Automatic Verification of Cryptographic Protocols
in the Symbolic Model
Automatic Verifier ProVerif

Bruno Blanchet
Inria, Paris
Bruno.Blanchet@inria.fr

Year 2016-17
Overview of the protocol verifier ProVerif

Protocol:
Pi calculus + cryptography
Primitives: rewrite rules, equations

Properties to prove:
Secrecy, authentication, process equivalences

Automatic translator

Horn clauses
Derivability queries

Resolution with selection

Non-derivable: the property is true
Derivation

Attack: the property is false
False attack: I don’t know
Overview

1. A dialect of the applied pi calculus
2. Undecidability
3. The Horn clause representation
4. The resolution algorithm
5. Experimental results
6. Formal translation from the applied pi calculus
7. Extension to correspondences
8. Extension to observational equivalence
9. Conclusion
Overview

1. A dialect of the applied pi calculus
2. Undecidability
3. The Horn clause representation
4. The resolution algorithm
5. Experimental results
6. Formal translation from the applied pi calculus
7. Extension to correspondences
8. Extension to observational equivalence
9. Conclusion
What is the applied pi calculus?

The **applied pi calculus** is an extension of the pi calculus designed to represent cryptographic protocols.

The **pi calculus** is a process calculus:

- processes **communicate**: they can send and receive messages on channels
- several processes can **execute in parallel**.

In the pi calculus, messages and channels are **names**, that is, atomic values $a, b, c, \ldots$. 
Example

\text{out}(c, a) \parallel \text{in}(c, x).\text{out}(d, x)

The first process sends $a$ on channel $c$, the second one inputs this message, puts it in variable $x$ and sends $x$ on channel $d$.

The link with cryptographic protocols is clear:

- Each participant of the protocol is represented by a process
- The messages exchanged by processes are the messages of the protocol.

However, in protocols, messages are not necessarily atomic values.

The names of the pi calculus are replaced by terms in the applied pi calculus.
Syntax of the process calculus

Pi calculus + cryptographic primitives

\[ M, N ::= \]
\[ \begin{align*}
    &x, y, z &\quad \text{variable} \\
    &a, b, c, k, s &\quad \text{name} \\
    &f(M_1, \ldots, M_n) &\quad \text{constructor application}
\end{align*} \]

\[ P, Q ::= \]
\[ \begin{align*}
    &\text{processes} \\
    &\text{output} \\
    &\text{input} \\
    &\text{destructor application} \\
    &\text{local definition} \\
    &\text{conditional} \\
    &\text{nil process} \\
    &\text{parallel composition} \\
    &\text{replication} \\
    &\text{restriction}
\end{align*} \]

Bruno Blanchet (Inria)
Constructors and destructors

Two kinds of operations:

- **Constructors** $f$ are used to build terms $f(M_1, \ldots, M_n)$
- **Destructors** $g$ manipulate terms
  
  ```
  let x = g(M_1, \ldots, M_n) in P else Q
  ```

  Destructors are defined by rewrite rules $g(M_1, \ldots, M_n) \to M$. 
Examples of constructors and destructors

Shared-key encryption: \( \{M\}_N \); one decrypts with the key \( N \)
- Constructor: Shared-key encryption \( \text{senc}(M, N) \).
- Destructor: Decryption \( \text{sdec}(M', N) \)

\[
\text{sdec}(\text{senc}(x, y), y) \rightarrow x.
\]

Perfect encryption assumption: one can decrypt only if one has the key.
Examples of constructors and destructors

Public-key encryption: $\{M\}_{pk}$; one decrypts with the secret key $sk$

- Constructors: Public-key encryption $aenc(M, N)$.
  Public key generation $pk(N)$.
- Destructor: Decryption $adec(M', N)$

$$adec(aenc(x, pk(y)), y) \rightarrow x.$$
Examples of constructors and destructors (continued)

**Signature:** \( \{M\}_{sk} \); one verifies with the public key \( pk \)

- **Constructor:** Signature \( \text{sign}(M, N) \).
- **Destructors:** Signature checking \( \text{check}(M', N') \)

\[
\text{check}(\text{sign}(x, y), \text{pk}(y)) \rightarrow x.
\]

**Message extraction** \( \text{getmess}(M') \)

\[
\text{getmess}(\text{sign}(x, y)) \rightarrow x.
\]

Here, we assume that the signed message \( \text{sign}(M, N) \) contains the message \( M \) in the clear.

**Exercise**

Model signatures that do not reveal the signed message.
One-way hash function:

- Constructor: One-way hash function \( H(M) \).

Very idealized model of a hash function (essentially corresponds to the random oracle model).
Examples of constructors and destructors (continued)

Tuples:
- Constructor: tuple $(M_1, \ldots, M_n)$.
- Destructors: projections $ith(M)$
  
  $$ith((x_1, \ldots, x_n)) \rightarrow x_i$$

Tuples are used to represent all kinds of data structures in protocols.
Example: The Denning-Sacco protocol

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

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

\((A)\)
\[
\quad ! \text{ in}(c, x_{-pk_B}).\text{new } k.\text{out}(c, \text{aenc}(\text{sign}(k, sk_A), x_{-pk_B})).
\]
\[
\quad \text{in}(c, x).\text{let } s = \text{sdec}(x, k) \text{ in } 0
\]

\((B)\)
\[
\quad \parallel ! \text{ in}(c, y).\text{let } y' = \text{adec}(y, sk_B) \text{ in}
\]
\[
\quad \text{let } k = \text{check}(y', pk_A) \text{ in out}(c, s\text{enc}(s, k))
\]
Demo: Denning-Sacco protocol

- examplesnd/demosimp/pidenning-sacco-v1
- examplesnd/demosimp/pidenning-sacco-corr-v1
Exercise: The Needham-Schroeder public-key protocol

Model the following protocol:

Message 1. \( A \to B \) \( \{ N_a, A \}^{pk_B} \) \( N_a \) fresh

Message 2. \( B \to A \) \( \{ N_a, N_b \}^{pk_A} \) \( N_b \) fresh

Message 3. \( A \to B \) \( \{ N_b \}^{pk_B} \)
The semantics is defined by *reduction* $P \rightarrow P'$: the execution of the process is modeled by transforming it into another process.

Main reduction rule = communication

$$\text{out}(N, M).Q \parallel \text{in}(N, x).P \rightarrow Q \parallel P\{M/x\}$$

**Example**

$$\text{out}(c, a) \parallel \text{in}(c, x).\text{out}(d, x) \rightarrow \text{out}(d, a)$$

The communicating processes are not always in the above form, so we need an equivalence relation to prepare the reduction.
Equivalence relation

\[ P \parallel 0 \equiv P \]
\[ P \parallel Q \equiv Q \parallel P \]
\[ (P \parallel Q) \parallel R \equiv P \parallel (Q \parallel R) \]
\[ \text{new } a_1.\text{new } a_2.P \equiv \text{new } a_2.\text{new } a_1.P \]
\[ \text{new } a.(P \parallel Q) \equiv P \parallel \text{new } a.Q \text{ if } a \notin \text{fn}(P) \]
\[ P \equiv Q \Rightarrow P \parallel R \equiv Q \parallel R \]
\[ P \equiv Q \Rightarrow \text{new } a.P \equiv \text{new } a.Q \]
\[ P \equiv P \]
\[ Q \equiv P \Rightarrow P \equiv Q \]
\[ P \equiv Q, Q \equiv R \Rightarrow P \equiv R \]
Reduction relation

\[
\text{out}(N, M).Q \parallel \text{in}(N, x).P \rightarrow Q \parallel P\{M/x\} \quad \text{(Red I/O)}
\]

let \(x = g(M_1, \ldots, M_n)\) in \(P\) else \(Q \rightarrow P\{M'/x\}
\]

if \(g(M_1, \ldots, M_n) \rightarrow M'\) \(\text{(Red Destr 1)}\)

let \(x = g(M_1, \ldots, M_n)\) in \(P\) else \(Q \rightarrow Q\)

if there exists no \(M'\) such that \(g(M_1, \ldots, M_n) \rightarrow M'\) \(\text{(Red Destr 2)}\)

\(!P \rightarrow P \parallel !P\) \(\text{(Red Repl)}\)

\(P \rightarrow Q \Rightarrow P \parallel R \rightarrow Q \parallel R\) \(\text{(Red Par)}\)

\(P \rightarrow Q \Rightarrow \text{new } a.P \rightarrow \text{new } a.Q\) \(\text{(Red Res)}\)

\(P' \equiv P, P \rightarrow Q, Q \equiv Q' \Rightarrow P' \rightarrow Q'\) \(\text{(Red } \equiv)\)
Example

\[
\text{in}(c, xpk_{A}).\text{in}(c, xpk_{B}).\text{out}(c, xpk_{B})
\]

\[
\parallel \quad \text{new } sk_{A}.\text{new } sk_{B}.\text{let } pk_{A} = \text{pk}(sk_{A}) \text{ in let } pk_{B} = \text{pk}(sk_{B}) \text{ in }
\]

\[
\text{out}(c, pk_{A}).\text{out}(c, pk_{B}).
\]

\[
( \quad \text{! in}(c, x_{-}pk_{B}).\text{new } k.\text{out}(c, \text{aenc(\text{sign}(k, sk_{A}), x_{-}pk_{B}))}. \\
\text{in}(c, x).\text{let } s = \text{sdec}(x, k) \text{ in } 0
\]

\[
\parallel \quad \text{! in}(c, y).\text{let } y^{'} = \text{adec}(y, sk_{B}) \text{ in }
\]

\[
\text{let } k = \text{check}(y^{'}, pk_{A}) \text{ in out}(c, \text{senc}(s, k)))
\]
Example (2)

\[
\rightarrow^* \Rightarrow \ \\
\text{in}(c, xpk_A).\text{in}(c, xpk_B).\text{out}(c, xpk_B) \parallel \text{new } sk_A.\text{new } sk_B.\text{out}(c, pk(sk_A)).\text{out}(c, pk(sk_B)).
\]

( \! \text{in}(c, xpk_B).\text{new } k.\text{out}(c, \text{aenc}(\text{sign}(k, sk_A), xpk_B)). \\
in(c, x).\text{let } s = \text{sdec}(x, k) \text{ in } 0
\]

\parallel \text{! in}(c, y).\text{let } y' = \text{adec}(y, sk_B) \text{ in } \\
\text{let } k = \text{check}(y', pk(sk_A)) \text{ in } \text{out}(c, \text{senc}(s, k)))
Example (3)

\[ \equiv \text{new } sk_A.\text{new } sk_B. \]

\[ (\text{out}(c, \text{pk}(sk_A)).\text{out}(c, \text{pk}(sk_B))). \]

\[ (\ ! \text{in}(c, x_{-pk_B}).\text{new } k.\text{out}(c, \text{aenc}(\text{sign}(k, sk_A), x_{-pk_B})). \]

\[ \text{in}(c, x).\text{let } s = \text{sdec}(x, k) \text{ in } 0 \]

\[ \| \ ! \text{in}(c, y).\text{let } y' = \text{adec}(y, sk_B) \text{ in } \]

\[ \text{let } k = \text{check}(y', \text{pk}(sk_A)) \text{ in } \text{out}(c, \text{senc}(s, k))) \]

\[ \| \text{in}(c, x_{pk_A}).\text{in}(c, x_{pk_B}).\text{out}(c, x_{pk_B})) \]
$\rightarrow^* \text{new } sk_A.\text{new } sk_B.$

