A Short Tutorial on Proverif

Alfredo Pironti and Riccardo Sisto

Politecnico di Torino, Italy

Outline

• PART 1: how the tool works (Riccardo Sisto)
  – Context: Abstract modelling of security protocols (Dolev-Yao)
  – Proverif architecture
  – Specifying protocols and properties
  – Translation into Horn clauses and solving method

• PART 2: practice (Alfredo Pironti)
  – Running the tool
  – Case study: Verifying the SSH transport protocol with Proverif
Abstract (Dolev-Yao) Modelling of Security Protocols

- Data represented as terms of a term algebra (e.g. `sencrypt(pair(x, y), k)`)

- Active attackers
  - Can eavesdrop on public channels
  - Can delete, modify, send messages on public channels
  - Can generate new nonces
  - Can forge new messages by combining known data and their generated nonces by public operations (under the constraints of perfect cryptography)
  - Cannot forge or guess data in other ways

Perfect Cryptography

- Models the ideal properties of cryptography

- Example: perfect shared-key encryption
  - A cyphertext can be decrypted only if the right key is known
  - The encryption key cannot be deduced from the cyphertext
  - An encrypted message is sufficiently redundant so that the decryption algorithm can detect whether or not it has succeeded
Why Dolev-Yao Models?

- Resistance against Dolev-Yao attackers means the protocol is free from big logical errors
  - i.e. errors that can be exploited even under the simplifying assumptions of the model

- Dolev-Yao models are simple and can be analysed automatically

- Good for first (and fast) sanity checks (before more accurate analyses)

Automating the Analysis

- Key points:
  - Undecidability of main security properties with unbounded number of sessions (Durgin et al, JCS, 2004)
  - NP-completeness of checking main security properties with bounded number of sessions (Rusinowitch & Turani, TCS 2003)

- Possible approaches:
  - Analyse protocols with few sessions (e.g. by state exploration) => a result is eventually obtained but errors can be missed
  - Analyse protocols with unbounded sessions (by automated procedures that may not terminate or may not give a response)
Proverif

- Tool for automated analysis of security protocols with Dolev-Yao models, developed by Bruno Blanchet (ENS, Paris)

- Protocol model expressed by a process calculus and then translated to a logic program

- Models unbounded sessions

- Automated resolution-based analysis of the logic program

- Note:
  - this tutorial just focuses on the main features of Proverif

Proverif Architecture

Diagram showing the Proverif architecture with steps involving translation, resolution, and security properties.
Specifying Protocols

• Input language:
  – pi calculus extended with cryptographic and data manipulation primitives
  – very close to applied pi-calculus (Abadi, Fournet, POPL’01)

• Each pi-caculus process models a protocol principal
  – Honest principals behave according to the protocol
  – Attacker can behave in any way

Extended Pi-Calculus: term Syntax

\[
M, N ::= \text{terms} \\
x, y, z \quad \text{variable} \\
a, b, c, k \quad \text{name} \\
f(M_1, \ldots, M_n) \quad \text{constructor application}
\]
Extended Pi-Calculus: process Syntax

\[ P, Q ::= \begin{array}{ll}
M & \text{processes} \\
M\langle N \rangle.P & \text{output} \\
M(x).P & \text{input} \\
0 & \text{nil} \\
P | Q & \text{parallel composition} \\
!P & \text{replication} \\
(\nu a)P & \text{restriction (new)} \\
\text{let } x=g(M_1,\ldots,M_n) \text{ in } P \text{ else } Q & \text{destructor application} \\
\text{if } M = N \text{ then } P \text{ else } Q & \text{conditional} \\
\text{let } x=M \text{ in } P & \text{binding} \\
\text{event}(M).P & \text{event} \\
\end{array} \]

Formal Semantics

- Process evolution defined operationally by a reduction relation on processes
- Destructor application defined by rewriting rules
Destructor Semantics

- Used to define the (ideal) properties of cryptographic and data manipulation primitives

- Example: modelling Shared-key encryption
  - Constructor: \( \text{senc}(x,y) \) encrypts \( y \) under key \( x \)
  - Destructor: \( \text{sdec}(x,y) \) decrypts \( y \) with key \( x \)
  - Rewrite rules: \( \text{sdec}(x,\text{senc}(x,y)) \rightarrow y \)
  - Proverif syntax: \[
  \text{fun senc/2.}
  \text{reduc sdec(x, senc(x,y)) = y.}
  \]

