = {a, b, c, d, ...} interconnection names
- P := 0 null process

output

- a(x).P input
- P | P parallel

- vx. P — restriction
- ! P — replication
- P + P — sum/choice
- |a(x).Q ⇒ P|Q {b/x} — communication

Proverif Internal Mechanism

- 1- Processes
- 2- Horn Clauses
- 3- Saturation
- 4- Verification using all lemmas and axioms.

ProVerif Initiation

On command line: ./proverif [options] ⟨filename⟩

Script:

(* This is a comment*)

free c : channel.

free RSA: bitstring [**private**].

query attacker (RSA) .

Process

out (c ,RSA) ;

0

```
Process 0 (that is, the initial process):
{1}out(c, RSA)

-- Query not attacker(RSA[]) in process 0.
Translating the process into Horn clauses...
Completing...
Starting query not attacker(RSA[])
goal reachable: attacker(RSA[])

Derivation:

1. The message RSA[] may be sent to the attacker at output {1}.
attacker(RSA[]).

2. By 1, attacker(RSA[]).
The goal is reached, represented in the following fact:
attacker(RSA[]).


A more detailed output of the traces is available with
  set traceDisplay = long.

out(c, ~M) with ~M = RSA at {1}

The attacker has the message ~M = RSA.
A trace has been found.
RESULT not attacker(RSA[]) is false.
----------------------------------------------------------------
Verification summary:

Query not attacker(RSA[]) is false.
```

- Proverif tests the query not attacker(RSA), it is true when the name is not derivable by the attacker.

- The attacker has, however, been able to obtain the free name RSA as denoted by the RESULT not attacker:(RSA[]) is false.

- ProVerif is also able to provide an attack trace:

  out(c, ~M) with ~M = RSA at {1}

  The attacker has the message ~M = RSA.

Operations

- Constructors are used to build terms: f() for instance, a shared-key encryption would be denoted by:

fun senc(bitstring, key) : bitstring**.**

- Destructors use rewrite rules g() M.

let x= g() **in** *P* **else** *Q*

It can be seen in decryption:

fun senc(bitstring, key) : bitstring

reduc forall m : bitstring, k : key; sdec(senc(m, k), k) = m.

In other words,

- for each constructor *f* of arity *n*, the clause

$$att() \wedge \ldots \wedge att() \Rightarrow att(f(,\ldots,))$$

is generated, representing that the adversary can compute *f(,...,)* by applying *f*

when it has , . . . , .

For instance, for shared-key encryption *senc*, the following clause is generated:

$$att(m) \wedge att(k) \Rightarrow att(senc(m, k))$$

- For each destructor *g*, defined by a rewrite rule *g(,...,)* → *M*, the clause

  att() ∧ ... ∧ att() ⇒ *att(M)*

is generated, representing that the adversary can compute *M* when it has *,..., ,* by applying *g*.

For instance, for shared-key decryption *sdec*, defined by *sdec(senc(m, k), k)* → *m*, the following clause is generated:

att(senc(m, k))∧att(k) ⇒ att(m)

If the adversary has the ciphertext *senc(m, k)* and key *k*, it can obtain the cleartext *m* by decryption.

Denning-Sacco Protocol

Message 1. $A \to B : \quad \{\{k\}_{sk_A}\}_{pk_B} \quad k$ fresh
Message 2. $B \to A : \quad \{s\}_k$

$$\text{new } sk_A.\text{new } sk_B.\text{let } pk_A = \text{pk}(sk_A) \text{ in let } pk_B = \text{pk}(sk_B) \text{ in}$$
$$\text{out}(c, pk_A).\text{out}(c, pk_B).$$

(A) $\quad ! \text{ in}(c, x\_pk_B).\text{new } k.\text{out}(c, \text{aenc}(\text{sign}(k, sk_A), x\_pk_B)).$
$\quad\quad \text{in}(c, x).\text{let } s = \text{sdec}(x, k) \text{ in } 0$

(B) $\quad \| \quad ! \text{ in}(c, y).\text{let } y' = \text{adec}(y, sk_B) \text{ in}$
$\quad\quad \text{let } k = \text{check}(y', pk_A) \text{ in out}(c, \text{senc}(\text{s}, k))$

WALTON Institute for Information and Communication Systems Science

Ollscoil Teicneolaíochta an Oirdheiscirt
South East Technological University

Thank you.

Formal Verification of Cryptographic Protocols

Ilias Cherkaoui