$\begin{align*}
& ( ( ! \text{in}(c, x.pk_B).\text{new } k.\text{out}(c, \text{aenc}(\text{sign}(k, sk_A), x.pk_B))). \\
& \quad \text{in}(c, x).\text{let } s = \text{sdec}(x, k) \text{ in } 0 \\
& \parallel ! \text{in}(c, y).\text{let } y' = \text{adec}(y, sk_B) \text{ in } \\
& \quad \text{let } k = \text{check}(y', \text{pk}(sk_A)) \text{ in } \text{out}(c, \text{senc}(s, k))) \\
& \parallel \text{out}(c, \text{pk}(sk_B)))
\end{align*}$
Example (5)

\[ \rightarrow^* \text{ new } sk_A.\text{new } sk_B. \]

\[ ( (\text{in}(c, x_{pk_B}).\text{new } k.\text{out}(c, \text{aenc}(\text{sign}(k, sk_A), x_{pk_B}))). \]

\[ \text{in}(c, x).\text{let } s = \text{sdec}(x, k) \text{ in } 0 \]

\[ \parallel ! \text{in}(c, x_{pk_B}).\ldots.) \]

\[ \parallel (\text{in}(c, y).\text{let } y' = \text{aenc}(y, sk_B) \text{ in } \]

\[ \text{let } k = \text{check}(y', \text{pk}(sk_A)) \text{ in } \text{out}(c, \text{senc}(s, k)) \]

\[ \parallel ! \text{in}(c, y).\ldots.)) \]

\[ \parallel \text{out}(c, \text{pk}(sk_B))) \]
Example (6)

\[ \equiv \text{new } sk_A. \text{new } sk_B. \]

\[ (\text{out}(c, \text{pk}(sk_B)) \]

\[ \parallel \text{in}(c, x \_ pk_B). \text{new } k. \text{out}(c, \text{aenc}(\text{sign}(k, sk_A), x \_ pk_B)). \]

\[ \text{in}(c, x). \text{let } s = \text{sdec}(x, k) \text{ in } 0 \]

\[ \parallel \text{in}(c, y). \text{let } y' = \text{adec}(y, sk_B) \text{ in } \]

\[ \text{let } k = \text{check}(y', \text{pk}(sk_A)) \text{ in } \text{out}(c, \text{senc}(s, k)) \]

\[ \parallel ! \text{in}(c, x \_ pk_B). \ldots. \]

\[ \parallel ! \text{in}(c, y). \ldots. ) \]
\[
\rightarrow \text{new } sk_A.\text{new } sk_B.
\]
\[
(\text{new } k.\text{out}(c, \text{aenc}(\text{sign}(k, sk_A), \text{pk}(sk_B))).
\text{in}(c, x).\text{let } s = \text{sdec}(x, k) \text{ in } 0
\]
\[
\| \text{in}(c, y).\text{let } y' = \text{adec}(y, sk_B) \text{ in }
\text{let } k = \text{check}(y', \text{pk}(sk_A)) \text{ in } \text{out}(c, \text{senc}(s, k))
\]
\[
\| ! \text{in}(c, x_{-pk_B}).\ldots.
\]
\[
\| ! \text{in}(c, y).\ldots.
\]
\[ \equiv \text{new } sk_A.\text{new } sk_B.\text{new } k. \]

\[
\begin{align*}
& \text{( out}(c, \text{aenc}(\text{sign}(k, sk_A), \text{pk}(sk_B))). \\
& \text{in}(c, x).\text{let } s = sdec(x, k) \text{ in } 0 \\
& \parallel \text{in}(c, y).\text{let } y' = \text{adec}(y, sk_B) \text{ in } \\
& \quad \text{let } k' = \text{check}(y', \text{pk}(sk_A)) \text{ in } \text{out}(c, \text{senc}(s, k'))) \\
& \parallel ! \text{in}(c, x._pk_B). \ldots. \\
& \parallel ! \text{in}(c, y). \ldots. 
\end{align*}
\]
Example (9)

\[ \rightarrow^* \text{ new } sk_A.\text{ new } sk_B.\text{ new } k. \]

\[
\begin{align*}
& (\\text{ in}(c, x).\text{ let } s = sdec(x, k) \text{ in } 0 \\
& \| \text{ let } y' = \text{ adec}(\text{ aenc}(\text{ sign}(k, sk_A), \text{ pk}(sk_B)), sk_B) \text{ in } \\
& \quad \text{ let } k' = \text{ check}(y', \text{ pk}(sk_A)) \text{ in } \text{ out}(c, \text{ senc}(s, k')) \\
& \| \text{ ! in}(c, x, pk_B).\ldots. \\
& \| \text{ ! in}(c, y).\ldots) 
\end{align*}
\]
\[ \rightarrow^* \text{new } \text{sk}_A.\text{new } \text{sk}_B.\text{new } k. \]

\[
(\text{in}(c, x).\text{let } s = \text{sdec}(x, k) \text{ in } 0
\parallel \text{out}(c, \text{senc}(s, k))
\parallel ! \text{in}(c, x.pk_B)\ldots.
\parallel ! \text{in}(c, y)\ldots)
\]
→* new \( sk_A \).new \( sk_B \).new \( k \).

\[
( \text{let } s = \text{sdec(senc}(s, k), k) \text{ in 0} \\
|| \text{! in}(c, x \_ pk_B) . . . . ) \\
|| \text{! in}(c, y) . . . . )
\]
Another presentation of the semantics

Semantic configurations are $\mathcal{E}, \mathcal{P}$ where
- $\mathcal{E}$ is a set of names
- $\mathcal{P}$ is a multiset of processes

Intuitively, $\mathcal{E}, \mathcal{P}$ where $\mathcal{E} = \{a_1, \ldots, a_n\}$ and $\mathcal{P} = \{P_1, \ldots, P_m\}$ corresponds to

$$\text{new } a_1 \ldots \text{new } a_n.(P_1 \parallel \ldots \parallel P_m)$$

Initial configuration for $P$: $\text{fn}(P), \{P\}$. 
Another presentation of the semantics: reduction relation

\[ E, \mathcal{P} \cup \{0\} \rightarrow E, \mathcal{P} \]  
(Red Nil)

\[ E, \mathcal{P} \cup \{!P\} \rightarrow E, \mathcal{P} \cup \{P, !P\} \]  
(Red Repl)

\[ E, \mathcal{P} \cup \{P \parallel Q\} \rightarrow E, \mathcal{P} \cup \{P, Q\} \]  
(Red Par)

\[ E, \mathcal{P} \cup \{\text{new } a.P\} \rightarrow E \cup \{a'\}, \mathcal{P} \cup \{P\{a'/a\}\} \]  
(Red Res)

where \(a' \notin E\).

\[ E, \mathcal{P} \cup \{\text{out}(N, M).Q, \text{in}(N, x).P\} \rightarrow E, \mathcal{P} \cup \{Q, P\{M/x\}\} \]  
(Red I/O)

\[ E, \mathcal{P} \cup \{\text{let } x = g(M_1, \ldots, M_n) \text{ in } P \text{ else } Q\} \rightarrow E, \mathcal{P} \cup \{P\{M'/x\}\} \]  
(Red Destr 1)

if \(g(M_1, \ldots, M_n) \rightarrow M'\)

\[ E, \mathcal{P} \cup \{\text{let } x = g(M_1, \ldots, M_n) \text{ in } P \text{ else } Q\} \rightarrow E, \mathcal{P} \cup \{Q\} \]  
(Red Destr 2)

if there exists no \(M'\) such that \(g(M_1, \ldots, M_n) \rightarrow M'\)
\{c, s\},
\{
\begin{align*}
in(c, xpk_A). & \text{in}(c, xpk_B). \text{out}(c, xpk_B) \\
\parallel & \text{new } sk_A. \text{new } sk_B. \text{let } pk_A = \text{pk}(sk_A) \text{ in let } pk_B = \text{pk}(sk_B) \text{ in} \\\n& \text{out}(c, pk_A). \text{out}(c, pk_B).
\end{align*}
\end{align*}
\]
Example (2)

→ \{c, s\},

\{ in(c, x pk_A).in(c, x pk_B).out(c, x pk_B),

new sk_A.new sk_B.let pk_A = pk(sk_A) in let pk_B = pk(sk_B) in
out(c, pk_A).out(c, pk_B).

( ! in(c, x _pk_B).new k.out(c, aenc(sign(k, sk_A), x _pk_B)).
in(c, x).let s = sdec(x, k) in 0

∥ ! in(c, y).let y′ = adec(y, sk_B) in
let k = check(y′, pk_A) in out(c, senc(s, k)))\}
Example (2)

→ \{c, s\},

\[
\begin{align*}
&\{ \text{in}(c, xpk_A).\text{in}(c, xpk_B).\text{out}(c, xpk_B), \\
&\quad \text{new } sk_A.\text{new } sk_B.\text{let } pk_A = \text{pk}(sk_A) \text{ in let } pk_B = \text{pk}(sk_B) \text{ in} \\
&\quad \text{out}(c, pk_A).\text{out}(c, pk_B). \\
&\quad ( \! \text{in}(c, x pk_B).\text{new } k.\text{out}(c, \text{aenc}(\text{sign}(k, sk_A), x pk_B)). \\
&\quad \text{in}(c, x).\text{let } s = \text{sdec}(x, k) \text{ in 0} \\
&\| \quad \! \text{in}(c, y).\text{let } y' = \text{adec}(y, sk_B) \text{ in} \\
&\quad \text{let } k = \text{check}(y', pk_A) \text{ in out}(c, \text{senc}(s, k)))} \}
\]
Example (3)

\[ \rightarrow^* \{ c, s, sk_A, sk_B \}, \]

\[ \{ \text{in}(c, xpk_A).\text{in}(c, xpk_B).\text{out}(c, xpk_B), \]  

let \( pk_A = \text{pk}(sk_A) \) in let \( pk_B = \text{pk}(sk_B) \) in  

out\((c, pk_A).\text{out}(c, pk_B). \]

\( ( \text{! in}(c, xpk_B).\text{new} \ k.\text{out}(c, \text{aenc}(\text{sign}(k, sk_A), x_{-pk_B})). \]  

\text{in}(c, x).\text{let} \ s = \text{sdec}(x, k) \text{ in} 0 \]

\[ || \text{! in}(c, y).\text{let} \ y' = \text{adec}(y, sk_B) \text{ in} \]  

let \( k = \text{check}(y', pk_A) \text{ in} \text{out}(c, \text{senc}(s, k))) \} \]
Example (3)

$\rightarrow^* \{ c, s, sk_A, sk_B \}$,

\[
\{ \text{in}(c, xpk_A).\text{in}(c, xpk_B).\text{out}(c, xpk_B), \\
\text{let } pk_A = \text{pk}(sk_A) \text{ in let } pk_B = \text{pk}(sk_B) \text{ in} \\
\text{out}(c, pk_A).\text{out}(c, pk_B). \\
( \! \text{in}(c, x \cdot pk_B).\text{new } k.\text{out}(c, \text{aenc}(\text{sign}(k, sk_A), x \cdot pk_B))). \\
\text{in}(c, x).\text{let } s = \text{sdec}(x, k) \text{ in } 0 \\
\| \! \text{in}(c, y).\text{let } y' = \text{adec}(y, sk_B) \text{ in} \\
\text{let } k = \text{check}(y', pk_A) \text{ in out}(c, \text{senc}(s, k))) \}
\]
Example (4)

\[ \rightarrow^* \{ c, s, sk_A, sk_B \}, \]

\[
\{ \begin{align*}
\text{in}(c, x pk_A) & . \text{in}(c, x pk_B) . \text{out}(c, x pk_B), \\
\text{out}(c, pk(sk_A)) & . \text{out}(c, pk(sk_B)).
\end{align*} \}
\]

( ! \text{in}(c, x pk_B) . \text{new} k . \text{out}(c, aenc(sign(k, sk_A), x pk_B)).