Process Semantics

- \( \mathcal{E} \) set of names
- \( \mathcal{P} \) multiset of processes

- If \( \mathcal{E} = \{a_1,\ldots,a_n\} \) and \( \mathcal{P} = \{P_1,\ldots,P_m\} \)

  semantic configuration \( \mathcal{E},\mathcal{P} \) corresponds to

  \[
  (\nu a_1) \cdots (\nu a_n) (P_1 \mid \ldots \mid P_m)
  \]

- Initially, for process \( P \), configuration is \( \text{fn}(P), \{P\} \)
Process Semantics

- $\varepsilon P \cup \{M<N>.P, M(x).Q\} \rightarrow \varepsilon P \cup \{Q[N/x]\}$ (Red I/O)
- $\varepsilon P \cup \{0\} \rightarrow \varepsilon P$ (Red Nil)
- $\varepsilon P \cup \{P|Q\} \rightarrow \varepsilon P \cup \{P,Q\}$ (Red Par)
- $\varepsilon P \cup \{!P\} \rightarrow \varepsilon P \cup \{P, !P\}$ (Red Repl)
- $\varepsilon P \cup \{(\nu a)P\} \rightarrow \varepsilon \cup \{a\}, P \cup \{P[a'/a]\}$ with $a' \notin \varepsilon$ (Red Res)
- $\varepsilon P \cup \{let\ x=g(M_1,\ldots,M_n)\ in\ P\ else\ Q\} \rightarrow \varepsilon P \cup \{P[M'/x]\}$ (Red Destr1)
  - if $g(M_1,\ldots,M_n) \rightarrow M'$
- $\varepsilon P \cup \{let\ x=g(M_1,\ldots,M_n)\ in\ P\ else\ Q\} \rightarrow \varepsilon P \cup \{Q\}$ (Red Destr2)
  - if there is no $M'$ such that $g(M_1,\ldots,M_n) \rightarrow M'$
- $\varepsilon P \cup \{if\ M = N\ then\ P\ else\ Q\} \rightarrow \varepsilon P \cup \{P\}$ (Red Cond1)
  - if $M = N$
- $\varepsilon P \cup \{let\ x=M\ in\ P\} \rightarrow \varepsilon P \cup \{P[M/x]\}$ (Red Bind)
- $\varepsilon P \cup \{event(M).P\} \rightarrow \varepsilon P \cup \{P\}$ (Red Event)

Example: Handshake Protocol

- Message 1  $S \rightarrow C$: $\{|k\}_{skS}^{pkC} \quad k$ fresh
- Message 2  $C \rightarrow S$: $\{s\}_{k}$

$PS = c(xpkC). (\nu k) c<penc(xpkC, sign(skS, k))>. c(x). let x$s = sdec(k, x) in 0$

$PC = c(y). let y' = pdec(skC, y) in$
  - if $checksign(xpkS, y') = ok$ then
  - let $xk = getmess(y')$ in
  - $c<senc(xk, s)>. 0$

$P = (\nu skS) (\nu skC) let xpkS = pk(skS) in let xpkC = pk(skC) in$
  - $c<xpksri, c<xpck>, (IPA \mid IPB)$
Specifying Properties: Secrecy

- **Intuitive property**: an attacker must not be able to get closed terms that are intended to be secret (e.g. names in the Handshake protocol)

Formal Specification of Secrecy

- **S-Adversary**: any closed process Q with fn(Q) ⊆ S

- **Trace** $T = \mathcal{E}_0 P_0 \rightarrow^* \mathcal{E}_r P'$ outputs N iff T contains a reduction $\mathcal{E}_r P \cup \{M<N>.P, M(x).Q\} \rightarrow \mathcal{E}_r P \cup \{P, Q[N/x]\}$ with $M \in S$

- Closed process $P$ preserves the secrecy of N from S-Adversaries if
  $\forall$ S-Adversary $Q$, $\forall T = fn(P) \cup S, \{P, Q\} \rightarrow^* \mathcal{E}_r P'$
  $T$ does not output $N$. 
Translation into Horn Clauses
(for secrecy verification)

• Definition of predicates:
  – attacker(x) the attacker may have message x
  – message(x,y) message y may appear on channel x

