Verifying security protocols using the ProVerif tool

Security protocols (bmevihim132)

Ta Vinh Thong
Phd student
BME Híradástechnikai Tanszék
Lab of Cryptography and System Security (CrySyS)
thong at crysys dot hu
Outline

- Introduction and overview
- Proverif automatic verification tool
Introduction and overview

- Motivation
  - Informal reasoning is error-prone.
    - Miss attacks, not systematic, not automatic.
    - Informal specification languages are often not unequivocal.

- Advantage of formal methods
  - Sound mathematics background.
  - Systematic, automatic proofs.
    - Finite automaton, resolution, bisimulations.
  - Expressive, unambiguous specification languages
    - Formal logic, process algebra (calculus).
Verifying security protocols

- What we need (formal syntax, semantics)?
  - cryptographic primitives and operations.
  - trusted/untrusted communication channels.
  - secrets keys, chipertext and cleartext.
  - security properties such as: secrecy, authenticity, anonymity.
  - attacker behaviour.

- Formal methods and tools proposed for security protocols
  - Logic: BAN logic.
  - Process algebra: spi-calculus, **applied pi-calculus**.
  - Automatic tools (based on algebra): CSP, **ProVerif**.
Abstractions

- Cryptography is assumed to be „perfect”
  - Attacker cannot learn anything from a ciphertext unless it has a secret key.
  - Attacker cannot learn even a small part of a ciphertext without a secret key.
  - If the attacker modifies ciphertexts without the proper key it will be detected.
  - Hash functions cannot be broken and forged.
  - Digital signatures and MACs cannot be forged without a secret key.
  - Random numbers, nonces, secret keys are unpredictable and unguessable.

- Find attacks that are independent of cryptographic algorithms and focus only on the weaknesses resulted from the careless protocol design.
Security properties

- **Secrecy**: Prevent unauthorized disclosure of secrets.
  - **Weak secrecy**: The attacker cannot obtain the secrets.
  - **Strong secrecy**: The attacker cannot even deduce any partial information about the secrets. More precisely, the attacker cannot distinguish one secret from the others. (E.g., guessing attack)

- **Authenticity**: Verifying user identity, message origin.

- **Integrity**: Prevention of unauthorized modification of messages.

- **Anonimity**: Preserve the privacy of users.
Proverif verification tool

Input
- Security protocol specified in syntax of applied pi-calculus

Automatic verification
- Protocol rules
- Attacker rules
- Horn-clauses
- Resolution based deduction algorithm

Output
- Yes
- No (attack trace)

PROVERIF tool

- A kind of theorem prover (based on logic), but fully automated.
- Security protocols are specified in a simplified form of applied pi calculus.
- Verify strong secrecy, weak secrecy, forward secrecy, authenticity, anonymity.
Variants of the pi-calculus

- Pure pi-calculus [Robin Milner. 1990]
  - Calculus for concurrent computation.
  - Main features: fresh names, communication channels, name passing, recursion.

- Spi-calculus [Martin Abadi, Andrew Gordon. 1997]
  - Extend the pure pi-calculus with fixed number of crypto primitives.
  - Main features: prove security properties using bisimulations.

- Applied pi-calculus [Martin Abadi, Cédric Fournet. 2001]
  - Extend spi-calculus with additional bisimilarities, more crypto primitives.
  - Main features: much easier usage to prove security properties.
Protocol examples – 1

• Syntax

Alice = out(c, m). in(c, y). 0
Bob = in(c, x). out(c, hash(x)). 0
Protocol = new m; (Alice | Bob)

m
hash(m)
c

Alice

Bob
• Syntax

Alice = out(c,(n,senc(hash(n),k))).

Bob = in(c,x). let (x_1,x_2) = x in let y = sdec(x_2,k) in
    if y = hash(x_1) then out(c,n).