\text{in}(c, x) . \text{let} \ s = sdec(x, k) \text{ in} 0

\parallel

! \text{in}(c, y) . \text{let} \ y' = adec(y, sk_B) \text{ in}

\text{let} \ k = \text{check}(y', pk(sk_A)) \text{ in} \text{out}(c, senc(s, k))) \}

Bruno Blanchet (Inria) ProVerif Year 2016-17 36 / 158
Example (4)

\[ \rightarrow^\ast \{ c, s, sk_A, sk_B \}, \]

\{ \in(c, xpk_A).\in(c, xpk_B).\out(c, xpk_B), \]

\out(c, pk(sk_A)).\out(c, pk(sk_B)). \]

( ! \in(c, x_{-}pk_B).\new k.\out(c, aenc(sign(k, sk_A), x_{-}pk_B)). \]
\in(c, x).\let s = sdec(x, k) \in 0 \]

\| ! \in(c, y).\let y' = adec(y, sk_B) \in \]
\let k = check(y', pk(sk_A)) \in \out(c, senc(s, k))) \}
Example (5)

\[ \rightarrow^* \{ c, s, sk_A, sk_B \}, \]

\[
\{ \text{out}(c, pk(sk_B)),

( ! \text{in}(c, x_{-pk_B}).\text{new} \ k. \text{out}(c, \text{aenc}(\text{sign}(k, sk_A), x_{-pk_B})).

\text{in}(c, x).\text{let} \ s = \text{sdec}(x, k) \text{ in } 0

\parallel ! \text{in}(c, y).\text{let} \ y' = \text{adec}(y, sk_B) \text{ in }

\text{let} \ k = \text{check}(y', \text{pk}(sk_A)) \text{ in } \text{out}(c, \text{senc}(s, k))) \}
\]
\[ \rightarrow^* \{ c, s, sk_A, sk_B \}, \]

\[
\{ \quad \text{out}(c, \text{pk}(sk_B)), \\
\quad ( ! \text{in}(c, x_{pk_B}).\text{new} \ k.\text{out}(c, \text{aenc}(\text{sign}(k, sk_A), x_{-pk_B}))). \\
\quad \text{in}(c, x).\text{let} \ s = \text{sdec}(x, k) \text{ in } 0 \\
\mid ! \text{in}(c, y).\text{let} \ y' = \text{adec}(y, sk_B) \text{ in } \\
\quad \text{let} \ k = \text{check}(y', \text{pk}(sk_A)) \text{ in } \text{out}(c, \text{senc}(s, k))) \}
\]
Example (6)

→ \{c, s, sk_A, sk_B\},

\{  out(c, pk(sk_B)),

  ! in(c, x \_ pk_B).new k.out(c, aenc(sign(k, sk_A), x \_ pk_B)).

  in(c, x).let s = sdec(x, k) in 0,

  ! in(c, y).let y' = adec(y, sk_B) in

  let k = check(y', pk(sk_A)) in out(c, senc(s, k)))\}
Example (6)

\[ \rightarrow \{ c, s, sk_A, sk_B \}, \]

\[ \{ \quad \text{out}(c, \text{pk}(sk_B)), \]

\[ \quad ! \text{in}(c, x_{_B}). \text{new } k. \text{out}(c, \text{aenc}(\text{sign}(k, sk_A), x_{_B})). \]

\[ \quad \text{in}(c, x). \text{let } s = \text{sdec}(x, k) \text{ in } 0, \]

\[ \quad ! \text{in}(c, y). \text{let } y' = \text{adec}(y, sk_B) \text{ in } \]

\[ \quad \text{let } k = \text{check}(y', \text{pk}(sk_A)) \text{ in } \text{out}(c, \text{senc}(s, k)) \} \]
Example (7)

\[
\to \{c, s, sk_A, sk_B\},
\]

\[
\{ \text{out}(c, \text{pk}(sk_B)), \\
\text{in}(c, x \_ pk_B).\text{new} \ k. \text{out}(c, \text{aenc}(\text{sign}(k, sk_A), x \_ pk_B)). \\\n\text{in}(c, x).\text{let} \ s = \text{sdec}(x, k) \text{ in} \ 0, \\
! \text{in}(c, x \_ pk_B).\ldots, \\
! \text{in}(c, y).\text{let} \ y' = \text{aenc}(y, sk_B) \text{ in} \\\n\text{let} \ k = \text{check}(y', \text{pk}(sk_A)) \text{ in} \text{out}(c, \text{senc}(s, k)))\}
\]
Example (7)

\[ \rightarrow \{c, s, sk_A, sk_B\}, \]

\[ \{ \]

\[ \text{out}(c, \text{pk}(sk_B)), \]

\[ \text{in}(c, x \_ pk_B).\text{new } k.\text{out}(c, \text{aenc}(\text{sign}(k, sk_A), x \_ pk_B)). \]

\[ \text{in}(c, x)\text{.let } s = \text{sdec}(x, k) \text{ in } 0, \]

\[ ! \text{in}(c, x \_ pk_B).\.\.\., \]

\[ ! \text{in}(c, y)\text{.let } y' = \text{adec}(y, sk_B) \text{ in } \]

\[ \text{let } k = \text{check}(y', \text{pk}(sk_A)) \text{ in } \text{out}(c, \text{senc}(s, k))) \}\]
→ \{c, s, sk_A, sk_B\},

\{ new k.out(c, aenc(sign(k, sk_A), pk(sk_B))).
  in(c, x).let s = sdec(x, k) in 0,

! in(c, x\_pk_B)\.\.\. ,

! in(c, y).let y' = adec(y, sk_B) in
  let k = check(y', pk(sk_A)) in out(c, senc(s, k)))\}
Example (8)

\[ \rightarrow \{ c, s, sk_A, sk_B \}, \]

\{ new \ k. \]
\begin{align*}
& \text{out}(c, \text{aenc}(\text{sign}(k, sk_A), \text{pk}(sk_B))). \\
& \text{in}(c, x). \text{let } s = \text{sdec}(x, k) \text{ in } 0, \\
& ! \text{in}(c, x.pk_B). . . . , \\
& ! \text{in}(c, y). \text{let } y' = \text{adec}(y, sk_B) \text{ in} \\
& \text{let } k = \text{check}(y', \text{pk}(sk_A)) \text{ in } \text{out}(c, \text{senc}(s, k))) \}
\]
Example (9)

→ \{c, s, sk_A, sk_B, k'\},

\{
  \text{out}(c, \text{aenc}(\text{sign}(k', sk_A), \text{pk}(sk_B))).
  \text{in}(c, x).\text{let } s = \text{sdec}(x, k') \text{ in } 0,
  ! \text{in}(c, x_pk_B). . . .,
  ! \text{in}(c, y).\text{let } y' = \text{adec}(y, sk_B) \text{ in }
  \text{let } k = \text{check}(y', \text{pk}(sk_A)) \text{ in } \text{out}(c, \text{senc}(s, k))\}

Example (9)

\[ \rightarrow \{ c, s, \text{sk}_A, \text{sk}_B, k' \}, \]

\[
\{ \text{out}(c, \text{aenc} (\text{sign}(k', \text{sk}_A), \text{pk}(\text{sk}_B))).
\text{in}(c, x).\text{let } s = \text{sdec}(x, k') \text{ in } 0, \\
! \text{in}(c, x_{-pk_B}). . . . , \\
! \text{in}(c, y).\text{let } y' = \text{adec}(y, \text{sk}_B) \text{ in } \text{let } k = \text{check}(y', \text{pk}(\text{sk}_A)) \text{ in out}(c, \text{senc}(s, k))) \}
\]
\[ \rightarrow^* \{ c, s, sk_A, sk_B, k' \}, \]

\[
\{ \begin{array}{l}
\text{in}(c, x). \text{let } s = \text{sdec}(x, k') \text{ in } 0,
\text{! in}(c, x \_ pk_B). . . . ,
\text{let } y' = \text{adec}(\text{aenc}(\text{sign}(k', sk_A), \text{pk}(sk_B)), sk_B) \text{ in }
\text{let } k = \text{check}(y', \text{pk}(sk_A)) \text{ in } \text{out}(c, \text{senc}(s, k))
\text{! in}(c, y). . . . \} \]
\[ \rightarrow^* \{ c, s, sk_A, sk_B, k' \}, \]

\[ \{ \quad \text{in}(c, x), \text{let } s = sdec(x, k') \text{ in } 0, \]

\[ \text{! in}(c, x_p k_B), \ldots, \]

\[ \text{let } y' = \text{adec(aenc(sign(k', sk_A), pk(sk_B)), sk_B)} \text{ in} \]

\[ \text{let } k = \text{check}(y', \text{pk}(sk_A)) \text{ in} \text{out}(c, \text{senc}(s, k)) \]

\[ \text{! in}(c, y), \ldots \} \]
Example (11)

\[ \rightarrow^* \{ c, s, sk_A, sk_B, k' \}, \]

\[
\{ \text{in}(c, x).\text{let } s = \text{sdec}(x, k') \text{ in } 0, \\
\quad ! \text{in}(c, x_{pk_B})\ldots, \\
\quad \text{out}(c, \text{senc}(s, k'))))) \\
\quad ! \text{in}(c, y)\ldots \}
\]
Example (11)

\[
\rightarrow^* \{c, s, sk_A, sk_B, k'\}, \\
\{ \text{in}(c, x) \text{. let } s = \text{sdec}(x, k') \text{ in } 0, \\
\text{! in}(c, x_{-}pk_B)\ldots, \\
\text{out}(c, \text{senc}(s, k'))) \\
\text{! in}(c, y)\ldots \} 
\]
Example (12)

\[
\rightarrow \{c, s, sk_A, sk_B, k'\},
\]

\[
\{ \text{let } s = sdec(senc(s, k'), k') \text{ in } 0, \\
\quad ! \text{ in}(c, x_pk_B)\ldots, \\
\quad ! \text{ in}(c, y)\ldots \}
\]
Example (12)

\[ \rightarrow \{ c, s, sk_A, sk_B, k' \}, \]

\[ \{ \text{let } s = \text{sdec(senc(s, k'), k')} \text{ in } 0, \]

\[ ! \text{in}(c, x_{pk_B}). . . . , \]

\[ ! \text{in}(c, y). . . . } \]
Comparison between the two semantics

The first semantics
- is more standard (comes from the original semantics of the pi calculus)
- makes it easier to add a context around an existing process (see definition of observational equivalence later)

The second semantics
- directs the reduction more precisely
- makes a minimal use of renaming (for restrictions only)

Except when mentioned explicitly, I will rely on the second semantics.
Adversary

The protocol is executed in parallel with an adversary. The adversary can be any process. $S = \text{finite set of names (initial knowledge of the adversary)}$.

**Definition**

The closed process $Q$ is an $S$-adversary $\iff \text{fn}(Q) \subseteq S$. 

Bruno Blanchet (Inria)
Secrecy

Intuitive definition
The secret $M$ cannot be output on a public channel

Definition
A trace $T = E_0, P_0 \rightarrow^* E', P'$ outputs $M$ if and only if $T$ contains a reduction $E, P \cup \{ \text{out}(c, M).Q, \text{in}(c, x).P \} \rightarrow E, P \cup \{ Q, P\{M/x\} \}$ for some $E, P, x, P, Q, c \in S$.

Definition
The closed process $P$ preserves the secrecy of $M$ from $S \iff \forall S$-adversary $Q$, $\forall T = \text{fn}(P) \cup S, \{ P, Q \} \rightarrow^* E', P'$, $T$ does not output $M$. 
Authenticity means:

if $A$ thinks he talks to $B$
then he really talks to $B$.

Authenticity can be defined by correspondence assertions [Woo and Lam, Oakland’93]:

If $A$ executes $e_A(B)$ ($A$ thinks he talks to $B$),
then $B$ must have executed $e_B(A)$ ($B$ has started a run with $A$).
Correspondences: events

Events record that some program point has been reached, with certain values of the variables.

Syntax:

\[ P, Q ::= \text{processes} \]
\[ \quad \ldots \]
\[ \text{event}(M).P \]

event

Semantics:

\[ \mathcal{E}, \{ \text{event}(M).P \} \cup \mathcal{P} \rightarrow \mathcal{E}, \{ P \} \cup \mathcal{P} \]  

(\text{Red Event})

An $S$-adversary does not contain events.

