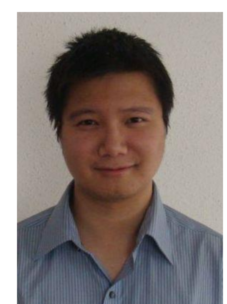
VERIFYING SECURITY PROTOCOLS USING THE PROVERIF TOOL

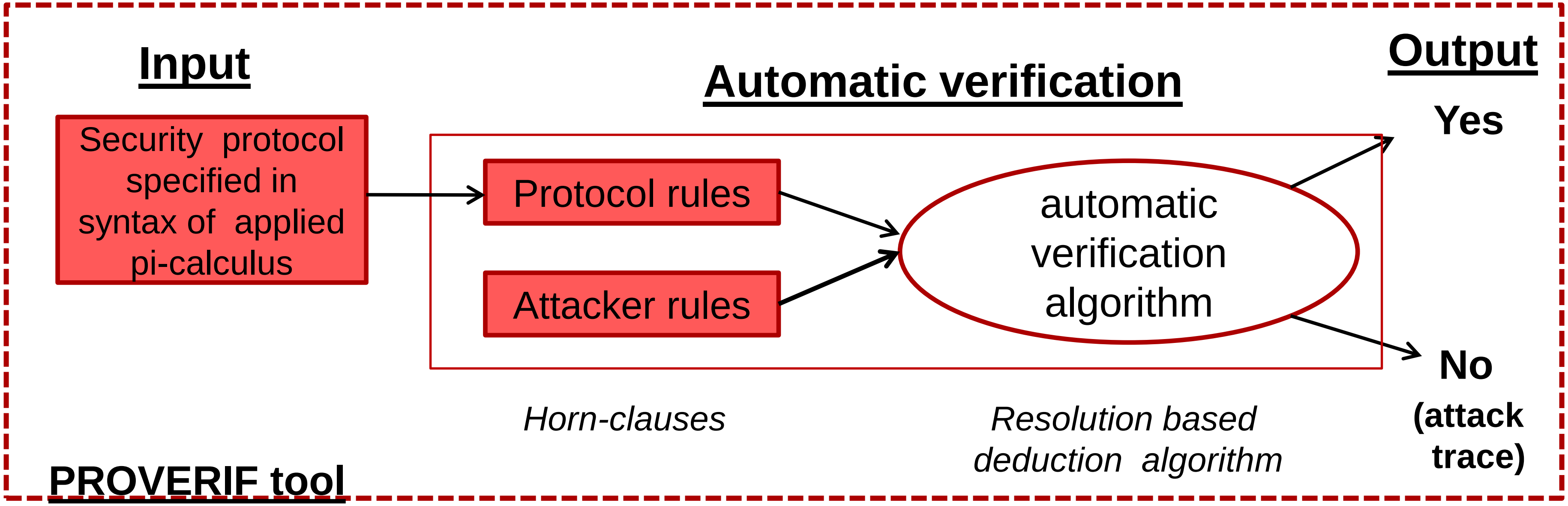
Security protocols (bmevihim132)

Ta Vinh Thong
BME Hálózati Rendszerek és Szolgáltatások Tanszék
Lab of Cryptography and System Security (CrySyS)
thong at crysys dot hu



2014. március 24., Budapest

Proverif verification tool



- Bruno Blanchet, ~ 2004.
- A kind of theorem prover (based on logic), but fully automated.
- Security protocols are specified in a simplified form of applied pi calculus.
- Can verify strong secrecy, weak secrecy, forward secrecy, authenticity, anonimity.

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.