Protocol = new m;(Alice|Bob)
Security properties

(* protocol without the attacker *)

\[ Prot = (Alice|Bob) \]

(* protocol including the attacker *)

\[ Protocol = (Alice|Bob) | \text{AttackerMallory} \]

- Reachability or weak secrecy of some secret \( M \): There is a trace of process \( Protocol \) that derive \( M \) in clear form.

\[ (*) \text{ Weak secrecy, reachability of } M (*) \]

\[ \exists \rightarrow^\ast \text{ such that: } Protocol \rightarrow^\ast M \]

- Strong secrecy of some secret \( x \): Communicating with \( Prot(M) \) and \( Prot(M') \), the attacker cannot distinguish \( Prot(M) \) from \( Prot(M') \). Where \( M, M' \) are two arbitrary values of the secret \( x \).

\[ (*) \text{ Strong secrecy of } M : \text{Observational equivalent } (*) \]

\[ Protocol(M) \approx Protocol(M') \]
ProVerif installation

- **Webpages**
  - Online demos: [http://server1.dsuresearch.org/ProVerif/demo/index2.php/](http://server1.dsuresearch.org/ProVerif/demo/index2.php/)
  - GUI Editor: [http://sourceforge.net/projects/proverifeditor/](http://sourceforge.net/projects/proverifeditor/)

- **ProVerif’s newest version: 1.86pl3.**
  - Devepoders: Bruno Blanchet, and his group. Ecole Normale Supérieure, France.
  - Developed in *Ocaml* logic programming language.
  - By default, has no GUI interface.

- **Installation in Linux/Mac** (download Ocaml 3.0, [http://caml.inria.fr/](http://caml.inria.fr/))
  - Create a directory `proverif1.86pl3` in the current directory.
    - Using GNU tar: 1) `tar -xzf proverif1.86pl3.tar.gz` ; 2) `tar -xzf proverifdoc1.86pl3.tar.gz`
    - Using `tar`: 1.) `gunzip proverif1.86pl3.tar.gz`; 2.) `tar -xf proverif1.86pl3.tar`; 3.) `gunzip proverifdoc1.86pl3.tar.gz`; 4) `tar -xf proverifdoc1.86pl3.tar`.
  - Build the ProVerif
    - `cd proverif1.86pl3`
    - `./build`
ProVerif installation, execution

Installation in Windows
- From binary
  - Does not require Ocaml !!!
  - Decomposes: `proverifbsd1.86pl3.tar.gz` and `proverifdoc1.86pl3.tar.gz` archives in the same directory.
- From source
  - Requires Ocaml installation, OCaml bytecode compiler.
  - Decomposes: `proverif1.86pl3.tar.gz` and `proverifdoc1.86.tar.gz` archives in the same directory.
  - In the command shell go to the directory where the two files were extracted and build `./build.bat`

Execution
- Command line: `./proverif [options] < filename.pv`
  - Windows: 1) open cmd; 2) cd to the directory; 3) `./proverif [options] < filename.pv`
  - `./proverif`: ProVerif’s binary
  - `< filename.pv>`: ProVerif’s source file.
    - By default, only honest processes should be defined by users.
  - `[options]`: options
**ProVerif’s features**

- Designed specifically for analysing security protocols.
- Expressive syntax and semantics for crypto primitives and operations.
- Modelling attacker.
- Verifying weak and strong secrecy, forward secrecy, authenticity, privacy.
- Able to handle more complex data structures than general purpose model-checking tools.
- Fast and sound. Return the attack trace when an attack is found.

- Not provide explicit time issues. E.g., protocol that based on timers.
  - UPPAAL model-checker tool can handle timer.
- Not provide probability issues, E.g., probabilistic protocols.
  - PRISM model-checker tool can handle probability.
- Cannot model partial information leakage
  - Such attacks in which the attacker periodically deduces bits of the secret until getting the whole secret.
- Limited possibility to model complex protocols that uses data cache/storage.
  - ProVerif provides the modelling of storage but only the operation read and add.
- Cryptography is assumed to be perfect.
- Not complete, sometimes does not terminate.
Assumptions made in ProVerif

- Communication channels are often untrusted.

- Adversary is a Dolev-Yao type attacker
  - Complete control of communication channels
  - Read, modify, delete and inject messages.