Definition

A trace $\mathcal{T} = \mathcal{E}_0, \mathcal{P}_0 \rightarrow^* \mathcal{E}', \mathcal{P}'$ executes $\text{event}(M)$ if and only if $\mathcal{T}$ contains a reduction $\mathcal{E}, \mathcal{P} \cup \{ \text{event}(M).P \} \rightarrow \mathcal{E}, \mathcal{P} \cup \{ P \}$ for some $\mathcal{E}, \mathcal{P}, P$. 
Non-injective correspondences

Intuitive definition

If event($M$) has been executed
then event($M_1$), . . . event($M_n$) have been executed

Definition

The closed process $P_0$ satisfies the correspondence

\[
\text{event}(M) \leadsto \bigwedge_{k=1}^{l} \text{event}(M_k)
\]

with respect to $S$-adversaries if and only if, for any $S$-adversary $Q$,
for any $\mathcal{E}_0$ containing $\text{fn}(P_0) \cup S \cup \text{fn}(M) \cup \bigcup_k \text{fn}(M_k)$,
for any substitution $\sigma$, for any trace $\mathcal{T} = \mathcal{E}_0, \{P_0, Q\} \rightarrow^* \mathcal{E}', \mathcal{P}'$,
if $\mathcal{T}$ executes $\sigma \text{event}(M)$,
then there exists $\sigma'$ such that $\sigma' M = \sigma M$ and,
for all $k \in \{1, \ldots, l\}$, $\mathcal{T}$ executes $\text{event}(\sigma' M_k)$ as well.
Intuitive definition

Each execution of event($M$) corresponds to distinct executions of event($M_1$), $\ldots$, event($M_n$)

Definition

The event event($M$) is executed at step $\tau$ in a trace $T = \mathcal{E}_0, \mathcal{P}_0 \rightarrow^* \mathcal{E}', \mathcal{P}'$ if and only if the $\tau$-th reduction of $T$ is of the form $\mathcal{E}, \mathcal{P} \cup \{\text{event}(M).\mathcal{P}\} \rightarrow \mathcal{E}, \mathcal{P} \cup \{\mathcal{P}\}$ for some $\mathcal{E}, \mathcal{P}, \mathcal{P}$. 
Injective correspondences

**Definition**

The closed process $P_0$ satisfies the correspondence

$$\text{event}(M) \sim \bigwedge_{k=1}^{l} \text{inj} \text{event}(M_k)$$

with respect to $S$-adversaries if and only if, for any $S$-adversary $Q$, for any $\mathcal{E}_0$ containing $\text{fn}(P_0) \cup S \cup \text{fn}(M) \cup \bigcup_k \text{fn}(M_k)$, for any trace $\mathcal{T} = \mathcal{E}_0, \{P_0, Q\} \rightarrow^* \mathcal{E}', \mathcal{P}'$, there exist partial injective functions $\phi_k$ from steps in $\mathcal{T}$ to steps in $\mathcal{T}$ such that for all $\tau$, if $\text{event}(\sigma M)$ is executed at step $\tau$ in $\mathcal{T}$ for some $\sigma$, then there exists $\sigma'$ such that $\sigma'M = \sigma M$ and, for all $k \in \{1, \ldots, l\}$, $\text{event}(\sigma'M_k)$ is executed at step $\phi_k(\tau)$ in $\mathcal{T}$. 

Bruno Blanchet (Inria)
Example (simplified Woo-Lam public key)

Message 1. $A \rightarrow B : \ pk_A$
Message 2. $B \rightarrow A : \ b \quad b \ fresh$
Message 3. $A \rightarrow B : \ \{pk_A, pk_B, b\}_{sk_A}$

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

$(A)$ \quad ! \ \text{in}$(c, \_pk_B$).\text{event}($e_A(\_pk_B)$).out$(c, pk_A$).\text{in}$(c, \_b$).
\quad \text{out}$(c, \text{sign}((pk_A, \_pk_B, \_b), sk_A))$

$(B)$ \quad \parallel \quad ! \ \text{in}$(c, \_pk_A$).new $b$.out$(c, b$).\text{in}$(c, m$).
\quad \text{if} \ (\_pk_A, pk_B, b) = \text{check}(m, \_pk_A) \ \text{then}
\quad \text{if} \ \_pk_A = pk_A \ \text{then} \ \text{event}($e_B(pk_B)$)
Demo: Woo-Lam public-key protocol

- examplesnd/demosimp/piwoolampk
Observational equivalence

Intuitive definition

Two processes are observationally equivalent when an adversary cannot distinguish them.

Using the first semantics.

An evaluation context $C[\ldots]$ is a context $C$ that contains only restrictions and parallel compositions above the $[\ldots]$.

$P \Downarrow m$ if and only if $P \rightarrow^\ast \equiv C[\text{out}(m, N).R]$ where $C$ is an evaluation context that does not bind $m$.

Observational equivalence $\approx$ is the largest symmetric relation $\mathcal{R}$ on closed processes such that $P \mathcal{R} Q$ implies

1. if $P \Downarrow m$ then $Q \Downarrow m$;
2. if $P \rightarrow^\ast P'$ then $Q \rightarrow^\ast Q'$ and $P' \mathcal{R} Q'$ for some $Q'$;
3. $C[P] \mathcal{R} C[Q]$ for all evaluation contexts $C$. 
**Strong secrecy**

**Intuitive definition**

Strong secrecy (or non-interference) is a formalization of secrecy: data $D$ are secret when the adversary cannot see a change of value of these data.

**Definition**

$P$ preserves the secrecy of its free variables if and only if for all closed substitutions $\theta$ and $\theta'$ such that $\text{dom}(\theta) = \text{dom}(\theta') = \text{fv}(P)$, $P\theta \approx P\theta'$.

(The substitution avoids name captures by first alpha renaming $P$ if necessary.)

Strong secrecy detects flows of partial information and implicit flows.
Usage of observational equivalence

- Secrecy of values among a small set:
  \[ P(\text{true}) \approx P(\text{false}) \]
- Satisfaction of a specification
  \[ P \approx P_{\text{spec}} \]

Authenticity can also be specified in this way.
Several variants of the applied pi calculus

- Presented variant [Abadi, Blanchet, POPL’02 and JACM’05]. Correspondences [Blanchet, SAS’02 and JCS’09]
- The spi-calculus [Abadi, Gordon, I&C, 1999]
- The applied pi calculus [Abadi, Fournet, POPL’01]
  Very powerful, thanks to equational theories
- A calculus for asymmetric communication [Abadi, Blanchet, FoSSaCS’01 and TCS’03]
Overview

1. A dialect of the applied pi calculus
2. Undecidability
3. The Horn clause representation
4. The resolution algorithm
5. Experimental results
6. Formal translation from the applied pi calculus
7. Extension to correspondences
8. Extension to observational equivalence
9. Conclusion
Secrecy is undecidable from a unbounded number of sessions.

**Theorem**

*Given a process $P_0$, a set of names $S$, and a term $M$, the question of whether $P_0$ preserves the secrecy of $M$ from $S$ is undecidable.*
To give an idea of the proof, we encode a Post correspondence problem.

The following problem is undecidable:

1. **Input:** two finite sequences $v_1, \ldots, v_n$, $w_1, \ldots w_n \in \{a, b\}^*$.
2. **Question:** are there a $k \geq 1$ and a sequence of indices $i_1, \ldots, i_k \in [1, n]$ such that $v_{i_1} \ldots v_{i_k} = w_{i_1} \ldots w_{i_k}$?
For any word $u \in \{a, b\}^*$, we define inductively a function $\tilde{u}$ from terms to terms, as follows:

1. If $u$ is the empty word, then $\tilde{u}$ is the identity.
2. $a.u(t) = (a, \tilde{u}(t))$
   $b.u(t) = (b, \tilde{u}(t))$

We define:

1. $P_A = \text{out}(c, \text{senc}((\tilde{v}_1(0), \tilde{w}_1(0)), k_{AB})) \ldots \text{out}(c, \text{senc}((\tilde{v}_n(0), \tilde{w}_n(0)), k_{AB}))$
2. $P_B = \text{in}(c, y).\text{let } (x, x') = \text{sdec}(y, k_{AB}) \text{ in } \text{out}(c, \text{senc}((\tilde{v}_1(x), \tilde{w}_1(x')), k_{AB})) \ldots \text{out}(c, \text{senc}((\tilde{v}_n(x), \tilde{w}_n(x')), k_{AB}))$
3. $P_C = \text{in}(c, y).\text{let } (x, x') = \text{sdec}(y, k_{AB}) \text{ in if } x = x' \text{ then } \text{out}(c, k_{AB})$
4. $P_0 = P_A \parallel \lnot P_B \parallel P_C$

$P_0$ preserves the secrecy of $k_{AB}$ from $\{c\}$ if and only if the Post correspondence problem has no solution.

The attacker does not have much to do: only select one output of $P_A$ or $P_B$ and send it to the next instance of $P_B$ or to $P_C$. 
Overview

1. A dialect of the applied pi calculus
2. Undecidability
3. The Horn clause representation
4. The resolution algorithm
5. Experimental results
6. Formal translation from the applied pi calculus
7. Extension to correspondences
8. Extension to observational equivalence
9. Conclusion
ProVerif is a verifier for cryptographic protocols

- Fully automatic
- For an unbounded number of sessions and an unbounded message size
  - Undecidable problem $\Rightarrow$ need for abstractions
- Handles many cryptographic primitives: shared- and public-key encryption, signatures, hash functions, Diffie-Hellman key agreements, ...
- Proves various properties: secrecy, correspondences, equivalences
- Efficient
Our solution

Two ideas (extending [Weidenbach, CADE’99]):

- a simple abstract representation of these protocols, by a set of Horn clauses;
- an efficient solving algorithm to find which facts can be derived from these clauses.

Using this, we can prove secrecy properties of protocols, or exhibit attacks showing why a message is not secret.
Protocol representation

- **Messages** $\mapsto$ **patterns**
  
  $$ p ::= x \mid f(p_1, \ldots, p_n) \mid k[p_1, \ldots, p_n] \mid \text{aenc}(c_0, \text{pk}(sk_A)). $$

- **Properties** $\mapsto$ **facts**
  
  $$ F ::= \text{attacker}(p). $$

- **Protocol, attacker** $\mapsto$ **Horn clauses**
  
  $$ F_1 \land \ldots \land F_n \Rightarrow F \land \text{attacker}(m) \land \text{attacker}(pk) \Rightarrow \text{attacker}(\text{aenc}(m, pk)). $$
Public-key encryption:

- Encryption $\text{aenc}(m, pk)$.
  \[
  \text{attacker}(m) \land \text{attacker}(pk) \Rightarrow \text{attacker}(\text{aenc}(m, pk))
  \]

- Public key generation $\text{pk}(sk)$.
  (builds a public key from a secret key)
  \[
  \text{attacker}(sk) \Rightarrow \text{attacker}(\text{pk}(sk))
  \]

- Decryption $\text{adec}(\text{aenc}(m, \text{pk}(sk)), sk) \rightarrow m$.
  \[
  \text{attacker}(\text{aenc}(m, \text{pk}(sk))) \land \text{attacker}(sk) \Rightarrow \text{attacker}(m)
  \]
General treatment of primitives

- **Constructors** \( f(M_1, \ldots, M_n) \)
  \[ \text{attacker}(x_1) \land \ldots \land \text{attacker}(x_n) \Rightarrow \text{attacker}(f(x_1, \ldots, x_n)) \]

- **Destructors** \( g(M_1, \ldots, M_n) \rightarrow M \)
  \[ \text{attacker}(M_1) \land \ldots \land \text{attacker}(M_n) \Rightarrow \text{attacker}(M) \]

(There may be several rewrite rules defining a function.)

**Exercise**

Give clauses for shared-key encryption and signatures
Clauses that represent the initial knowledge of the adversary:

\[ \text{attacker}(p) \]

if the adversary knows \( p \).