• Remark: new names generated in different sessions (e.g. k in Handshake Protocol) are not differentiated
  – Partial differentiation is introduced by turning k into k[xpkC]
  ⇒ Sessions that receive the same xpkC are not differentiated
  General rule: a new name becomes function of all the messages received by the process before creating it

Horn Clauses for the Attacker

• For each \( a \in S \cup \{b\} \) (b: fresh names created by attacker)
  \[ \text{attacker}(a[]) \] (Ri)

• For each public constructor \( f(M_1,\ldots,M_n) \)
  \[ \text{attacker}(x_1) \land \ldots \land \text{attacker}(x_n) \Rightarrow \text{attacker}(f(x_1,\ldots,x_n)) \] (Rf)

• For each public destructor rule \( g(M_1,\ldots,M_n) \rightarrow M \)
  \[ \text{attacker}(M_1) \land \ldots \land \text{attacker}(M_n) \Rightarrow \text{attacker}(M) \] (Rg)

• message(x,y) \land attacker(x) \Rightarrow attacker(y) (Rl)

• attacker(x) \land attacker(y) \Rightarrow message(x,y) (Rs)
Horn Clauses for the Protocol

• For each message $N$ the protocol can send on a channel $M$ after having received messages $N_1, \ldots, N_n$ on channels $M_1, \ldots, M_n$ respectively

$$\text{message}(M_1, N_1) \land \cdots \land \text{message}(M_n, N_n) \Rightarrow \text{message}(M, N)$$

• If all the channels are public, the clauses can be changed into

$$\text{attacker}(N_1) \land \cdots \land \text{attacker}(N_n) \Rightarrow \text{attacker}(N)$$

Remark

• The number of times a message appears is ignored
  - The Horn clauses approximate the behaviour of the protocol with one that can send and receive each message an arbitrary number of times
Relating the two Models

Correctness (for secrecy):

- Let $R_{P,S}$ be the set of clauses modelling the protocol process $P$ combined with $S$-Adversaries.

- If there exist an $S$-Adversary $Q$ and a trace $T = fn(P) \cup S, (P,Q) \rightarrow^* \ell, \ell'$ that outputs $N$, then the fact $\text{attacker}(N)$ is derivable from $R_{P,S}$.

=> If one proves $\text{attacker}(N)$ is not derivable from $R_{P,S}$, one has proved $P$ preserves the secrecy of $N$ against $S$-Adversaries.

Relating the two Models

- The logic theory described by the Horn clauses over-approximates the behavior of the real protocol.

  - Freshness, repetition of send/receive

=> false positives are possible.
Example

Process

\[(\forall privc) \ (\text{privc}<s>. \text{pubc}<\text{privc}>. \ 0 \ | \ \text{privc}(x). \ 0)\]

preserves the secrecy of \(s\) against \{\text{pubc}\}-Adversaries

but Proverif cannot prove it, because the Proverif model

\[(\forall privc) \ (\neg \text{privc}<s>. \text{pubc}<\text{privc}>. \ 0 \ | \ \neg \text{privc}(x). \ 0)\]

Resolution

- Approximations are key factor for automatic and efficient solution algorithm
- However, a standard Prolog engine would not terminate
  - Main problem: clauses like
    \[\text{attacker}(\text{senc}(x,y)) \land \text{attacker}(x) \Rightarrow \text{attacker}(y)\]
    generate bigger and bigger facts by SLD-resolution
- Proverif solves the problem by a custom (and clever) resolution algorithm (with free selection)
Resolution Algorithm

Initial set of clauses $R_0 = R_{P,S}$

saturate

Simplified set of clauses $R_1$

Fact F
derivable

Set of instances of F derivable from clauses $R_1$

Phase 1 : Saturation

• Simplifies the set of clauses by
  – applying resolution to pairs of clauses with free selection
    (generate combined clauses $C \lor F_0 C'$ when $F_0$ is selected)
  – simplifying redundant clauses (function simplify())
  – eliminating subsumed clauses (function elim())

• Note: hypotesis attacker(x) is never selected
Saturation Algorithm

saturate(R0):

1. \( R \leftarrow \emptyset \)

2. \( \forall C \in R_0, \quad R \leftarrow \text{elim}(\text{simplify}(C) \cup R) \)