Typed applied pi-calculus

$M, N ::=$	terms
a, b, c, k, m, n, s	names
x, y, z	variables
$(M_1, \ldots, M_k)$	tuple
$h(M_1, \ldots, M_k)$	constructor/destructor application
$C ::=$	conditions
$M = N$	term equality
$M <> N$	term inequality
M	term (of type bool)
$C \ \&\& \ C$	conjunction
$C \ \|\| \ C$	disjunction
$\mathbf{not}(C)$	negation
$P, Q ::=$	processes
0	null process
$P \mid Q$	parallel composition
$!P$	replication
$\mathbf{new}\ n : t;\ P$	name restriction
$\mathbf{in}(M, x : t);\ P$	message input
$\mathbf{out}(M, N);\ P$	message output
$\mathbf{if}\ \ C\ \mathbf{then}\ P\ \mathbf{else}\ Q$	conditional
$\mathbf{let}\ \ x = M\ \mathbf{in}\ P\ \mathbf{else}\ Q$	term evaluation
$R(M_1, \ldots, M_k)$	macro usage

Pattern matching

$T ::=$	patterns
$x : t$	typed variable
x	variable without explicit type
$(T_1, ..., T_n)$	tuple
$=M$	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 *(T1,…,Tn)* matches tuples *(M1,…,Mn)* where each component *Mi* is recursively matched with *Ti.*
- The pattern =*M* matches terms *N* where *M* = *N*.
  - *in(=M);P waits a term that matches pattern M, then behaves as P.*

Input e.g.,: in(c, x:t); Q can be written as in(c, =M); Q

ProVerif's input file – 1

- **Declaration, process definition, queries**

$< decl >::=$	*declaration*
$type\ t$	user defined type
$free\ \ name : t$	public free names
$free\ \ name : t\ [private]$	private free names
$const\ \ c{:}t$	constant c of type t
$table\ \ d(t_1,...,t_n)$	table d takes records of type $t_1,..,t_n$
$fun\ \ f(t_1,...,t_n) : t$	public constructor function
$fun\ \ f(t_1,...,t_n) : t\ \ [private]$	private constructor function
$reduc\ \ forall\ x_1{:}t_1,..., x_n{:}t_n;\ g(M_1,..., M_k) = M$	public destructor application
$reduc\ \ forall\ x_1{:}t_1,..., x_n{:}t_n;\ g(M_1,..., M_k) = M\ [private]$	private destructor application

ProVerif's input file – 2

$< procmacro > ::=$ *sub-process definition*

$let\ R(x_1, ..., x_n) = P.$ define a process with the name R.

$< mainproc > ::=$ *main process definition*

$process\ P$

$< query > ::=$ *define properties to be verified.*

$query\ attacker(M).$ 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.

$noninterf\ x\ among\ \{M_1, ..., M_k\}$ 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

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.
  - 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

$fun\ \ f(t_1,...,t_n):t$ public constructor function

$fun\ \ f(t_1,...,t_n):t\ [private]$ private constructor function

- 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

$$reduc\ \ forall\ \ x_1{:}t_1,\ldots,x_n{:}t_n;\ g(M_1,\ldots,M_k)=M$$ public destructor application

$$reduc\ \ forall\ \ x_1{:}t_1,\ldots,x_n{:}t_n;\ g(M_1,\ldots,M_k)=M\ [private]$$ private destructor application

- Destructor applications:
  - *g* is a function of arity *k*.
  - *t1,…, tn* are types of the variables *x1, …, xn*.
  - *x1, …, xn* are variables in terms *M1, …, Mk.*
  - destructor applications are available to all participants (honest and attacker).
  - private destructor applications are available to only honest participants.

Constructor and Destructor examples

Symmetric encryption (User defined)

type bitstring, key.

fun senc(bitstring,key):bitstring

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

Asymmetric encryption (User defined)

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)

$type\ bitstring, sskey, spkey.$

$fun\ spk(sskey){:}spkey$

$fun\ sign(bitstring, sskey){:}bitstring$

$reduc\ forall\ m{:}\ bitstring, k : sskey; checksign\ (sign(m, spk(k)), k) = m.$

One-way functions (User defined)

$type\ \ bitstring, key.$

$fun\ \ hash(bitstring) : bitstring.$

$fun\ \ mac(bitstring, key).$

Tuples (bult-in)