**Example**

For the Denning-Sacco protocol:

\[ \text{attacker}(pk(sk_A[])) \]
\[ \text{attacker}(pk(sk_B[])) \]
Normally, fresh names are created each time the protocol is run. Here, we only distinguish two names when they are created after receiving different messages.

Each name \( k \) becomes a function of the messages previously received:

\[
  k[p_1, \ldots, p_n].
\]

(Skolemisation)

These functions can only be applied by the principal that creates the name, not by the attacker.

The attacker can create his own fresh names: \( \text{attacker}(b[]) \).
Denning-Sacco protocol

- **A → B**: \( \{ \{ k \}_{sk_A} \}_{pk_B} \) \( k \) fresh

  A talks with any principal represented by its public key \( pk(x) \).
  A sends to it the message \( \{ \{ k \}_{sk_A} \}_{pk(x)} \).

  \( \text{attacker}(pk(x)) \Rightarrow \text{attacker}(\text{aenc}(\text{sign}(k[pk(x)], sk_A[]), pk(x))) \).  

- **B → A**: \( \{ s \} k \)

  B has received a message \( \{ \{ y \}_{sk_A} \}_{pk_B} \).
  B sends \( \{ s \} y \).

  \( \text{attacker}(\text{aenc}(\text{sign}(y, sk_A[]), pk(sk_B[]))) \Rightarrow \text{attacker}(\text{senc}(s, y)) \).
If a principal $A$ has received the messages $p_1, \ldots, p_n$ and sends the message $p$,

$$\text{attacker}(p_1) \land \ldots \land \text{attacker}(p_n) \Rightarrow \text{attacker}(p).$$

**Exercise**

Model the Needham-Shroeder public key protocol:

- **Message 1.** $A \rightarrow B \quad \{N_a, A\}_{pk_B} \quad N_a$ fresh
- **Message 2.** $B \rightarrow A \quad \{N_a, N_b\}_{pk_A} \quad N_b$ fresh
- **Message 3.** $A \rightarrow B \quad \{N_b\}_{pk_B}$
Approximations

- The **freshness** of nonces is partially modeled.
- The **number of times** a message appears is ignored, only the fact that it has appeared is taken into account.
- The **state** of the principals is not fully modeled.

These approximations are keys for an efficient verification. Solve the state space explosion problem. No limit on the number of runs of the protocols. ⇒ essential for the **certification** of protocols.
Secrecy criterion

*If attacker(\(p\)) cannot be derived from the clauses, then \(p\) is secret.*

The pattern \(p\) cannot be built by an attacker.
(This is a weaker notion of secrecy than non-interference.)

The solving algorithm will determine whether a given fact can be derived from the clauses.
Overview

1. A dialect of the applied pi calculus
2. Undecidability
3. The Horn clause representation
4. The resolution algorithm
5. Experimental results
6. Formal translation from the applied pi calculus
7. Extension to correspondences
8. Extension to observational equivalence
9. Conclusion
Which resolution algorithm

A standard Prolog system would not terminate:

\[
\text{attacker}(\text{sen}(x, y)) \land \text{attacker}(y) \Rightarrow \text{attacker}(x)
\]

generates bigger and bigger facts by SLD-resolution.

We need a different resolution strategy.
Completion of the clause base, by **resolution with free selection**.

Selection function \( \text{sel}(F_1 \land \ldots \land F_n \Rightarrow F) \subseteq \{F_1, \ldots, F_n\} \).

\[
\text{sel}(F_1 \land \ldots \land F_n \Rightarrow F) = \begin{cases}
\emptyset & \text{if } \forall i \in \{1, \ldots, n\}, F_i = \text{attacker}(x) \\
\{F_i\} & \text{different from } \text{attacker}(x), \\
\text{of maximal size, otherwise}
\end{cases}
\]
Saturation (2)

\[
R = F_1 \land \ldots \land F_n \Rightarrow F \quad R' = F_1' \land \ldots \land F_n' \Rightarrow F'
\]

\[
\sigma F_1 \land \ldots \land \sigma F_n \land \sigma F_2' \land \ldots \land \sigma F_n' \Rightarrow \sigma F'
\]

where \( \sigma \) is the most general unifier of \( F \) and \( F_1' \),
\( \text{sel}(R) = \emptyset \), and \( F_1' \in \text{sel}(R') \).

Starting from an initial set of clauses \( \mathcal{R}_0 \),
perform this resolution step until a fixed point is reached,
eliminating subsumed clauses: \( H \Rightarrow C \) subsumes \( H' \Rightarrow C' \) when there exists \( \sigma \) such that \( \sigma H \subseteq H' \) (multiset inclusion) and \( \sigma C = C' \).

\( \text{saturate}(\mathcal{R}_0) \) is the set of obtained clauses \( R \) such that \( \text{sel}(R) = \emptyset \).
Example of a step:

\[
\begin{align*}
\text{attacker}(x) \land \text{attacker}(y) & \Rightarrow \text{attacker}(\text{aenc}(x, y)) \\
\text{attacker}(\text{aenc}(\text{sign}(z, sk_A[]), \text{pk}(sk_B[]))) & \Rightarrow \text{attacker}(\text{senc}(s, z)) \\
\text{attacker}(\text{sign}(z, sk_A[])) \land \text{attacker}(\text{pk}(sk_B[])) & \Rightarrow \text{attacker}(\text{senc}(s, z))
\end{align*}
\]

**Theorem**

The clauses obtained after saturation $\text{saturate}(R_0)$ prove the same facts as the starting clauses $R_0$. 

Bruno Blanchet (Inria)

ProVerif

Year 2016-17 79 / 158
Proof (1): some notations

If \( R = H \Rightarrow C \), \( R' = F_0 \land H' \Rightarrow C' \), and \( \sigma \) is the most general unifier of \( C \) and \( F_0 \), then \( R \circ_{F_0} R' = \sigma H \land \sigma H' \Rightarrow \sigma C' \).

If \( R \) subsumes \( R' \), \( R \sqsupseteq R' \).

- \( \mathcal{R}_0 \): initial set of clauses.
- \( \mathcal{R}_1 \): set of clauses when the fixpoint is reached.
- \( \mathcal{R}_2 = \text{saturate}(\mathcal{R}_0) = \{ R \in \mathcal{R}_1 \mid \text{sel}(R) = \emptyset \} \)
Proof (2): derivation

Definition (Derivation)

Let $F$ be a closed fact. Let $\mathcal{R}$ be a set of clauses. A derivation of $F$ from $\mathcal{R}$ is a finite tree defined as follows:

1. Its nodes (except the root) are labeled by clauses $R \in \mathcal{R}$.
2. Its edges are labeled by closed facts. (Edges go from a node to each of its sons.)
3. If the tree contains a node labeled by $R$ with one incoming edge labeled by $F_0$ and $n$ outgoing edges labeled by $F_1, \ldots, F_n$, then $R \sqsubseteq \{F_1, \ldots, F_n\} \Rightarrow F_0$.
4. The root has one outgoing edge, labeled by $F$. The unique son of the root is named the subroot.
Lemma (Resolution)

Consider a derivation containing a node $\eta'$, labeled $R'$. Let $F_0$ be a hypothesis of $R'$. Then there exists a son $\eta$ of $\eta'$, labeled $R$, such that the edge $\eta' \rightarrow \eta$ is labeled by an instance of $F_0$, $R \circ_{F_0} R'$ is defined, and one obtains a derivation of the same fact by replacing the nodes $\eta$ and $\eta'$ with a node $\eta''$ labeled $R'' = R \circ_{F_0} R'$. 
Lemma (Subsumption)

If a node $\eta$ of a derivation $D$ is labeled by $R$, then one obtains a derivation $D'$ of the same fact as $D$ by relabeling $\eta$ with a clause $R'$ such that $R' \sqsupseteq R$.

By transitivity of $\supseteq$.
Proof (5): saturation properties

**Lemma (Saturation)**

\( \mathcal{R}_1 \) satisfies the following properties:

1. **For all** \( R \in \mathcal{R}_0 \), **there exists** \( R' \in \mathcal{R}_1 \) **such that** \( R' \sqsubseteq R \);
2. **Let** \( R, R' \in \mathcal{R}_1 \). **Assume** that \( \text{sel}(R) = \emptyset \) **and there exists** \( F_0 \in \text{sel}(R') \) **such that** \( R \circ F_0 R' \) **is defined. In this case, there exists** \( R'' \in \mathcal{R}_1 \), \( R'' \sqsupseteq R \circ F_0 R' \).

1. A clause is removed only when it is subsumed by another one.
2. The fixpoint is reached.
Proof (6): If $F$ is derivable from $\mathcal{R}_0$, then $F$ is derivable from $\text{saturate}(\mathcal{R}_0)$.

Consider a derivation of $F$ from $\mathcal{R}_0$.

For each $R \in \mathcal{R}_0$, there exists $R' \in \mathcal{R}_1$ such that $R' \sqsupseteq R$ (Lemma saturation, Property 1).

We relabel each node labeled by $R \in \mathcal{R}_0 \setminus \mathcal{R}_1$ with $R' \in \mathcal{R}_1$ such that $R' \sqsupseteq R$ (by Lemma subsumption).

Therefore, we obtain a derivation $D$ of $F$ from $\mathcal{R}_1$.

Next, we build a derivation of $F$ from $\mathcal{R}_2$, by transforming $D$. 
Proof (7): If $F$ is derivable from $R_0$, then $F$ is derivable from $\text{saturate}(R_0)$ (continued).

If $D$ contains a clause not in $R_2$, we transform $D$ as follows.
Let $\eta'$ be a lowest node of $D$ labeled by a clause not in $R_2$. All sons of $\eta'$ are labeled by elements of $R_2$.
Let $R'$ be the clause labeling $\eta'$. Since $R' \notin R_2$, $\text{sel}(R') \neq \emptyset$. Take $F_0 \in \text{sel}(R')$.

By Lemma resolution, there exists a son of $\eta$ of $\eta'$ labeled by $R$, such that $R \circ_{F_0} R'$ is defined. Since all sons of $\eta'$ are labeled by elements of $R_2$, $R \in R_2$. Hence $\text{sel}(R) = \emptyset$. So, by Lemma saturation, Property 2, there exists $R'' \in R_1$ such that $R'' \sqsubseteq R \circ_{F_0} R'$.

By Lemma resolution, we replace $\eta$ and $\eta'$ with $\eta''$ labeled by $R \circ_{F_0} R'$. By Lemma subsumption, we replace $R \circ_{F_0} R'$ with $R''$.
The total number of nodes strictly decreases since $\eta$ and $\eta'$ are replaced with a single node $\eta''$. Hence, this replacement process terminates.

Upon termination, we obtain a derivation of $F$ from $R_2$. 
Why it works

The facts attacker($x$) unify with all facts attacker($p$).

If we allow resolution on facts attacker($x$), we will create many clauses.

The choice of the selection function implies that we avoid performing resolution upon attacker($x$).

⇒ This is key to obtaining termination in most cases.
Derivation

\[
\text{solve}_{\mathcal{R}_0}(\text{pred}(p_1, \ldots, p_n)) = \{ H \Rightarrow \text{pred}(p'_1, \ldots, p'_n) \mid H \Rightarrow \text{pred}'(p'_1, \ldots, p'_n) \in \text{saturate}(\mathcal{R}_1) \}, \text{ where pred}' \text{ is a new predicate and } \mathcal{R}_1 = \mathcal{R}_0 \cup \{ \text{pred}(p_1, \ldots, p_n) \Rightarrow \text{pred}'(p_1, \ldots, p_n) \}.
\]

\[
\sigma \text{pred}(p_1, \ldots, p_n) \text{ is derivable from } \mathcal{R}_0 \text{ if and only if } \sigma \text{pred}'(p_1, \ldots, p_n) \text{ is derivable from } \mathcal{R}_1 \text{ if and only if } \sigma \text{pred}'(p_1, \ldots, p_n) \text{ is derivable from } \text{saturate}(\mathcal{R}_1) \text{ (previous theorem) if and only if there exists a clause } H \Rightarrow \text{pred}(p'_1, \ldots, p'_n) \text{ in solve}_{\mathcal{R}_0}(\text{pred}(p_1, \ldots, p_n)) \text{ and a substitution } \sigma' \text{ such that } \sigma' \text{pred}(p'_1, \ldots, p'_n) = \sigma \text{pred}(p_1, \ldots, p_n) \text{ and } \sigma' H \text{ is derivable from } \text{saturate}(\mathcal{R}_1).
\]

If solve_{\mathcal{R}_0}(F) = \emptyset, then no instance of F is derivable from \mathcal{R}_0.