3. Repeat until a fixpoint is reached
   • \( \forall C \in R \) such that the conclusion is selected
   • \( \forall C' \in R, \quad \forall F_0 \) such that \( F_0 \) is selected and \( C \circ_{F_0} C' \) is defined
     • \( R \leftarrow \text{elim}(\text{simplify}(C \circ_{F_0} C') \cup R) \)

4. Return \( \{C \in R \mid \text{no hypothesis is selected in } C\} \)

Simplifications

• Elimination of tautologies

• Elimination of duplicate hypotheses

• Elimination of hypotheses \( \text{attacker}(x) \) when \( x \) does not appear elsewhere (always fulfilled)

• Elimination of hypotheses \( \text{attacker}(N) \) when \( N \) is assumed to be secret (secrecy assumptions)
Correctness of Saturation

• Let $F_{not}$ be a set of facts the user claims to be non-derivable (secrecy assumptions)

• If $\forall F' \in F_{not}$ no instance of $F'$ is derivable from $\text{saturate}(R_0)$

then a closed fact $F$ is derivable from $R_0$ iff $F$ is derivable from $\text{saturate}(R_0)$

Phase 2 : Derivation Search

• Searches a derivation of a given fact $F$ using a set of rules $R$ by backward depth-first search:

  - $\text{deriv}(F, R) = \text{deriv}(F => F, \emptyset, R)$

  - $\text{deriv}(C, R', R) = \emptyset$ if $\exists C' \in R$ that subsumes $C$

  - $\text{deriv}(=> F, R', R) = \{F\}$

  - $\text{deriv}(C, R', R) = U\{\text{deriv} (\text{simplify}'(C' \circ_{R_0} C), \{C\} \cup R', R) | C' \in R, F_0$ is selected and $C \circ_{R_0} C'$ is defined $\}$
Correctness of Derivation Search

- If $\forall F'' \in F_{not}$ derivable$(F'',R1) = \emptyset$

  then a closed instance $F'$ of $F$ is derivable from $R1$ iff $\exists F'' \in$ derivable$(F,R1)$ and a substitution $\sigma$ such that $\sigma F'' = F'$

Summary of Resolution Algorithm

- Let $F_{not}$ be a set of facts the user claims to be non-derivable (secrecy assumptions)

- solve$_{P,S}(F)$:
  1. $R1 = \text{saturate}(R_{P,S})$
  2. $\forall F' \in F_{not}$, if derivable$(F',R1) \neq \emptyset$, terminate with error
  3. Return derivable$(F,R1)$
Extension to Equational Theories

- Proverif can handle some equational theories
- Useful for modelling the special properties of some cryptosystems
- Example: Diffie-Hellman key generation
  - We need to model the equality $g^{xy} \mod p = g^{yx} \mod p$
  - We can define two constructors $f(v,w) = w^v \mod p$, $f'(z) = g^z \mod p$ and the equality $f(y,f'(x)) = f(x,f'(y))$

Extension to Equational Theories

- Proverif automatically translates equations into (non-deterministic) rewrite rules for function symbols
  - Example: for Diffie-Hellman key generation functions the equality $f(y,f'(x)) = f(x,f'(y))$ is translated into the rewrite rules
    
    $f(y,f'(x)) \rightarrow f(x,f'(y))$  
    $f(x,y) \rightarrow f(x,y)$

- The solving algorithm is adapted to perform unification modulo the equational theory by reducing it to syntactic unification via the rewriting rules
- The solving algorithm may not terminate (it has been shown to work well with the Diffie-Hellman equality)
Correspondence Properties

- Specify order relationships that bind trace events
- Can be used to specify authentication (e.g. agreement)

Example: Authentication in the Handshake Protocol

PS = c(xpkC). (v k)
    event(bS(xpkS,xpkC,k)).
    c<penc(xpkC, sign(skS, k))>.
    c(x). let xs=sdec(k,x) in 0

PC = c(y). let y'=pdec(skC,y) in
    if checksign(xpkS, y')=ok then let xk=getmess(y') in
    event(eC(xpka,xpkb,xk)).
    c<senc(xk,s)> in 0

- In each trace, if event eC(x,y,z) occurs, event bS(x,y,z) must have occurred before
  (ev:eC(x,y,z) ==> ev:bS(x,y,z)).
Verification of Correspondences

• Logic theory is extended with new predicates:
  – event(M) event(M) may have been executed
  – m-event(M) event(M) must have been executed

