FORMAL ANALYSIS OF SECURITY PROTOCOLS

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
- Applied pi-calculus
- 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).
Computer aided formal methods

- Model checking
  - System to be analysed is modelled in finite automaton
    - Automaton states are intermediate states of system run.
    - Explore all possible executions during verification.
  - Property to be proven is given in temporal logic (LTL, CTL)
    - E.g., $F$ property (reachability), $G$ property (invariance).
  - Well-known tools: SPIN (partial order reduction), SAL (symbolic), UPPAAL (handle time), PRISM (probabilistic).

😊 Fully automated (press a button), handle large state set, sound.
😊 Scalability problem: Handle finite state set.
😊 Not complete: If stop without attack, often it doesn’t mean anything.
😊 Not generalizable: If system is secure for 2 entities we still don’t know if it is secure for N.
Computer aided formal methods cont’d

- Theorem proving
  - Based on formal logic (first order, higher order).
  - System to be analysed is modelled as a finite set of logic rules and facts.
  - Property to be proven is given as a fact.
  - Axioms = system logic rules + additional inference logic rules (e.g. attacker rules).
  - Deduction proof is refutation based
    - Prove that from the Axioms the **negated property**
      - cannot be derived: property always hold.
      - can be derived: found a counterexample.
  - Well-known tools: Otter-prover, PVS, Prover9, Isabelle.

😊 Can be complete, scalability, systematic, sound.
رياض Not fully automated: required user interaction, user guide.
رياض Only expert can use.
Verifying security protocols

- What we need to model (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.
General concept

Informal protocol description

Informal property description

Abstractions

Formal model of protocol

Formal model of security property

Automatic verification engine

OK

Counterexamples

Informal protocol description

Informal property description

Abstractions

Formal model of protocol

Formal model of security property

Automatic verification engine

OK

Counterexamples
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 a 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)
  - **Forward secrecy**: Even if some participants get corrupted (so their secret keys are leaked to the adversary), the secrets exchanged in sessions that took place before the corruption are preserved.

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

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

- **Anonimity**: Preserve the privacy of users.
Proverif verification 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.
Specification language – syntax

\[ M, N ::= \quad \text{terms} \quad \# \text{elements of messages, comm. channel}\#
\]
\[ a, b, c, \ldots, m, n, \ldots, s \quad \text{name} \quad \# \text{channel, nonce, constant}\#
\]
\[ x, y, z \quad \text{variable} \quad \# \text{represents any terms}\#
\]
\[ f(M_1, \ldots, M_k) \quad \text{constructor function} \quad \# \text{model cryptographic primitives}\#
\]

\[ P, Q, R ::= \quad \text{processes} \quad \# \text{Internal behaviour}\#
\]
\[ 0 \quad \text{null process} \quad \# \text{inactive process, stop running}\#
\]
\[ P | Q \quad \text{parallel composition} \quad \# \text{concurrent execution of P and Q}\#
\]
\[ !P \quad \text{replication} \quad \# P | P | P | \ldots \#
\]
\[ \text{new n; P} \quad \text{name restriction} \quad \# \text{create fresh n: nonce, keys, private channel}\#
\]
\[ \text{if } M = N \text{ then } P \text{ else } Q \quad \text{condition} \#
\]
\[ \text{let } x = g(M_1, \ldots, M_n) \text{ in } P \text{ else } Q \quad \text{destructor appl.} \quad \# \text{inverse of constructor function}\#
\]
\[ \text{let } x = M \text{ in } P \quad \text{local binding} \quad \# \text{bind M to } x \text{ in P}\#
\]
\[ \text{in}(c, x).P \quad \text{message input} \quad \# \text{receive some term on chan. c, bind it to } x \text{ in P}\#
\]
\[ \text{out}(c, M).P \quad \text{message output} \quad \# \text{send term M on chan. c, bind it to } x \text{ in P}\#\]
Specification language – semantics

Reduction Relation (\(\rightarrow\))

\[
out(c,M).P | in(c,x).Q \rightarrow P | Q\{M/x\}
\]

if \(M = M\) then \(P\) else \(Q\) \(\rightarrow\) \(P\)

if \(M = N\) then \(P\) else \(Q\) \(\rightarrow\) \(Q\), if \(M \neq N\)

let \(x = M\) in \(P\) \(\rightarrow\) \(P\{M/x\}\)

let \(x = g(M_1,\ldots,M_k)\) in \(P\) else \(Q\) \(\rightarrow\) \(P\{M/x\}\), if \(g(M_1,\ldots,M_k) = M\)

let \(x = g(M_1,\ldots,M_k)\) in \(P\) else \(Q\) \(\rightarrow\) \(Q\), if \(g(M_1,\ldots,M_k)\) is undefined