$fun\ \ (t_1,...,t_k) : bitstring$

$reduc\ \ forall\ x_1{:}t_1,...,x_k{:}t_k;\ ith((M_1,...,M_k)) = M_i$

Tables

Tables and operations

$$table\ \ d(t_1,...,t_n) \qquad (*\ \ declation*)$$

$$insert\ d(M_1,...,M_n); P$$

$$get\ d(T_1,...,T_m)\ \ in\ \ P$$

$$get\ d(T_1,...,T_m)\ \ suchthat\ C\ \ in\ \ P$$

- *d* is the name of the table which takes records of type *t1, . . . , tn.*
- Processes may populate and access tables, but deletion is forbidden.
- Process *insert d(M1, . . . ,Mn); P* inserts the record *M1, . . . ,Mn* into the table *d* and then executes *P.*
- Process *get d(T1, . . . , Tn)* in *P* attempts to retrieve a record in accordance with patterns *T1, . . . , Tn.*
  - 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 *x1,…xn* of type *t1,…,tn* are variable of *P.*
- *R(M1, . . . ,Mn)* will then expand to *P* with *M1* substituted for *x1, . . . , Mn* substituted for *xn*.

$$let \quad R(x_1 : t_1, ..., x_n : t_n) = P$$

Bracketing, precedence

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

1. new n : t ; out(c , n) | new n : t; in (c, x : t); 0 | if x = n then 0 | out (c, n), is braketed as
   new n : t ; (out (c, n) | new n : t; (in(c, x : t); 0 | if x = n then (0 | out (c, n))))

2. if M = M′ then if N = N′ then P else Q, is braketed 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 braketed as
   let x = M in (let y = N then P else Q)

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.
8. process
9. out(c, K_2);
10. 0

Public channel

Secret keys

Can Dolev-Yao attacker(s) obtain keys?

Main process

Send key K2 on channel c

Termination of process. Can be omitted.

- 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

Hello world with process macros

```
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)
```

Macros

Main process

ProVerif output

Process :

[Process]

The specified protocol. Each row is labelled with {n}, where n is an integer.

- - Query [Query]

Completing...

Starting query [Query]

RESULT [un]reachable : Goal.

Verifying the property given in [Query] by user.

[Attack derivation]

Reconstruct and return the detected attack

set traceDisplay = long.

[Attack trace]

RESULT [Query] [result].

Finally, ProVerif reports if the Query was satisfied.

- Internally, ProVerif attempts to prove that *a state in which a property is violated is unreachable*;
- ProVerif shows the (un)reachability of some [Goal].

Running result of the hello world script

Process

$\{1\}\,\text{out}(c, K_2)$

-- 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(K1[ ])**

attacker(K1[ ]) is false

The attacker can obtain K2.

Attack trace is returned.

Queries full syntax

$q ::=$	query
F	fact
$F \texttt{==>} H$	correspondence
$H ::=$	hypothesis
F	fact
$H \, \&\& \, H$	conjunction
$H \, \vert\vert \, H$	disjunction
$(F \texttt{==>} H)$	nested correspondence
$F ::=$	fact
$\mathrm{attacker}(M)$	the adversary has M (in any phase)
$\mathrm{attacker}(M)$ **phase** n	the adversary has M in phase n
$\mathrm{mess}(N, M)$	M is sent on channel N (in the last phase)
$\mathrm{mess}(N, M)$ **phase** n	M is sent on channel N in phase n
$\mathbf{event}(e(M_1, \ldots, M_n))$	non-injective event
$\mathbf{inj-event}(e(M_1, \ldots, M_n))$	injective event
$M{=}N$	equality
$M{<>}N$	inequality

Security properties, Queries

- Reachability and weak secrecy
  - which term will be ontained 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 occurence of **e(M1,…,Mj)**, event **e' (N1,…,Ni)** has occured 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 transation started by the client.
  - Query: **query x1:t1,…, xn:tn; inj-event e(M1,…,Mj) ==> inj-event e' (N1,…,Ni)**

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.

```
free c : channel .
free s : bitstring [private] .
query attacker(s) .

let clientC(pkC: pkey, skC : skey, pkS: spkey ) =
out(c, pkC) ;
in(c, x : bitstring);
let y = adec(x, skC) in
let (=pkS, k : key) = checksign(y, pkS) in
out(c, senc(s, k )).

let serverS(pkS: spkey, skS : sskey) =
in(c, pkX: pkey);
new k : key;
out(c, aenc(sign((pkS, k), skS), pkX));
in(c, x : bitstring);
let z = sdec(x, k) in 0.

process
new skC : skey;
new skS : sskey;
let pkC = pk(skC) in out(c, pkC);
let pkS = spk(skS) in out(c, pkS);
( (! 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*.

Client accepted to run the protocol with Server with the *symm. key*

Server accepted to run the protocol with **a client** with the *symm. key* and *client's public key*.

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

```
free c : channel.
free s : bitstring [private].
query attacker(s).

event  acceptsClient(key).
event  acceptsServer(key, pkey).
event  termClient(key, pkey).
event  termServer(key).