• Correspondence ev:eC(x,y,z) ==> ev:bS(x,y,z) can be proved by proving that in each trace event(eC(x,y,z)) may have been executed only if event(bS(x,y,z)) must have been executed

Verification of Correspondences

• Fresh names generated in different sessions now are differentiated (by instrumenting the protocol with session identifiers)
  – k is turned into k[xpkC,i] where i identifies the session

• Clauses are generated for event actions: For each event(N) the protocol can execute after having received messages N₁,…,Nₙ on channels M₁,…,Mₙ respectively

message(M₁, N₁) ∨ … ∨ message(Mᵣ, Nᵣ) ==> event(N)
Verification of Correspondences

- An m-event(N') hypothesis is added to a clause for message output when event event(N') occurs before the output

\[
\text{message}(M_1, N_1) \land \ldots \land \text{message}(M_n, N_n) \land \text{m-event}(N') \Rightarrow \\
\text{message}(M, N)
\]

- m-event(N) predicates occur only as hypotheses and are never selected during resolution

Modified Derivation Search

- \text{derive}(F, R) now returns a set of clauses without selectable hypotheses (a clause may have m-event() hypotheses)

- The clauses $H \Rightarrow F' \in \text{derivable}(F, R)$ are derived from the rules in $R$ and have $F'$ that is an instance of $F$

  - $\text{deriv}(C, R', R) = \emptyset$ if $\exists C' \in R$ that subsumes $C$

  - $\text{deriv}(C, R', R) = \{C\}$ if $C$ has no selectable hypothesis

  - $\text{deriv}(C, R', R) = \bigcup \{ \text{deriv}(\text{\textlangle\textrangle}simplify('C' \circ_{F_0} C'), \{C\} \cup R', R) | \\
    C' \in R, F_0 \text{ is selected and } C \circ_{F_0} C' \text{ is defined} \}$
Example

- Verification of $ev:e(x_1,...,x_n) \implies ev:e'(x_1,...,x_n)$
  - Let $R_{P,S}$ be the set of clauses modelling the protocol process $P$ combined with $S$-Adversaries
  - If $\forall C \in \text{solve}_{P,S}(\text{event}(e(x_1,...,x_n)))$ such that $C = H \implies \text{event}(e(p_1,...,p_n))$ we have $m\text{-event}(e(p_1,...,p_n)) \in H$, then $P$ satisfies the correspondence $ev:e(x_1,...,x_n) \implies ev:e'(x_1,...,x_n)$ against $S$-Adversaries

Injective Correspondences

- $ev:e(x_1,...,x_n) \implies ev:e'(x_1,...,x_n)$ is true even when the same execution of event $e'(x_1,...,x_n)$ corresponds to more executions of event $e(x_1,...,x_n)$

- Injective correspondence $ev:e(x_1,...,x_n) \implies evinj:e'(x_1,...,x_n)$ requires that each occurrence of event $e(x_1,...,x_n)$ corresponds to a distinct occurrence of event $e'(x_1,...,x_n)$

- In order to prove injective correspondences Proverif adds session identifiers to $m\text{-event}()$ predicates
General Correspondences

• Correspondences can be combined together to form more complex queries:
  - \(\text{ev}:e(x_1,x_2) \Rightarrow (\text{evinj}:e_2(x_1,x_2) \Rightarrow \text{evinj}:e_1(x_1))\)
  - \(\text{ev}:e(x_1,x_2) \Rightarrow (\text{evinj}:e_2(x_1,x_2) \Rightarrow \text{evinj}:e_1(x_1)) \mid (\text{evinj}:e_4(x_1,x_2) \Rightarrow \text{evinj}:e_3(x_1))\)

• The query
  - \(\text{ev}:e(x_1,x_2)\) means event \(e(x_1,x_2)\) is never executed

Observational Equivalence

• Informally, two processes are observationally equivalent if an attacker cannot distinguish them.