- \(\text{def}(g)\) is a set of all defined equations.
- \(g(N_1,\ldots,N_k)\) is defined iff there exists an equation \(g(M_1,\ldots,M_k) = M\) in \(\text{def}(g)\) and a substitution \(\sigma\) such that \(M_i \sigma = N_i\), \(i \in \{1,\ldots,k\}\). This case \(g(N_1,\ldots,N_k) = M\sigma\).
- E.g. Let equation \(\text{dec}(\text{enc}(x,y),y) = y\) in \(\text{def}(g)\). Then, \(\text{dec}(\text{enc}(m,k),k)\) is defined and \(\text{dec}(\text{enc}(m,k),k) = m\), where \(\sigma = \{M/x, k/y\}\).
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)

• Semantics

Protocol = new m; (out(c, m).in(c, y).0 | in(c, x).out(c, hash(x)).0) →
new m; (in(c, y).0 | out(c, hash(x)) {m/x}).0 →
new m; (0{hash(m)/y}0) → new m; (0|0) → 0
Protocol examples – 2

- 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)

- Semantics

Protocol = new m;(Alice|Bob) → new m;(let (x_1, x_2 ) = (n, senc(hash(n),k)) in
  let y = sdec(senc(hash(n),k),k) in if y = hash(n) then out(c,n)). →
  if hash(n) = hash(n) then out(c,n)). → new m;(out(c,n)).
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.

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

- 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 \( x \).

\[ \left( \ast \text{ Strong secrecy of } M : \text{Observational equivalent } \ast \right) \]

\[ Protocol(M) \approx Protocol(M') \]
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: 1.85.
  - 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
  - Create a directory `proverif1.85` in the current directory.
    - Using GNU tar: 1) `tar -xzvf proverif1.85.tar.gz` ; 2) `tar -xzvf proverifdoc1.85.tar.gz`
    - Using tar: 1) `gunzip proverif1.85.tar.gz`; 2) `tar -xf proverif1.85.tar`; 3) `gunzip proverifdoc1.85.tar.gz`; 4) `tar -xf proverifdoc1.85.tar`.
  - Build the ProVerif
    - `cd proverif1.85`
    - `./build`
ProVerif installation, execution

Installation in Windows

- From binary
  - Decomposes: `proverifbsd1.85.tar.gz` and `proverifdoc1.85.tar.gz` archives in the same directory.
- From source
  - Requires OCaml installation, OCaml bytecode compiler.
  - Decomposes: `proverif1.85.tar.gz` and `proverifdoc1.85.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)

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

**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 \( M = N \).
  - \( in(=M);P \) waits a term that matches pattern \( M \), then behaves as \( P \).
ProVerif’s input file – 1

- Declaration, process definition, queries

```
< decl > ::= declaration
  type t
  free name : t
  free name : t [private]
  const c : t
  table d(t₁,...,tₙ)
  fun f(t₁,...,tₙ) : t
  fun f(t₁,...,tₙ) : t [private]
  reduc forall x₁:t₁,...,xₙ:tₙ; g(M₁,...,Mₖ) = M
  reduc forall x₁:t₁,...,xₙ:tₙ; g(M₁,...,Mₖ) = M [private]
```

- Declaration
  - user defined type
  - public free names
  - private free names
  - constant c of type t
  - table d takes records of type t₁,...,tₙ
  - public constructor function
  - private constructor function
  - public destructor application
  - private destructor application
ProVerif’s input file – 2

\[ \langle \text{procmacro} \rangle ::= \]

\[ \text{let } R(x_1,\ldots,x_n) = P. \]

\[ \text{sub-process definition} \]

\[ \text{define a process with the name } R. \]

\[ \langle \text{mainproc} \rangle ::= \]

\[ \text{process } P \]

\[ \text{main process definition} \]

\[ \langle \text{query} \rangle ::= \]

\[ \text{query attacker}(M). \]

\[ \text{define properties to be verified.} \]

\[ \text{want to verify weak secrecy of } M. \]

\[ \text{query } \text{event}(M) \implies \text{event}(N). \]

\[ \text{in any trace: event } N \text{ is always before event } M. \]

\[ \text{query } \text{inj} - \text{event}:M \implies \text{inj} - \text{event}:N. \]

\[ \text{in any trace: at least as many } N \text{-events as } M \text{-events.} \]

\[ \text{noninterf } x \]

\[ \text{want to verify strong secrecy of } x. \]

\[ \text{noninterf } x \text{ among } \{M_1,\ldots,M_k\} \]

\[ \text{want to verify strong secrecy of } x \text{ taken value from a set.} \]