- Cryptography is assumed to be perfect.
Term and processes in ProVerif (typed applied pi-calculus)

\[
M, N ::= \\
\quad a, b, c, k, m, n, s \\
\quad x, y, z \\
\quad (M_1, \ldots, M_k) \\
\quad h(M_1, \ldots, M_k)
\]

\[C ::= \]
\[
M = N \\
M \not\leftrightarrow N \\
M \\
C \land C \\
C \lor C \\
\neg(C)
\]

\[
P, Q ::= \]
\[
0 \\
P | Q \\
!P \\
\text{new } n : t; P \\
\text{in}(M, x : t); P \\
\text{out}(M, N); P \\
\text{if } C \text{ then } P \text{ else } Q \\
\text{let } x = M \text{ in } P \text{ else } Q \\
R(M_1, \ldots, M_k)
\]

terms
- names
- variables
- tuple
- constructor/destructor application

conditions
- term equality
- term inequality
- term (of type bool)
- conjunction
- disjunction
- negation

processes
- null process
- parallel composition
- replication
- name restriction
- message input
- message output
- conditional
- term evaluation
- macro usage
Pattern matching

\[ T ::= \]
\[ x : t \]
\[ x \]
\[ (T_1, \ldots, T_n) \]
\[ =M \]

patterns

typed variable
variable without explicit type
tuple
equality test

• Variable pattern \( x:t \) matches any term of type \( t \) and binds the matched term to \( x \).
• \( x \) can be used only when the type of \( x \) can be inferred from the context.
• Tuple pattern \((T_1, \ldots, T_n)\) matches tuples \((M_1, \ldots, M_n)\) where each component \( M_i \) is recursively matched with \( T_i \).
• The pattern \( =M \) matches terms \( N \) where \( M = N \).
  • \( \text{in}(=M);P \) waits a term that matches pattern \( M \), then behaves as \( P \).

Input e.g.,: \( \text{in}(c, x:t):Q \) can be written as \( \text{in}(c, =M):Q \)
ProVerif’s input file – 1

- Declaration, process definition, queries

\[
< \text{decl} > ::= \quad \text{declaration} \\
\text{type } t \quad \text{user defined type} \\
\text{free name : } t \quad \text{public free names} \\
\text{free name : } t \ [\text{private}] \quad \text{private free names} \\
\text{const } c : t \quad \text{constant } c \text{ of type } t \\
\text{table } d(t_1, \ldots, t_n) \quad \text{table } d \text{ takes records of type } t_1, \ldots, t_n \\
\text{fun } f(t_1, \ldots, t_n) : t \quad \text{public constructor function} \\
\text{fun } f(t_1, \ldots, t_n) : t \ [\text{private}] \quad \text{private constructor function} \\
\text{reduc forall } x_1 : t_1, \ldots, x_n : t_n; g(M_1, \ldots, M_k) = M \quad \text{public destructor application} \\
\text{reduc forall } x_1 : t_1, \ldots, x_n : t_n; g(M_1, \ldots, M_k) = M \ [\text{private}] \quad \text{private destructor application}
\]
ProVerif’s input file – 2

< procmacro > ::= 

let R(x₁,...,xₙ) = P.  sub-process definition

define a process with the name R.

< mainproc > ::= 

process P main process definition

< query > ::= 

query attacker(M). define properties to be verified.

want to verify weak secrecy of M .

query event(M) ==> event(N). in any trace: event N is always before event M.

query inj - event:M ==> inj - event:N. in any trace: at least as many N - events as M - events.

noninterf x want to verify strong secrecy of x.

noninterf x among {M₁,...,Mₖ} want to verify strong secrecy of x taken value from a set.

The structure of the whole ProVerif source file is

< decl >* < query >* < procmacro >* < mainproc >
Types and Names

<table>
<thead>
<tr>
<th>type t</th>
<th>user defined type.</th>
</tr>
</thead>
<tbody>
<tr>
<td>free name:t</td>
<td>free names.</td>
</tr>
<tr>
<td>free name:t [private]</td>
<td>private free names.</td>
</tr>
</tbody>
</table>

- User defined type can be anything except for pre-defined, built-in types.
  - User defined: bitstring, skey, pkey, etc.
- Free names model public data, which are available for the attacker.
  - Public keys, communication channels, participant IDs, etc.
- Private names model secret data, which are not known by the attacker.
  - Secret keys, etc.
- Identifiers range over an unlimited sequence of letters (a-z, A-Z), digits (0-9), underscores (_), singlequotes (’), and accented letters from the ISO Latin 1 character set.
  - E.g. free secretKey: skey.
**Constructor functions**