Technique similar to the ordered resolution with selection [Weidenbach, CADE’99].
Optimizations

- Elimination of tautologies
- Elimination of duplicate hypotheses
- Elimination of hypotheses $\text{attacker}(x)$ when $x$ does not appear elsewhere.

Tuples

- Secrecy assumptions: use conjectures to prune the search space.
The saturation algorithm does not always terminate, but we have proved that it terminates for tagged protocols. That is, when each encryption, signature, ... is distinguished from others by a constant tag $c_i$

$$\{c_i, M_1, \ldots, M_n\}_K$$

- Large class of protocols
- Easy to add tags
- Good design practice

[Blanchet, Podelski, Fossacs’03]
Enforcing termination for all cases

Termination can be enforced by additional approximations.

Example: approximate clauses with clauses in decidable class $\mathcal{H}_1$. [Nielson, Nielson, Seidel, SAS’02; Goubault-Larrecq, JFLA’04]

$\mathcal{H}_1 = \text{clauses whose conclusion is } P(f(x_1, \ldots, x_n)), \text{ with distinct variables } x_1, \ldots, x_n.$

\[
\begin{align*}
H \Rightarrow P(f(p_1, \ldots, p_n)) & \quad p_1, \ldots, p_n \text{ are not all variables} \\
Q_1(x_1), \ldots, Q_n(x_n) \Rightarrow P(f(x_1, \ldots, x_n)) & \quad H \Rightarrow Q_i(p_i) \\
H \Rightarrow P(f(x_1, \ldots, x_i, \ldots, x_i, \ldots, x_n)) & \\
H, H\{x/x_i\} \Rightarrow P(f(x_1, \ldots, x_i, \ldots, x, \ldots, x_n))
\end{align*}
\]
Termination

- Ordered resolution with factorization and splitting
  [Comon, Cortier, 2003]
  Terminates on clauses with at most one variable.
  Protocols which blindly copy at most one term.
- Decision procedure for a class of tagged protocols
  without blind copies.
  [Ramanujam, Suresh, 2003]
Overview

1. A dialect of the applied pi calculus
2. Undecidability
3. The Horn clause representation
4. The resolution algorithm
5. **Experimental results**
6. Formal translation from the applied pi calculus
7. Extension to correspondences
8. Extension to observational equivalence
9. Conclusion
# Experimental results

Pentium III, 1 GHz.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Result</th>
<th>ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Needham-Schroeder public key</td>
<td>Attack [Lowe]</td>
<td>10</td>
</tr>
<tr>
<td>Needham-Schroeder public key corrected</td>
<td>Secure</td>
<td>7</td>
</tr>
<tr>
<td>Needham-Schroeder shared key</td>
<td>Attack [Denning]</td>
<td>52</td>
</tr>
<tr>
<td>Needham-Schroeder shared key corrected</td>
<td>Secure</td>
<td>109</td>
</tr>
<tr>
<td>Denning-Sacco</td>
<td>Attack [AN]</td>
<td>6</td>
</tr>
<tr>
<td>Denning-Sacco corrected</td>
<td>Secure</td>
<td>7</td>
</tr>
<tr>
<td>Otway-Rees</td>
<td>Secure</td>
<td>10</td>
</tr>
<tr>
<td>Otway-Rees, variant of Paulson98</td>
<td>Attack [Paulson]</td>
<td>12</td>
</tr>
<tr>
<td>Yahalom</td>
<td>Secure</td>
<td>10</td>
</tr>
<tr>
<td>Simpler Yahalom</td>
<td>Secure</td>
<td>11</td>
</tr>
<tr>
<td>Main mode of Skeme</td>
<td>Secure</td>
<td>23</td>
</tr>
</tbody>
</table>
Overview

1. A dialect of the applied pi calculus
2. Undecidability
3. The Horn clause representation
4. The resolution algorithm
5. Experimental results
6. **Formal translation from the applied pi calculus**
7. Extension to correspondences
8. Extension to observational equivalence
9. Conclusion
We consider a protocol $P_0$, executed in the presence of an $S$-adversary.

A protocol is translated into a set of Horn clauses using 2 predicates:

- $\text{attacker}(p)$: the adversary may have $p$
- $\text{mess}(p, p')$: the message $p'$ may be sent on the channel $p$
Translation: attacker clauses

For each $a \in S$, attacker($a[]$)  \hspace{2cm} (Init)

attacker($b[]$) where $b$ does not occur in $P_0$ \hspace{1cm} (Name gen)

For each constructor $f$ of arity $n$,
\[
\text{attacker}(x_1) \land \ldots \land \text{attacker}(x_n) \Rightarrow \text{attacker}(f(x_1, \ldots, x_n))
\]  \hspace{1cm} (Constr)

For each destructor $g$, for each rewrite rule $g(M_1, \ldots, M_n) \rightarrow M$,
\[
\text{attacker}(M_1) \land \ldots \land \text{attacker}(M_n) \Rightarrow \text{attacker}(M)
\]  \hspace{1cm} (Destr)

mess($x, y$) $\land$ attacker($x$) $\Rightarrow$ attacker($y$)  \hspace{1cm} (Listen)

attacker($x$) $\land$ attacker($y$) $\Rightarrow$ mess($x, y$) \hspace{1cm} (Send)
Translation: protocol clauses

\( \rho \): environment (variables, names \( \mapsto \) patterns)
\( h \): hypothesis (messages that must be received before reaching the current process)

- \([0]_{\rho h} = \emptyset\)
- \([P \parallel Q]_{\rho h} = [P]_{\rho h} \cup [Q]_{\rho h}\)
- \([!P]_{\rho h} = [P]_{\rho h}\)
- \([\text{new } a.P]_{\rho h} = [P](\rho[a \mapsto a[p_1, \ldots, p_n]])h\)
  when \( h = \text{mess}(c_1, p_1) \land \ldots \land \text{mess}(c_n, p_n). \)
Translation: protocol clauses (continued)

- $[[\text{in}(M, x).P]]\rho h = [[P]](\rho[x \mapsto x']) (h \land \text{mess}(\rho(M), x'))$
  - $x'$ new variable
- $[[\text{out}(M, N).P]]\rho h = [[P]]\rho h \cup \{h \Rightarrow \text{mess}(\rho(M), \rho(N))\}$
- $[[\text{if } M = N \text{ then } P \text{ else } Q]]\rho h = [[P]](\sigma\rho)(\sigma h) \cup [[Q]]\rho h$
  where $\sigma$ is the most general unifier of $\rho(M)$ and $\rho(N)$.
- $[[\text{let } x = g(M_1, \ldots, M_n) \text{ in } P \text{ else } Q]]\rho h =$
  $\cup \{ [[P]]((\sigma\rho)[x \mapsto \sigma' p'])(\sigma h) \mid g(p'_1, \ldots, p'_n) \rightarrow p' \text{ is a rewrite rule of } g$
  and $(\sigma, \sigma')$ is a most general pair of substitutions such that $\sigma\rho(M_1) = \sigma' p'_1, \ldots, \sigma\rho(M_n) = \sigma' p'_n\} \cup [[Q]]\rho h$. 
Example: Denning-Sacco protocol

Message 1. \( A \rightarrow B : \ \{{}\{k\}_{sk_A}\}_{pk_B} \quad k \ \text{fresh} \)

Message 2. \( B \rightarrow A : \ \{s\}_k \)

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

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

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

\[ P_0 = \text{new } sk_A.\text{new } sk_B.\text{let } pk_A = pk(sk_A) \text{ in let } pk_B = pk(sk_B) \text{ in out}(c, pk_A).\text{out}(c, pk_B).(!P_A \parallel !P_B) \]

\[ \llbracket P_0 \rrbracket \{ c \mapsto c[ ] \} \emptyset \]
Example: protocol clauses

\[ P_0 = \text{new } sk_A.\text{new } sk_B.\text{let } pk_A = \text{pk}(sk_A) \text{ in let } pk_B = \text{pk}(sk_B) \text{ in out}(c, pk_A).\text{out}(c, pk_B).(!P_A \parallel !P_B) \]

\[
\mathbb{P}_0 \{ c \mapsto c[] \} \emptyset = \mathbb{P} \{ \text{let } \ldots \{ c \mapsto c[], \text{sk}_A \mapsto \text{sk}_A[], \text{sk}_B \mapsto \text{sk}_B[] \} \emptyset
\]
Example: protocol clauses

\[P_0 = \text{new } sk_A.\text{new } sk_B.\text{let } pk_A = \text{pk}(sk_A) \text{ in let } pk_B = \text{pk}(sk_B) \text{ in out}(c, pk_A).\text{out}(c, pk_B).(!P_A || !P_B)\]

\([P_0]\{c \mapsto c[]\}\emptyset
\]

\[\quad= [\text{let } \ldots]\{c \mapsto c[], sk_A \mapsto sk_A[], sk_B \mapsto sk_B[]\}\emptyset\]

\[\quad= [\text{out}(c, pk_A) \ldots]\rho_0\emptyset\]

\[\rho_0 = \{c \mapsto c[], sk_A \mapsto sk_A[], sk_B \mapsto sk_B[], pk_A \mapsto \text{pk}(sk_A[]), pk_B \mapsto \text{pk}(sk_B[])\}\]
Example: protocol clauses

\[ P_0 = \text{new } sk_A.\text{new } sk_B.\text{let } pk_A = \text{pk}(sk_A) \text{ in let } pk_B = \text{pk}(sk_B) \text{ in out}(c, pk_A).\text{out}(c, pk_B).(!P_A \parallel !P_B) \]

\[ \llbracket P_0 \rrbracket \{ c \mapsto c[\] \} \emptyset \]
\[ = \llbracket \text{let } \ldots \rrbracket \{ c \mapsto c[\], sk_A \mapsto sk_A[\], sk_B \mapsto sk_B[\] \} \emptyset \]
\[ = \llbracket \text{out}(c, pk_A) \ldots \rrbracket \rho_0 \emptyset \]
\[ \rho_0 = \{ c \mapsto c[\], sk_A \mapsto sk_A[\], sk_B \mapsto sk_B[\], \]
\[ pk_A \mapsto \text{pk}(sk_A[\]), pk_B \mapsto \text{pk}(sk_B[\]) \} \]
\[ = \llbracket !P_A \parallel !P_B \rrbracket \rho_0 \emptyset \]
\[ \bigcup \{ \text{mess}(c[\], \text{pk}(sk_A[\])); \quad \text{comes from } \text{out}(c, pk_A) \}
\[ \text{mess}(c[\], \text{pk}(sk_B[\])) \} \quad \text{comes from } \text{out}(c, pk_B) \]
Example: protocol clauses

\[ P_0 = \text{new } sk_A.\text{new } sk_B.\text{let } pk_A = \text{pk}(sk_A) \text{ in let } pk_B = \text{pk}(sk_B) \text{ in} \]
\[
\text{out}(c, pk_A).\text{out}(c, pk_B).(!P_A \parallel !P_B) \]

\[
\llbracket P_0 \rrbracket \{c \mapsto c[\ ]\} \emptyset
\]
\[
= \llbracket \text{let } \ldots \rrbracket \{c \mapsto c[\ ], sk_A \mapsto sk_A[\ ], sk_B \mapsto sk_B[\ ]\} \emptyset
\]
\[
= \llbracket \text{out}(c, pk_A) \ldots \rrbracket \rho_0 \emptyset
\]
\[
\rho_0 = \{c \mapsto c[\ ], sk_A \mapsto sk_A[\ ], sk_B \mapsto sk_B[\ ],
\]
\[
    pk_A \mapsto \text{pk}(sk_A[\ ]), pk_B \mapsto \text{pk}(sk_B[\ ])
\]
\[
= \llbracket !P_A \parallel !P_B \rrbracket \rho_0 \emptyset
\]
\[
\cup \{\text{mess}(c[\ ], \text{pk}(sk_A[\ ])), \text{comes from out}(c, pk_A)\}
\]
\[
\text{mess}(c[\ ], \text{pk}(sk_B[\ ])), \text{comes from out}(c, pk_B)\}
\]
\[
= \llbracket P_A \rrbracket \rho_0 \emptyset \cup \llbracket P_B \rrbracket \rho_0 \emptyset \cup \{\text{attacker}(\text{pk}(sk_A[\ ])), \text{attacker}(\text{pk}(sk_B[\ ]))\}
\]