The structure of the whole ProVerif source file is

\[ \langle \text{decl} \rangle^* \langle \text{query} \rangle^* \langle \text{procmacro} \rangle^* \langle \text{mainproc} \rangle \]
Types and Names

| type t              | user defined type.                  |
| free name:t        | free names.                         |
| free name:t [private] | private free names.                |

• User defined type can be anything except for pre-defined, built-in types.
  • 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>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fun f(t₁,...,tₙ) : t</code></td>
<td>Public constructor function</td>
</tr>
<tr>
<td><code>fun f(t₁,...,tₙ) : t [private]</code></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 `t₁,...,tn` 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.

\[\begin{align*}
\text{reduc} & \quad \forall x_1 : t_1, \ldots, x_n : t_n; \quad g(M_1, \ldots, M_k) = M \\
\text{reduc} & \quad \forall x_1 : t_1, \ldots, x_n : t_n; \quad g(M_1, \ldots, M_k) = M [\text{private}] \\
\end{align*}\]
Constructor and Destructor examples

Symmetric encryption (User defined)

\[
\text{type bitstring, key.} \\
\text{fun senc(bitstring, key): bitstring} \\
\text{reduc forall m: bitstring, k: key; sdec (senc (m, k), k) = m.}
\]

Asymmetric encryption (User defined)

\[
\text{type bitstring, skey, pkey.} \\
\text{fun pk(skey): pkey} \\
\text{fun penc(bitstring, pkey): bitstring} \\
\text{reduc forall m: bitstring, k: skey; pdec (penc (m, pk(k)), k) = m.}
\]
Constructor and Destructor examples

Digital signatures (User defined)

\[
\text{type bitstring, sskey, spkey.}
\]
\[
\text{fun spk(sskey):spkey}
\]
\[
\text{fun sign(bitstring, sskey):bitstring}
\]
\[
\text{reduc forall m: bitstring, k : sskey; checksign (sign(m, spk(k) ), k) = m.}
\]

One-way functions (User defined)

\[
\text{type bitstring, key.}
\]
\[
\text{fun hash(bitstring):bitstring.}
\]
\[
\text{fun mac(bitstring, key).}
\]

Tuples (built-in)

\[
\text{fun (t_1, ..., t_k): bitstring}
\]
\[
\text{reduc forall x_1:t_1, ..., x_k:t_k; ith((M_1, ..., M_k )) = M_i}
\]
Tables and operations

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 \( \text{insert } d(M_1, \ldots, M_n); P \) inserts the record \( M_1, \ldots, M_n \) into the table \( d \) and then executes \( P \).

Process \( \text{get } d(T_1, \ldots, T_n) \text{ in } P \) 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.

\[
\begin{align*}
\text{table } d(t_1, \ldots, t_n) & \quad (* \text{ declation } *) \\
\text{insert } d(M_1, \ldots, M_n); P \\
\text{get } d(T_1, \ldots, T_n) \text{ in } P \\
\text{get } d(T_1, \ldots, T_n) \text{ suchthat } C \text{ in } P
\end{align*}
\]
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. *x1,…xn* of type *t1,…,tn* are variables of *P*.
- *R(M1, …, Mn)* will then expand to *P* with *M1* substituted for *x1*, …, *Mn* substituted for *xn*.

\[
\text{let } R(x_1 : t_1, ..., x_n : t_n) = P
\]
Hello word script

1. (* hello.pv: Hello World Script *)
2. free c : channel.
3. free $K_1$ : bitstring[private].
4. free $K_2$ : bitstring[private].
5. query attacker($K_1$).
6. query attacker($K_2$).
7. process
8. out(c,$K_2$);
9. 0
10. The verification algorithm attempts to prove that the states in which the keys are obtained by the attacker are unreachable.
Hello world with process macros

1. (* Hello World Script with Macros *)
2. free c : channel.
3. free $K_1$ : bitstring[private].
4. free $K_2$ : bitstring[private].
5. query attacker($K_1$).
6. query attacker($K_2$).
7.
8. let $R(x : \text{bitstring}) = \text{out}(c, x);0.$
9. let $R'(y : \text{bitstring}) = 0.$
10. process $R(K_1) | R'(K_2)$
**ProVerif output**

- Process:
  - [Process]

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

[Attack derivation]

- set traceDisplay = long.