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

let clientC(pkC: pkey, skC: skey, pkS: spkey ) =
out(c, pkC) ;
in(c, x : bitstring);
let y = adec(x, skC) in
let (=pkS, k : key) = checksign(y, pkS) in
event  acceptsClient(k).
out(c, senc(s, k )).
event  termClient(k, pkC).

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

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

$$A \rightarrow S : pk_A$$

$$S \rightarrow A : \{\{pk_S, k\}_{sk_S}\}_{pk_A}$$

$$C \rightarrow S : pk_C$$

$$A \rightarrow C : \{\{pk_S, k\}_{sk_S}\}_{pk_C}$$

$$C \rightarrow S : \{s\}_k$$

- **Man-in-the-middle** (and Interleaving) attack
  - The *attacker starts a session with the server*.
  - Server send the session key to the attacker.
  - The attacker decrypts the received message with its public key, and then encrypts the result with pkC.
  - 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 ≈ 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} ≈ Protocol{1/secret}.

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

$$/*\ \text{free}\ x_1, \dots, x_n\ [\mathit{private}].*/$$

$$1.\ \text{noninterf}\ x_1, \dots, x_n \quad \text{or}$$

$$2.\ \text{noninterf}\ x_1\ \text{among}\ (N_1, \dots, N_k)$$

$$\dots,$$

$$x_n\ \text{among}\ (M_1, \dots, M_t)$$

- When the process under consideration is P, this query is true if and only if

P{M1/x1, . . . ,Mn/xn} ≈ P{N1/x1, . . . , Nn/xn}

Observation equivalent examples

```
free c : channel .

(*  Types *)
type key.
type bitstring.

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

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

(*  Query  *)
free  secret : bitstring [private]
.
(* Prove  strong secrecy of secret *)
noninterf secret.

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

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

Observation equivalent examples cont'd

```
free c : channel .

(*  Types *)
  type key
  type bitstring

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

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

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

(*  Main process *)
  process  out(c, x)
```

- Prove that the attacker cannot distinguish *n* from *hash(n).*

Observation equivalent examples cont'd

- Voters encrypt their votes with the administrator's public key.
- Prove that the attacker cannot guess voter's vote.

```
  free c : channel .

(*  Types *)
  type skey.
  type pkey.

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

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

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

(*  Main process *)
 process
     new skA : skey;
     let skA = pk(skA) in
     out(c, pkA);
     ! ( Voter(pkA) | Administrator(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”.

$$\text{Prove } P(M) \approx P(N) \text{ for some different } M, N \text{ then } P(choice[M, N])$$

- *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*

$$\text{To prove } P(skA, vote1) \mid P(skB, vote2) \approx P(skA, vote2) \mid P(skB, vote1): \quad P(skA, choice[vote1, vote2]) \mid P(skB, choice[vote2, vote1])$$

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: 1.86pl3 (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 undertand 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] *)

ProVerif installation

- 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/

- ProVerif's versions: 1.85; 1.86pl3; 1.87beta6 (newest)
  - 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/)
  - 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***

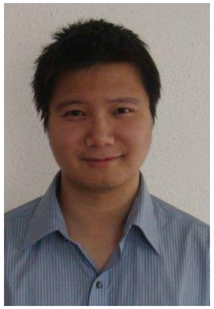
- Execution
  - Command line: *./proverif [options]* < *filename.pv* >

    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

Kérdések?

KÖSZÖNÖM A FIGYELMET!

Híradástechnikai Tanszék



Ta Vinh Thong

BME *Hálózati Rendszerek és Szolgáltatások Tanszék* k

Lab of Cryptography and System Security (CrySyS)

thong at crysys dot hu