<table>
<thead>
<tr>
<th>fun</th>
<th>( f(t_1, \ldots, t_n) : t )</th>
<th>public constructor function</th>
</tr>
</thead>
<tbody>
<tr>
<td>fun</td>
<td>( f(t_1, \ldots, t_n) : t \ [private] )</td>
<td>private constructor function</td>
</tr>
</tbody>
</table>

- Constructor function:
  - \( f \) is a function of arity \( n \).
  - \( t \) is a type of the returned value, and \( t_1, \ldots, t_n \) are types of the arguments.
  - constructor functions are available to all participants (honest and attacker).
  - private constructor function are available to only honest participants.
    - E.g., key databased (table) stored by the honest nodes.
Destructor applications:

- **Destructor applications:**
  - $g$ is a function of arity $k$.
  - $t_1, \ldots, t_n$ are types of the variables $x_1, \ldots, x_n$.
  - $x_1, \ldots, x_n$ are variables in terms $M_1, \ldots, M_k$.
  - Destructor applications are available to all participants (honest and attacker).
  - Private destructor applications are available to only honest participants.

Formal analysis of security protocols
**Constructor and Destructor examples**

### Symmetric encryption (User defined)

```ocaml
type bitstring, key.
fun senc(bitstring, key): bitstring
reduc forall m: bitstring, k : key; sdec (senc (m, k), k) = m.
```

### Asymmetric encryption (User defined)

```ocaml
type bitstring, skey, pkey.
fun pk(skey): pkey
fun penc(bitstring, pkey): bitstring
reduc forall m: bitstring, k: skey; pdec (penc (m, pk(k)), k) = m.
```
Constructor and Destructor examples

Digital signatures (User defined)

```plaintext
type bitstring, sskey, spkey.
fun spk(sskey):spkey
fun sign(bitstring, sskey):bitstring
reduc forall m: bitstring, k : sskey; checksign (sign(m, spk(k)), k) = m.
```

One-way functions (User defined)

```plaintext
type bitstring, key.
fun hash(bitstring):bitstring.
fun mac(bitstring, key).
```

Tuples (built-in)

```plaintext
fun (t₁,...,tₖ):bitstring
reduc forall x₁:t₁,...,xₖ:tₖ; ith((M₁,...,Mₖ)) = Mᵢ
```
Tables and operations

<table>
<thead>
<tr>
<th>table $d(t_1,\ldots,t_n)$</th>
<th>(* declation *)</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert $d(M_1,\ldots,M_n);P$</td>
<td></td>
</tr>
<tr>
<td>get $d(T_1,\ldots,T_n)$ in $P$</td>
<td></td>
</tr>
<tr>
<td>get $d(T_1,\ldots,T_n)$ suchthat $C$ in $P$</td>
<td></td>
</tr>
</tbody>
</table>

- $d$ is the name of the table which takes records of type $t_1, \ldots, t_n$.  
- Processes may populate and access tables, but deletion is forbidden.  
- Process $insert d(M_1, \ldots, M_n); P$ inserts the record $M_1, \ldots, M_n$ into the table $d$ and then executes $P$.  
- Process $get d(T_1, \ldots, T_n)$ in $P$ attempts to retrieve a record in accordance with patterns $T_1, \ldots, T_n$.  
  - When no such record is found, the process blocks.  
  - In case there are several identical records, one is chosen from them.  
- $C$ is an additional condition for the records.
Process Macros, subprocesses

- Defining *sub-processes* that will be used in the main process.
- $R$ is a *macro name*, $P$ is a *sub-process* being defined. $x_1, \ldots, x_n$ of type $t_1, \ldots, t_n$ are variable of $P$.
- $R(M_1, \ldots, M_n)$ will then expand to $P$ with $M_1$ substituted for $x_1$, \ldots, $M_n$ substituted for $x_n$.

\[
\text{let } R(x_1 : t_1, ..., x_n : t_n) = P
\]
Bracketing, precedence

1. \[ \text{new } n : t \text{ ; out}(c, n) \mid \text{new } n : t \text{ ; in}(c, x : t); 0 \mid \text{if } x = n \text{ then } 0 \mid \text{out}(c, n), \] is bracketed as
   \[ \text{new } n : t \text{ ; (out}(c, n) \mid \text{new } n : t; (\text{in}(c, x : t); 0 \mid \text{if } x = n \text{ then } (0 \mid \text{out}(c, n))) \) \]