\text{attacker}(p) \text{ is equivalent to mess}(c[\ ], p) \text{ when } c \in S, \text{ by (Listen) and (Send).}
Example: protocol clauses (A)

\[ P_A = \text{in}(c, x_{\_pk_B}).\text{new } k.\text{out}(c, \text{aenc}(\text{sign}(k, sk_A), x_{\_pk_B})). \text{in}(c, x).\text{let } s = \text{sdec}(x, k) \text{ in } 0 \]

\[ \llbracket P_A \rrbracket_{\rho_0} \emptyset \]

\[ = \llbracket \text{new } k. \ldots \rrbracket \rho_0[x_{\_pk_B} \mapsto x_{pk_B}] \text{ mess}(c[], x_{pk_B}) \]

\[ = \llbracket \text{out}(c, \text{aenc}(\ldots)) \ldots \rrbracket \rho_0[x_{\_pk_B} \mapsto x_{pk_B}, k \mapsto k[x_{pk_B}]] \text{ mess}(c[], x_{pk_B}) \]

\[ = \llbracket \text{in}(c, x) \ldots \rrbracket \rho_0[x_{\_pk_B} \mapsto x_{pk_B}, k \mapsto k[x_{pk_B}]] \text{ mess}(c[], x_{pk_B}) \]

\[ \cup \{ \text{mess}(c[], x_{pk_B}) \Rightarrow \text{mess}(c[], \text{aenc}(\text{sign}(k[x_{pk_B}], sk_A[]), x_{pk_B})) \} \]

\[ = \{ \text{mess}(c[], x_{pk_B}) \Rightarrow \text{mess}(c[], \text{aenc}(\text{sign}(k[x_{pk_B}], sk_A[]), x_{pk_B})) \} \]
Example: protocol clauses (B)

\[ P_B = \text{in}(c, y). \text{let } y' = \text{adec}(y, sk_B) \text{ in} \]

\[ \text{let } k = \text{check}(y', pk_A) \text{ in out}(c, \text{senc}(s, k)) \]

\[
\begin{array}{l}
\llbracket P_B \rrbracket \rho_0 \emptyset \\
= \llbracket \text{let } y' \ldots \rrbracket \rho_0[y \mapsto y] \text{ mess(c[], y)} \\
= \llbracket \text{let } k \ldots \rrbracket \rho_0[y \mapsto \text{aenc}(y', pk(sk_B[])), y' \mapsto y'] \\
\text{ mess(c[], aenc(y', pk(sk_B[])))} \\
= \llbracket \text{out(c, \ldots)} \rrbracket \rho_0[y \mapsto \text{aenc(sign(k, sk_A[])), pk(sk_B[])), y' \mapsto sign(k, sk_A[]), k \mapsto k] \text{ mess(c[], aenc(sign(k, sk_A[]), pk(sk_B[])))} \\
= \{ \text{mess(c[], aenc(sign(k, sk_A[]), pk(sk_B[]))) } \Rightarrow \text{mess(c[], senc(s, k))} \}
Proof of secrecy

Closed process: $P_0$
Initial knowledge of the adversary: $S$ finite set of names
Clauses for the protocol and the adversary: $\mathcal{R}_{P_0,S}$.

Theorem

*If attacker(s) cannot be derived from $\mathcal{R}_{P_0,S}$, then $P_0$ preserves the secrecy of $s$ from $S$."

Theorem

*If $\text{solve}_{\mathcal{R}_{P_0,S}}(\text{attacker(s)}) = \emptyset$, then $P_0$ preserves the secrecy of $s$ from $S$."

Bruno Blanchet (Inria)
For the Denning-Sacco protocol, attacker(s) is derivable from the clauses.

The derivation corresponds to the description of the known attack.

For the corrected version, attacker(s) is not derivable from the clauses: 
\texttt{s is secret}. 
Demo:

- Denning-Sacco protocol
  - examplesnd/demosimp/pidenning-sacco-orig
  - examplesnd/demosimp/pidenning-sacco-corr-orig

- Needham-Schroeder public-key protocol
  - examplesnd/demosimp/pineedham-orig
  - examplesnd/demosimp/pineedham-corr-orig
We have defined a generic type system for the explained variant of the applied pi calculus.

**Theorem**

A secrecy property can be proved by the Horn clause verifier

\[\Leftrightarrow\]

it can be proved by any instance of the type system.

A tight relation between two superficially different frameworks.
Goal: Establish a shared key between two participants

Message 1. $A \rightarrow B$: $g^{n_0}$, $n_0$ fresh

Message 2. $B \rightarrow A$: $g^{n_1}$, $n_1$ fresh

$A$ computes $k = (g^{n_1})^{n_0}$, $B$ computes $k = (g^{n_0})^{n_1}$. The exponentiation is such that these quantities are equal.

$$(g^{n_1})^{n_0} = (g^{n_0})^{n_1}$$

The exponentiation is computed in a cyclic multiplicative subgroup $G$ of $\mathbb{Z}_p^*$, where $p$ is a prime and $g$ is a generator of $G$. 
Extension to equational theories: Diffie-Hellman example

Simplified version of the secure shell protocol (SSH):

Message 1. \( C \rightarrow S : \ KExDHInit, g^{n_0} \quad n_0 \ \text{fresh} \)

Message 2. \( S \rightarrow C : \ KExDHReply, pk_S, g^{n_1}, \{ h \}_{sk_S} \quad n_1 \ \text{fresh} \)

where \( K = (g^{n_1})^{n_0} = (g^{n_0})^{n_1} \)

and \( h = H((pk_S, g^{n_0}, g^{n_1}, K)) \).

\( K \) and \( h \) are shared secrets between \( C \) (client) and \( S \) (server). They are used to compute encryption keys.
Extension to equational theories: other examples

- XOR: associative, commutative, $\text{xor}(x, x) = 0$, $\text{xor}(x, 0) = x$

- Primitives whose success is not observable
  (for decryption for instance)

  $$s\text{dec}(s\text{enc}(x, y), y) = x$$
  $$s\text{enc}(s\text{dec}(x, y), y) = x$$

- Subtle interactions between primitives
  Example: XOR and crc

  $$\text{crc}(\text{xor}(x, y)) = \text{xor}(\text{crc}(x), \text{crc}(y))$$
We have built algorithms that translate the equations into a set of rewrite rules, which generates enough terms (equal modulo the equational theory). [Blanchet, Abadi, Fournet, JLAP’08]

We have shown that, for each trace with equations, there is a corresponding trace with rewrite rules, and conversely.

Efficient because it avoids unification modulo. (Standard syntactic resolution can still be used.)

Still fairly limited, since it leads to non-termination for many equational theories. (For example, cannot handle theories that contain associativity.)
Equation:

\[(g^x)^y = (g^y)^x\]

is translated into the rewrite rules:

\[g \rightarrow g \quad x^y \rightarrow x^y \quad (g^x)^y \rightarrow (g^y)^x\]

Terms may have several normal forms: applying \(^\) to \(g^x\) and \(y\) yields two normal forms of \((g^x)^y\): \((g^x)^y\) and \((g^y)^x\).
Extension to equational theories: encryption

Equations:

\[
\begin{align*}
\text{sdec}(\text{senc}(x, y), y) &= x \\
\text{senc}(\text{sdec}(x, y), y) &= x
\end{align*}
\]

are translated into the rewrite rules:

\[
\begin{align*}
\text{sdec}(x, y) &\rightarrow \text{sdec}(x, y) \\
\text{sdec}(\text{senc}(x, y), y) &\rightarrow x \\
\text{senc}(x, y) &\rightarrow \text{senc}(x, y) \\
\text{senc}(\text{sdec}(x, y), y) &\rightarrow x
\end{align*}
\]

Each term has a single normal form, irreducible by
\[
\begin{align*}
\text{sdec}(\text{senc}(x, y), y) &\rightarrow x \\
\text{senc}(\text{sdec}(x, y), y) &\rightarrow x
\end{align*}
\]
Unification modulo the equational theory could be used, for example to handle associativity and commutativity.

- Better model of **Diffie-Hellman** (modelling the multiplicative group plus the exponentiation).
  [Meadows, Narendran, WITS’02]
  [Goubault-Larrecq, Roger, Verma, JLAP’04]
- **XOR**
  [Comon, Shmatikov, LICS’03]
  [Chevalier, Küsters, Rusinowitch, Turuani, LICS’03] (Bounded number of sessions)
Some protocols can be broken into phases.

This is formalized by a phase prefix:
The process $\text{phase } n.P$ executes only in phase $n$.

For example, model the compromise of some long-term keys:
- First phase: the protocol runs normally
- Second phase: reveal some long-term keys
Are secrets of the normal sessions still preserved?

Modeled by using predicates $\text{attacker}_n$ and $\text{mess}_n$ for each phase $n$. 
Overview

1. A dialect of the applied pi calculus
2. Undecidability
3. The Horn clause representation
4. The resolution algorithm
5. Experimental results
6. Formal translation from the applied pi calculus
7. Extension to correspondences
8. Extension to observational equivalence
9. Conclusion
Back to the Woo-Lam example

Message 1. \( A \rightarrow B : \ pk_A \)

Message 2. \( B \rightarrow A : \ b \quad b\ fresh \)

Message 3. \( A \rightarrow B : \ \{pk_A, pk_B, b\}_{sk_A} \)

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

\( (A) \quad ! \ \text{in}(c, x_{-pk_B}).\text{event}(e_A(x_{-pk_B})).\text{out}(c, pk_A).\text{in}(c, x_{-b}). \)
\( \quad \text{out}(c, \text{sign}((pk_A, x_{-pk_B}, x_{-b}), sk_A)) \)

\( (B) \quad \parallel \quad ! \ \text{in}(c, x_{-pk_A}).\text{new} \ b.\text{out}(c, b).\text{in}(c, m). \)
\( \quad \text{if} \ (x_{-pk_A}, pk_B, b) = \text{check}(m, x_{-pk_A}) \) then
\( \quad \text{if} \ x_{-pk_A} = pk_A \) then \( \text{event}(e_B(pk_B)) \)
Overview of the proof technique

Our technique *overapproximates* occurrences of events.

Suppose we want to prove a correspondence \( \text{event}(e_1(x)) \leadsto \text{event}(e_2(x)) \).

We can overapproximate occurrences of \( e_1 \):
If the correspondence is proved with \( e_1 \) overapproximated, then the correspondence still holds in the exact semantics.

We extend the technique for secrecy by introducing a fact \( \text{event}(p) \) which means that \( \text{event}(p) \) may have been executed.