• Example: \((\nu \ k) \ M<H(k)>.0 \approx (\nu \ k) \ M<k>.0\)

• Example: \((\nu \ k) \ M<H(k),k>.0 \neq (\nu \ k) \ M<k,k>.0\)

because they can be distinguished by the attacker

\(M(x). \ \text{let} \ l=\text{fst}(x) \ \text{in} \ \text{let} \ r=\text{snd}(x) \ \text{in} \ \text{if} \ l=H(r) \ \text{then} \ \text{success} \ \text{else} \ \text{fail}\)
**Strong Secrecy**

- Observational equivalence can be used to give a stronger definition of secrecy, closer to non-interference

- Intuitive meaning: strong secrecy of a secret is preserved when an adversary cannot see any difference when the secret changes

- Strong secrecy takes into account implicit flows (e.g. different observable behaviours depending on a test on the secret). Example:

  \[
  \text{let } x = \text{sdec}(s,k) \text{ in } M < N > .0 \text{ else } M < N' > .0
  \]

---

**Strong Secrecy: Formal Definition**

- Process P preserves the strong secrecy of its free variables iff for all closed substitutions \( \sigma \) and \( \sigma' \) of domain \( \text{fv}(P) \),

  \[
  \sigma P \approx \sigma' P
  \]
Observational Equivalence: Formal Definition

• C (evaluation context): an expression built from [], C|P, P|C, (ν a)C

• P\uparrow M (P emits on M): P (→∪≡)* C[M<ν>M].Q and C does not bind M

• ≈ (Observational equivalence): the largest symmetric relation R on closed processes such that P R Q implies
  – If P\uparrow M then Q\uparrow M
  – If P→P’ then ∃Q’ such that Q→*Q’ and P’ R Q’
  – C[P] R C[Q] for any C

Verification of Observational Equivalence

• Proverif supports verification of observational equivalence between processes that differ only by some terms

• Example: (ν k) (ν k’) M<choice[k,k’]>0 can be defined to verify that

(ν k) M<H(k)>0 ≈ (ν k’) M<H(k’)>0

i.e. that (ν k) M<H(k)>0 preserves strong secrecy of k
Verification of Observational Equivalence

- Proverif tries to prove a stronger condition that implies observational equivalence
  - Intuitively: each reduction step proceeds uniformly for all the values of choice terms

- The logic theory is extended by introducing the predicate \text{testunif}(x,y)
  - Intuitively: \text{testunif}(M,M') is true if the existence of a substitution that unifies M and M' depends on the values of the choice terms

Termination

- The saturation algorithm may not terminate

- Termination has been proved for a class of tagged protocols (Blanchet, Podelski, TCS 2005):
  - Limited set of primitives (including all the main ones)
  - No private channels
  - Crypto functions always applied to tagged data (with different tags for each occurrence of each function)
  - Tags always checked on application of destructors
  - No else in destructor applications
  - Atomic keys
Example: Handshake Protocol

- **Message 1**  
  $S \rightarrow C$: \( \{ T_0, \{ T_1, k \}_{skS} \}_{pkC} \) k fresh

- **Message 2**  
  $C \rightarrow S$: \( \{ T_2, s \}_k \)

**PS**  
$c(xpkC). (\forall \: k) \: c<(T_0, \: penc \: (xpkC, \: sign(skS, k)))>. c(x). let \: y = sdec(k, x) in let \: \text{fst}(y) = T_2 in xs = \text{snd}(xst) in 0$

**PC**  
$c(y). let \: y' = pdec(skC, y) in let \: \text{fst}(y') = T_0 in let \: y'' = \text{snd}(y') in if \: \text{checksign}(xpkS, y'') = \text{ok} then let \: z = \text{getmess}(y'') in let \: \text{fst}(z) = T_1 in let \: xk = \text{snd}(z) in c<\text{senc}(xk, s)>. 0$

---

**Attack Reconstruction**

- Let us assume we have found a derivation $D$ of a fact $F$ that prevents building the desired proof

- Attack Reconstruction looks for an *attack trace compatible with* $D$.

- **Method:**
  - Use a trace semantics restricted (i.e. driven) by $D$
  - Exhaustively search an attack in the (finite) set of traces of the restricted semantics
Attack Reconstruction

- If Attack Reconstruction finds an attack trace $T$, then $T$ is a real attack (counterexample found).
  - Does not work with equivalence verification (e.g. strong secrecy)
  - For injective correspondences the reconstructed trace is not necessarily a real attack
- If no attack trace is found by Attack Reconstruction on found derivations, an attack (not included in the restricted semantics) may still exist

References

- Verification of Secrecy

- Verification of Correspondences

- Verification of Strong Secrecy and Observational Equivalences and Support for Equational Theories
References

• Termination for Tagged Protocols

• Attack Reconstruction