2. \[ \text{if } M = M' \text{ then if } N = N' \text{ then } P \text{ else } Q, \] is bracketed as
   \[ \text{if } M = M' \text{ then (if } N = N' \text{ then } P \text{ else } Q) \]

3. \[ \text{let } x = M \text{ in let } y = N \text{ then } P \text{ else } Q, \] is bracketed as
   \[ \text{let } x = M \text{ in (let } y = N \text{ then } P \text{ else } Q) \]
Hello word script

1. (* hello.pv : Hello World Script *)
2. free c : channel.
3. free K₁ : bitstring[private].
4. free K₂ : bitstring[private].
5. query attacker(K₁).
6. query attacker(K₂).
7. 
8. process
9. out(c,K₂);
10. 0

- The verification algorithm attempts to prove that the states in which the keys are obtained by the attacker are unreachable. not attacker(K1) is true/false

Public channel
Secret keys
Can Dolev-Yao attacker(s) obtain keys?
Send key K₂ on channel c
Termination of process. Can be omitted.
1. (* Hello World Script with Macros *)
2. free c : channel.
3. free K₁ : bitstring[private].
4. free K₂ : bitstring[private].
5. query attacker(K₁).
6. query attacker(K₂).
7.
8. let R(x : bitstring) = out(c, x);0.
9. let R'(y : bitstring) = 0.
10. process R(K₁) | R'(K₂)
The specified protocol. Each row is labelled with \{n\}, where n is an integer.

- Internally, ProVerif attempts to prove that a state in which a property is violated is unreachable;

- ProVerif shows the (un)reachability of some [Goal].

Verifying the property given in [Query] by user.

Process:
[Process]

-- Query [Query]
Completing...
Starting query [Query]
RESULT [un]reachable : Goal.

[Attack derivation]

set traceDisplay = long.
[Attack trace]

RESULT [Query][result].

Finally, ProVerif reports if the Query was satisfied.
Running result of the hello world script

Process
{1} out(c, K₂)
-- Query not attacker(K₁[])
Completing...
Starting query not attacker(K₁[])
RESULT not attacker(K₁[]) is true.
-- Query not attacker(K₂[])
Completing...
Starting query not attacker(K₂[])
goal reachable : attacker(K₂[]).

The message K₂[] may be sent to the attacker at output{1}. attacker(K₂[]).

set traceDisplay = long.
out(c, K₂) at {1}
The attacker has the message K₂.
A trace has been found.
RESULT not attacker(K₂[]) is false
Queries full syntax

\[ q ::= \]
\[ F \]
\[ F \rightarrow H \]

\[ H ::= \]
\[ F \]
\[ H \&\& H \]
\[ H \parallel H \]
\[ (F \rightarrow H) \]

\[ F ::= \]
\[ \text{attacker}(M) \]
\[ \text{attacker}(M) \text{ phase } n \]
\[ \text{mess}(N,M) \]
\[ \text{mess}(N,M) \text{ phase } n \]
\[ \text{event}(e(M_1,\ldots,M_n)) \]
\[ \text{inj-event}(e(M_1,\ldots,M_n)) \]
\[ M=N \]
\[ M<>N \]

query
fact
\[ \text{correspondence} \]

hypothesis
fact
\[ \text{conjunction} \]
\[ \text{disjunction} \]
\[ \text{nested correspondence} \]

fact
the adversary has \( M \) (in any phase)
the adversary has \( M \) in phase \( n \)
\( M \) is sent on channel \( N \) (in the last phase)
\( M \) is sent on channel \( N \) in phase \( n \)
\( \text{non-injective event} \)
\( \text{injective event} \)
equality
inequality

Formal analysis of security protocols
Ta Vinh Thong, Híradástechnikai Tanszék
Budapesti Műszaki és Gazdaságtudományi Egyetem
Security properties, Queries

• Reachability and weak secrecy
  • which term will be obtained by the attacker during protocol runs.
  • E.g., **query attacker(M)** - can the attacker obtain term M?
    M is ground term, does not contain destructor applications.