[Attack trace]

- RESULT [Query][result].

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.

Reconstruct and return the detected attack.

Finally, ProVerif reports if the Query was satisfied.
Bracketing, precedence

- $P \mid Q$ binds most closely, $P \mid Q$ is equal to $!(P \mid Q)$
- if $C$ then $Q$ else $P$
- let $x=M$ in $P$ else $Q$
- unary processes bind least closely.
- E.g.,

1. new $n : t$ ; out($c$, $n$) $\mid$ new $n : t$; in ($c$, $x : t$); 0 $\mid$ if $x = n$ then 0 $\mid$ out ($c$, $n$), is bracketed as
   new $n : t$ ; (out ($c$, $n$) $\mid$ new $n : t$ ; (in($c$, $x : t$); 0 $\mid$ if $x = n$ then (0 $\mid$ out ($c$, $n$))))

2. if $M = M'$ then if $N = N'$ then $P$ else $Q$, is bracketed as
   if $M = M'$ then (if $N = N'$ then $P$ else $Q$)

3. let $x = M$ in let $y = N$ then $P$ else $Q$, is bracketed as
   let $x = M$ in (let $y = N$ then $P$ else $Q$)
Running result of the hello world script

Process
\{1\} out(c, K_2)
-- Query not attacker(K_1[])
Completing...
Starting query not attacker(K_1[])
RESULT not attacker(K_1[]) is true.
-- Query not attacker(K_2[])
Completing...
Starting query not attacker(K_2[])
goal reachable : attacker(K_2[]).

The message K_2[] may be sent to the attacker at output{1}.
attacker(K_2[]).

set traceDisplay = long.
out(c, K_2) at {1}
The attacker has the message K_2.
A trace has been found.
RESULT not attacker(K_2[]) is false

Try to prove not attacker(K_1[])
attacker(K_1[]) is false
The attacker can obtain K_2.
Attack trace is returned.
Security properties, Queries

• Reachability and weak secrecy
  • which term will be obtained by the attacker during protocol runs.
  • E.g., can the attacker obtain term M? \textbf{query attacker}(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 $e(M_1,\ldots,M_n)$; P …. 
  • declaration: \textit{event} $e(t_1,\ldots,t_n)$. Where $t_1,\ldots,t_n$ are types of arguments.
  • Query: \textbf{query} $x_1:t_1,\ldots,x_n:t_n$; event $e(M_1,\ldots,M_j)$ $\Rightarrow$ event $e'$ ($N_1,\ldots,N_i$)
    • for each occurrence of $e(M_1,\ldots,M_j)$, event $e'$ ($N_1,\ldots,N_i$) 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: \textbf{query} $x_1:t_1,\ldots,x_n:t_n$; inj-event $e(M_1,\ldots,M_j)$ $\Rightarrow$ inj-event $e'$ ($N_1,\ldots,N_i$)
Queries full syntax

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

\[
H ::= \\
F \\
H & H \\
H \lor H \\
(F \implies H)
\]

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

\[
\text{query} \\
\text{fact} \\
\text{correspondence}
\]

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

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

1. (* hello _ext.pv: Extended Hello World Script *)
2. free c: channel.
3. free K_1: bitstring[private].
4. free K_2: bitstring[private].
5. event recvK_1.
6. event recvK_2.
5. query event(recvK_1) => event(recvK_2)
7.
8. process
9. out(c, K_2);
10. in(c, x : bitstring);
11. if x = K_1 then
12. event recvK_1; event recvK_2
13. else event recvK_2

Event receiving key K1, K2.
Prove that for all execution of the protocol recvK2 occurs before recvK1.
Event of receiving key K1, K2.
Naiv handshake protocol

\begin{align*}
C & \to S : pk_C \\
S & \to C : \{ \{ pk_S, k \}^s \}^p k_C \\
C & \to S : \{ s \}^k
\end{align*}

• 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.
Attack against the naive handshake protocol

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

- Man-in-the-middle (and Interleaving) attack
  - The attacker starts a session with the server.
  - Server sends 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.
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 Server using symm. key and its pub. key.