• Correspondence assertions, events, authentication
  • if an event e has been executed, then event e’ has been previously executed.
  • processes are annotated with **events**, which mark stages reached by the protocol.
  • but do not affect the behaviour of processes.
  • definition in process: **Process …. event e(M1,...,Mn); P ….**
  • declaration: **event e(t1,...,tn)**. Where t1,...,tn are types of arguments.
  • Query: **query x1:t1,..., xn:tn; event e(M1,...,Mj) ==> event e’ (N1,...,Ni)**
  • for each occurrence of e(M1,...,Mj), event e’ (N1,...,Ni) has occurred before.

• Injective correspondence
  • one-to-one relationship between the number of protocol runs performed by each participant.
  • E.g., Financial transaction
    • server request payment from the client.
    • server should complete payment only once for each transaction started by the client.
  • Query: **query x1:t1,..., xn:tn; inj-event e(M1,...,Mj) ==> inj-event e’ (N1,...,Ni)**
Extended hello word script

1. (* hello _ext.pv : Extended Hello World Script *)
2. free c : channel.
3. free K₁ : bitstring[private].
4. free K₂ : bitstring[private].
5. event recvK₁.
6. event recvK₂.
7. query event(recvK₁) => event(recvK₂)
8. process
9. out(c, K₂);
10. in(c, x : bitstring);
11. if x = K₁ then
12. event recvK₁ ; event recvK₂
13. else event recvK₂
ProVerif output – ext. Hello world

- Process:
  - {1} out(c, RSA);
  - {2} in(c, x: bitstring);
  - {3} if x = K1 then
    - {4} event recvK1;
    - {5} event recvK2
  - else
    - {6} event recvK2

- Query event(recvK1) ==> event(recvK2)
- Completing...
- Starting query event(recvK1) ==> event(recvK2)
- RESULT event(recvK1) ==> event(recvK2) is true.
Naiv handshake protocol

\[
C \rightarrow S : pk_C \\
S \rightarrow C : \{ \{ pk_S , k \} _{sk_S} \} _{pk_C} \\
C \rightarrow S : \{ s \} _k
\]

- Each principal has a public/private keypair.
- The client knows the server’s public key.
- The aim of the protocol: client shares the secret \( s \) with the server.

- Security properties would like to provide
  - **Secrecy**: the value \( s \) is known only to Client and Server.
  - **Authentication of Client to Server**: if Server reaches the end of the protocol and he believes he has shared the key \( k \) with Client, then Client was indeed his interlocutor and she has shared \( k \).
  - **Authentication of Server to Client**: if Client reaches the end of the protocol with shared key \( k \) then Server indeed proposed \( k \) for use to Client.
Naiv handshake prot. Declaration

Symmetric encryption
- **type** key.
- **fun** senc(bitstring, key) : bitstring.
- **reduc for all** m: bitstring, k:key; sdec(senc(m,k), k) = m.

Assymmetric encryption
- **type** skey.
- **type** pkey.
- **fun** pk(skey) : pkey.
- **fun** aenc(bitstring, pkey) : bitstring.
- **reduc for all** m: bitstring, k : skey; adec(aenc(m, pk(k)),k) = m.

Digital signature
- **type** sskey.
- **type** spkey.
- **fun** spk(sskey) : spkey.
- **fun** sign(bitstring, sskey) : bitstring.
- **reduc for all** m: bitstring, k : sskey; getmess(sign(m, k)) = m.
- **reduc for all** m: bitstring, k : sskey; checksign(sign(m, k), spk(k)) = m.
Verifying weak secrecy in the Handshake protocol.

```plaintext
1  free c : channel .
2  free s : bitstring [private] .
3  query attacker(s) .
4
5  let clientC(pkC: pkey, skC : skey, pkS: spkey ) =
6     out(c, pkC) ;
7     in(c, x : bitstring);
8     let y = adec(x, skC) in
9     let (=pkS, k : key) = checksign(y, pkS) in
10    out(c, senc(s, k )).
11
12  let serverS(pkS: spkey, skS : sskey) =
13     in(c, pkX: pkey);
14     new k : key;
15     out(c, aenc(sign((pkS, k), skS), pkX));
16     in(c, x : bitstring);
17     let z = sdec(x, k) in 0.
18
19  process
20     new skC : skey;
21     new skS : sskey;
22     let pkC = pk(skC) in out(c, pkC);
23     let pkS = spk(skS) in out(c, pkS);
24     ( (! clientC(pkC, skC , pkS)) | (! serverS(pkS, skS)) )
```

Client subprocess (Process Macro)

Server subprocess (Process Macro)

Main process
Verifying authenticity in the Handshake protocol.

Client believes it has terminated a protocol run with parameters symm. key and its pub. key.

Server believes it has terminated a protocol run with Client using symm. key.

Client wants to be sure that when it completes the protocol it has indeed shared its secret with Server.

Not injective: Server’s msg can be replayed

Injective corresp.: Server believes that at the end of the prot. Client was indeed its partner.

One-to-one relation
Attack against the naiv handshake protocol

\[
\begin{align*}
A & \rightarrow S : pk_A \\
S & \rightarrow A : \{ \{ pk_s, k \} \}_{sk_s} p_k A \\
C & \rightarrow S : pk_C \\
A & \rightarrow C : \{ \{ pk_s, k \} \}_{sk_s} p_k C \\
C & \rightarrow S : \{ s \} _k 
\end{align*}
\]

- **Man-in-the-middle** (and Interleaving) attack
  - The *attacker starts a session with the server.*
  - Server send the session key to the attacker.
  - The attacker decrypts the received message with its public key, and then encrypts the result with \( pk_C \).
  - When Client initiates a session with Server, the attacker impersonates Server.
  - The attacker knows the session key \( k \), and can later use it to decrypt messages that Client sends to Server.
Observational equivalence - noninterf

• Intuitively, two processes P and Q are observationally equivalent, written $P \approx Q$, when an active adversary cannot distinguish P from Q.

• Can be used to prove strong secrecy of some value s.

• Capture the adversary’s ability to learn partial information about the secret.
  • A protocol provides strong secrecy of secret if the attacker cannot distinguish Protocol\{0/secret\} from Protocol\{1/secret\}, that is, Protocol\{0/secret\} $\approx$ Protocol\{1/secret\}.

The strong secrecy of values $x_1, \ldots, x_n$ is defined in ProVerif by:

```plaintext
/* free $x_1, \ldots, x_n [\text{private}]. */
1. noninterf $x_1, \ldots, x_n$ or
2. noninterf $x_1$ among ($N_1, \ldots, N_k$)

..., $x_n$ among ($M_1, \ldots, M_t$)
```

• When the process under consideration is P, this query is true if and only if $P\{M_1/x_1, \ldots, M_n/x_n\} \approx P\{N_1/x_1, \ldots, N_n/x_n\}$
Observation equivalent examples

1 \texttt{free} \ c : \text{channel}.

2 (* Types *)
3 \texttt{type} \ key.
4 \texttt{type} \ bitstring.

(* Shared key encryption, decryption *)
5 \texttt{fun} \ \text{senc}(\text{bitsring}, \text{key}) : \text{bitstring}
6 \texttt{reduc} \ \texttt{forall} \ x : \text{bitstring}, \ y : \text{key}; \ \text{sdec}(\text{senc}(x, y), y) = x.

7 (* The shared key *)
8 \texttt{free} \ k : \text{key [private]}.

9 (* Query *)
10 \texttt{free} \ secret : \text{bitstring [private]}.

11 (* Prove strong secrecy of secret *)
12 \texttt{noninterf} \ secret.

(* Main process *)
13 \texttt{process}
14 (\texttt{!out}(c, \text{senc}(secret, k))) | (\texttt{!in}(c, x : \text{bitstring}); \ \text{let} \ s = \text{sdec}(x, k) \ \text{in} \ 0)

\textbf{Prove that the attacker cannot obtain partial information (guess) of secret.}
Observation equivalent examples cont’d

1. \textbf{free} \(c : \text{channel}\).

2. (* Types *)
3. \textbf{type} \(\text{key}\)
4. \textbf{type} \(\text{bitstring}\)

   (* Hash *)
5. \textbf{fun} \(\text{hash}(:\text{bitstring}) : \text{bitstring}\)

6. (* The shared key *)
7. \textbf{free} \(x, n : \text{bitstring}[\text{private}]\).

8. (* Prove strong secrecy of secret *)
9. \textbf{noninterf} \(x\) among \((n, \text{hash}(n))\).