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

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

1. free c : channel.
2. free s : bitstring [private].
3. query attacker(s).

4. event acceptsClient(key).
5. event acceptsServer(key, pkey).
6. event termClient(key, pkey).
7. event termServer(key).

8. query x:key, y:pkey; event(termClient(x,y)) ==> event(acceptsServer(x,y)).
9. query x:key; inj-event(termServer(x)) ==> inj-event(acceptsClient(x)).

11. let clientC(pkC: pkey, skC: skey, pkS: spkey ) =
12.  out(c, pkC) ;
13.  in(c, x : bitstring);
14.  let y = adec(x, skC) in
15.  let (=pkS, k : key) = checksign(y, pkS) in
16.  event acceptsClient(k).
17.  out(c, senc(s, k )).
18.  event termClient(k, pkC).

19. let serverS(pkS: spkey, skS : sskey) =
20.  in(c, pkX: pkey);
21.  new k : key;
22.  event acceptsServer(k, pkX).
23.  out(c, aenc(sign((pkS, k), skS), pkX));
24.  in(c, x : bitstring);
25.  let z = sdec(x, k) in if  pkX = pkC then event  termServer(k).

Client accepted to run the protocol with Server with the symm. key.
Server accepted to run the protocol with Client with the symm. key and client’s public key.

Injective corresp.: Server believes that at the end of the prot. Client was indeed its partner.
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 two processes conform each other.
  • E.g, \( \text{IdealOperation} \approx \text{RealOperation} \).
• Can be used to prove strong secrecy of some value \( s \).
• The value of \( s \) should not affect the observable behavior of the protocol.
• Capture the adversary’s ability to learn partial information about the secret.
  • not attacker(secret) is true, that is, secret is not obtained by the attacker.
  • however, if the attacker knows secret can be taken from the set \{0,1\}.
• 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 \( x_1, \ldots, x_n \) [private]. */
1. noninterf \( x_1, \ldots, x_n \) or
2. noninterf \( x_i \) among \( (N_1, \ldots, N_k) \)
   ...
   \( x_n \) among \( (M_1, \ldots, M_l) \)
```

• When the process under consideration is \( P \), this query is true if and only if
\[
P\{M_1/x_1, \ldots, M_n/x_n\} \approx P\{M'_1/x_1, \ldots, M'_n/x_n\}
\]
Observation equivalent examples

- Prove that the attacker cannot obtain partial information (guess) of secret.

```plaintext
1  free c : channel.
2  (* Types *)
3  type key.
4  type bitstring.

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

7  (* The shared key *)
8  free k : key [private].

9  (* Query *)
10  free secret : bitstring [private].

11  (* Prove strong secrecy of secret *)
12  noninterf secret.

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

© Ta Vinh Thong, Híradástechnikai Tanszék
Budapesti Műszaki és Gazdaságtudományi Egyetem
Observation equivalent examples cont’d

1. free c : channel .

2. (* Types *)
3. type key
4. type bitstring

(* Hash *)
5. fun hash(bitsring): bitstring

(* The shared key *)
6. free x, n : bitstring [private].

(* Prove strong secrecy of secret *)
7. noninterf x among ( n, hash(n) ).

(* Main process *)
8. process out(c, x)

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

1  free c : channel.

2  (* Types *)
3  type skey.
4  type pkey.

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

8  (* The shared key *)
9  free vote : bitstring [private].
10  weaksecret vote.

(* Process macros, that is, sub-processes. *)
11  let Voter(pkA : pkey) = out(c, aenc(vote, pkA)).
12  let Administrator(pkA : skey) = in(c, x : bitstring); let v’ = adec(x, skA) in 0.

(* Main process *)
13  process
14     new skA : skey;
15     let skA = pk(skA) in
16     out(c, pkA);
17     ! ( Voter(pkA) | Administrator(skA) )

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

• Prove that the attacker cannot guess voter’s vote.
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** $P(skA,vote1) \mid P(skB,vote2) \approx P(skA,vote2) \mid P(skB,vote1) :$

$$P(skA,\text{choice}[vote1,vote2]) \mid P(skB,\text{choice}[vote2,vote1])$$
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: 1.85 (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[@]crysyst.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