   (* Main process *)
10. \textbf{process} \(\text{out}(c, x)\)

• Prove that the attacker cannot distinguish \(n\) from \(\text{hash}(n)\).
Observation equivalent examples cont’d

1 \textbf{free} \, c : \text{channel} .

• Voters encrypt their votes with the administrator’s public key.

2 (* Types *)
3 \textbf{type} \, skey.
4 \textbf{type} \, pkey.

• Prove that the attacker cannot guess voter’s vote.

(* Hash *)
5 \textbf{fun} \, \text{pk}(\text{skey}) : \text{pkey}.
6 \textbf{fun} \, \text{aenc}(\text{bitstring}, \text{pkey}) : \text{bitstring}.
7 \textbf{reduc forall} \, m : \text{bitstring}, \, k : \text{skey}; \, \text{adec}(\text{adec}(m, \text{pk}(k)), \, k) \, = \, m .

(* The shared key *)
8 \textbf{free} \, \text{vote} : \text{bitstring} [\text{private}].
9 \textbf{weaksecret} \, \text{vote}.

(* Process macros, that is, sub-processes. *)
10 \textbf{let} \, \text{Voter}(\text{pkA} : \text{pkey}) = \text{out}(c, \text{aenc}(\text{vote}, \text{pkA})).
11 \textbf{let} \, \text{Administrator}(\text{pkA} : \text{skey}) = \text{in}(c, \, x : \text{bitstring}); \, \textbf{let} \, \nu' = \text{adec}(x, \, \text{skA}) \, \textbf{in} \, 0.

(* Main process *)
12 \textbf{process}
13 \textbf{new} \, \text{skA} : \text{skey};
14 \textbf{let} \, \text{skA} = \text{pk}(\text{skA}) \, \textbf{in}
15 \text{out}(c, \, \text{pkA});
16 ! ( \, \text{Voter}(\text{pkA}) \, \mid \, \text{Administrator}(\text{skA}) \, )
Observational equivalence - choice

• Prove two processes $P$ and $Q$ are observationally equivalent.
• $P$ and $Q$ have the same structure but differ only in the choice of terms.
• ProVerif provides encoding both $P$ and $Q$ using a „biprocess“.

Prove $P(M) \approx P(N)$ for some different $M, N$ then

$P(\text{choice}[M, N])$

• $P$ is the voter process, $skA$ and $skB$ are secret key of the voters.
• $vote1$ and $vote2$ are votes made by voters.
• **Goal: to prove privacy property.**
  • The attacker cannot distinguish the situation when $A$ makes $vote1$ and $B$ votes $vote2$,
  • and the situation when $A$ makes $vote2$ and $B$ votes $vote2$.

To prove $P(skA, vote1) \parallel P(skB, vote2) \approx P(skA, vote2) \parallel P(skB, vote1)$:

$P(skA, choice[\text{vote1}, vote2]) \parallel P(skB, choice[\text{vote2}, vote1])$
Summary

• Relevant webpages
  Official page: http://www.proverif.ens.fr/
  Online demos: http://server1.dsuresearch.org/ProVerif/demo/index2.php/
  GUI Editor: http://sourceforge.net/projects/proverifeditor/

• The full manual of ProVerif (by Bruno Blanchet) will be uploaded to the course’s page.

• Instructions:
  1. Download ProVerif’s newest version: 1.86pl3 (archive file)
  2. Extract the archive file and install the program is shown in slides 16-17.
  3. Try examples that can be found in the archive file.
  • The best way to understand ProVerif is to play with example files.
  • Example files can be found in the directory docs: docs/<filename>.pv
  • Several examples with detailed comments: Needham-Schroeder, Diffie-Hellman, etc.

If do you have any question please contact me via mail:
thong[@]crysys[.]hu (* remove [ and ] *)
Kérdések?

**KÖSZÖNÖM A FIGYELMET!**

Híradástechnikai Tanszék

Ta Vinh Thong
Phd student
BME Híradástechnikai Tanszék
Lab of Cryptography and System Security (CrySyS)
thong at crysys dot hu

Formal analysis of security protocols

Ta Vinh Thong, Híradástechnikai Tanszék
Budapesti Műszaki és Gazdaságtudományi Egyetem