If the protocol executes \( \text{event}(p) \) after receiving \( p'_1, \ldots, p'_n \) on channels \( p_1, \ldots, p_n \), we generate

\[
\text{mess}(p_1, p'_1) \land \ldots \land \text{mess}(p_n, p'_n) \Rightarrow \text{event}(p)
\]
We must **not** overapproximate occurrences of $e_2$:
If the correspondence is proved with $e_2$ overapproximated, we are not sure that $e_2$ has been executed in the exact semantics, so the correspondence $\text{event}(e_1(x)) \nsucceq \text{event}(e_2(x))$ may actually not hold.

We fix the exact set $\mathcal{E}$ of allowed events $e_2(p)$, and, in order to show $\text{event}(e_1(x)) \nsucceq \text{event}(e_2(x))$, we check that only events $e_1(p)$ for $p$ such that $e_2(p) \in \mathcal{E}$ can be executed.

If we prove this property for all $\mathcal{E}$, we have proved the desired correspondence.
We introduce a predicate \( m\text{-event} \), such that \( m\text{-event}(e_2(p)) \) is true if and only \( e_2(p) \in E \).

If the protocol outputs \( p' \) on channel \( p \) after executing the event \( e_2(p_0) \) and receiving \( p'_1, \ldots, p'_n \) on channels \( p_1, \ldots, p_n \), we generate

\[
\text{mess}(p_1, p'_1) \land \ldots \land \text{mess}(p_n, p'_n) \land m\text{-event}(e_2(p_0)) \Rightarrow \text{mess}(p, p')
\]

The output of \( p' \) on \( p \) can be executed only when \( m\text{-event}(e_2(p_0)) \) is true, that is, \( e_2(p_0) \in E \).
The resolution will be performed for a fixed but unknown value of $E$.

We have to keep m-event facts with trying to evaluate them.

To do that, we simply never select m-event facts.

Then the result holds for any $E$. 
Difficulty: This reasoning does not work when the correspondence between terms and patterns is not injective. (Otherwise, the true m-event facts will not correspond exactly to the events of the trace.)

With the previous definitions, we do not have injectivity.

To recover injectivity, we need to distinguish names created in different copies of the same process, even after receiving the same messages.

So add one more argument to patterns, a session identifier, that is, a variable that takes a different value in each copy of a process.
Instrumentation

- Add **session identifiers** to replications:

  \[ !P \text{ becomes } !^iP \]

  \[ !^iP \text{ means } P\{\lambda_1/i\} \parallel P\{\lambda_2/i\} \parallel P\{\lambda_3/i\} \parallel \ldots \]

- Add **patterns (types)** to restrictions:

  new \( a.P \) becomes new \( a:p.P \)

  Fresh name \( a \leadsto \text{ function } p = a[x_1, \ldots, x_n, i_1, \ldots, i_{n'}] \) of all variables (in particular inputs and sessions identifiers) bound above \( a \) (skolemization).

→ distinguish bound names created in different sessions.
new $sk_A:sk_A[]$.new $sk_B:sk_B[]$.let $pk_A = \text{pk}(sk_A)$ in
let $pk_B = \text{pk}(sk_B)$ in out$(c, pk_A)$.out$(c, pk_B)$.

(A) $^{i_A}$.in$(c, x\_pk_B)$.event$(e_A(x\_pk_B))$.out$(c, pk_A)$.in$(c, x\_b)$.
\hspace{1cm} out$(c, \text{sign}((pk_A, x\_pk_B, x\_b), sk_A))$

(B) $^{i_B}$.in$(c, x\_pk_A)$.new $b:b[x\_pk_A, i_B]$.out$(c, b)$.in$(c, m)$.
\hspace{1cm} if $(x\_pk_A, pk_B, b) = \text{check}(m, x\_pk_A)$ then
\hspace{1cm} if $x\_pk_A = pk_A$ then event$(e_B(pk_B))$
A protocol is translated into a set of Horn clauses using 4 predicates:

- `attacker(p)` the adversary may have p
- `mess(p, p')` the message p' may be sent on the channel p
- `m-event(p)` the event event(p) must have been executed
- `event(p)` the event event(p) may have executed

Attacker clauses are as before, except that we give an infinite number of names to the attacker: `attacker(b[i])` instead of `attacker(b[])`. 
Translation: protocol clauses

\( \rho \): environment (variables, names \( \mapsto \) patterns)

\( h \): hypothesis (events that must be executed before reaching the current process)

\[
\begin{align*}
\llbracket \text{in}(M, x).P \rrbracket \rho h &= [P](\rho[x \mapsto x'])(h \land \text{mess}(\rho(M), x')) \\
&\text{\( x' \) new variable}
\end{align*}
\]

\[
\begin{align*}
\llbracket \text{out}(M, N).P \rrbracket \rho h &= [P] \rho h \cup \{ h \Rightarrow \text{mess}(\rho(M), \rho(N)) \}
\end{align*}
\]

\[
\begin{align*}
\llbracket \text{event}(M).P \rrbracket \rho h &= [P] \rho (h \land \text{m-event}(\rho(M))) \cup \{ h \Rightarrow \text{event}(\rho(M)) \}
\end{align*}
\]

\[
\begin{align*}
\llbracket !i.P \rrbracket \rho h &= [P](\rho[i \mapsto i'])h \quad \text{\( i' \) new variable}
\end{align*}
\]

\[
\begin{align*}
\llbracket \text{new} \ a : p.P \rrbracket \rho h &= [P](\rho[a \mapsto \rho(p)])h
\end{align*}
\]
Example: protocol clauses (initialization)

Process:

new $sk_A : sk_A[]$.new $sk_B : sk_B[]$.let $pk_A = pk(sk_A)$ in let $pk_B = pk(sk_B)$ in
out($c, pk_A$).out($c, pk_B$)...

Clauses:

attacker($pk(sk_A[])$)

attacker($pk(sk_B[])$)

Note: mess($c, p$) is equivalent to attacker($p$) because $c$ is public
Example: protocol clauses (for $A$)

Process:

$$
(A) \quad !^{i_A} \text{in}(c, x_{-pk_B}).\text{event}(e_A(x_{-pk_B})).\text{out}(c, p k_A).\text{in}(c, x_{-b}).
\text{out}(c, \text{sign}((p k_A, x_{-pk_B}, x_{-b}), s k_A))
$$

Note: clause $\text{attacker}(x_{-pk_B}) \Rightarrow \text{event}(e_A(x_{-pk_B}))$ useless for proving a correspondence $e_B(x) \leadsto e_A(x)$. 
Example: protocol clauses (for $A$)

Process:

$$(A) \quad !^A \text{in}(c, x_{\text{pk}_B}).\text{event}(e_A(x_{\text{pk}_B})).\text{out}(c, \text{pk}_A).\text{in}(c, x_b).\text{out}(c, \text{sign}((\text{pk}_A, x_{\text{pk}_B}, x_b), \text{sk}_A))$$

Note: clause $\text{attacker}(x_{\text{pk}_B}) \Rightarrow \text{event}(e_A(x_{\text{pk}_B}))$ useless for proving a correspondence $e_B(x) \rightsquigarrow e_A(x)$.

Clauses:

$$\text{attacker}(x_{\text{pk}_B}) \land \text{m-event}(e_A(x_{\text{pk}_B})) \Rightarrow \text{attacker}(\text{pk}(\text{sk}_A[[]]))$$
Example: protocol clauses (for A)

Process:

\[(A) \quad !^{i_A} \text{in}(c, x_{-pk_B}).\text{event}(e_A(x_{-pk_B})).\text{out}(c, pk_A).\text{in}(c, x_{-b}).\text{out}(c, \text{sign}((pk_A, x_{-pk_B}, x_{-b}, sk_A)))\]

Note: clause attacker\((x_{-pk_B}) \Rightarrow \text{event}(e_A(x_{-pk_B}))\) useless for proving a correspondence \(e_B(x) \sim e_A(x)\).

Clauses:

\[
\begin{align*}
\text{attacker}(x_{-pk_B}) \land m\text{-event}(e_A(x_{-pk_B})) & \Rightarrow \text{attacker}(pk(sk_A[[]])) \\
\text{attacker}(x_{-pk_B}) \land m\text{-event}(e_A(x_{-pk_B})) \land \text{attacker}(x_{-b}) & \Rightarrow \text{attacker}(\text{sign}((pk(sk_A[[]), x_{-pk_B}, x_{-b}, sk_A[[]])))
\end{align*}
\]
Example: protocol clauses (for $B$)

Process:

\[
(B) \quad \!^{i_B} \text{in}(c, x_{pk_A}).\text{new } b : b[x_{pk_A}, i_B].\text{out}(c, b).\text{in}(c, m).
\]

if $(x_{pk_A}, pk_B, b) = \text{check}(m, x_{pk_A})$ then
if $x_{pk_A} = pk_A$ then event($e_B(pk_B)$)

Clauses:

attacker($x_{pk_A}$) $\Rightarrow$ attacker($b[x_{pk_A}, i_B]$)
Example: protocol clauses (for $B$)

Process:

$$(B) \quad !^i_B \text{in}(c, x_{-}pk_A).\text{new } b : b[x_{-}pk_A, i_B].\text{out}(c, b).\text{in}(c, m).
\text{if } (x_{-}pk_A, pk_B, b) = \text{check}(m, x_{-}pk_A) \text{ then}
\text{if } x_{-}pk_A = pk_A \text{ then } \text{event}(e_B(pk_B))$$

Clauses:

$$\text{attacker}(x_{-}pk_A) \Rightarrow \text{attacker}(b[x_{-}pk_A, i_B])$$
$$\text{attacker}(pk(sk_A[])) \land$$
$$\text{attacker}(\text{sign}((pk(sk_A[]), pk(sk_B[]), b[pk(sk_A[]), i_B]), sk_A[]))$$
$$\Rightarrow \text{event}(e_B(pk(sk_B[])))$$
Exercise

1. Model mutual authentication in the Needham-Schroeder public-key protocol, by correspondences in a applied pi calculus process.

2. Give the corresponding Horn clauses.
Proof of correspondences

Closed process: $P_0$
Instrumentation of $P_0$: $P'_0$
Clauses for the protocol and the adversary: $\mathcal{R}_{P'_0, S}$.
Closed facts defining m-event: $\mathcal{F}_{m\text{-event}}$.

**Theorem**

Consider any trace $\mathcal{T}$ of $P'_0$ in the presence of an $S$-adversary.

Suppose that, if $\text{event}(p)$ executed in $\mathcal{T}$, then $m\text{-event}(p) \in \mathcal{F}_{m\text{-event}}$.

Suppose that $\text{event}(p')$ executed in $\mathcal{T}$.

Then, $\text{event}(p')$ derivable from $\mathcal{R}_{P'_0, S} \cup \mathcal{F}_{m\text{-event}}$. 
The selection function is now:
\[ \text{sel}(F_1 \land \ldots \land F_n \Rightarrow F) = \]
\[
\begin{cases} 
\emptyset & \text{if } \forall i \in \{1, \ldots, n\}, F_i = \text{attacker}(x) \text{ or } \text{m-event}(p) \\
\{F_i\} & \text{different from } \text{attacker}(x) \text{ and } \text{m-event}(p) 
\end{cases}
\]

**Theorem**

*F* can be derived from \( \mathcal{R}_{P_0',S} \cup \mathcal{F}_{m\text{-event}} \) if and only if it can be derived from \( \text{saturate}(\mathcal{R}_{P_0',S}) \cup \mathcal{F}_{m\text{-event}} \).
Proof of correspondences

Closed process: $P_0$
Instrumentation of $P_0$: $P'_0$
Clauses for the protocol and the adversary: $\mathcal{R}_{P'_0, S}$.

**Theorem**

Consider any trace $\mathcal{T}$ of $P'_0$ in the presence of an $S$-adversary. If an instance $\text{event}(p'')$ of $\text{event}(p')$ is executed in $\mathcal{T}$, then there exist a clause $H \Rightarrow C \in \text{solve}_{\mathcal{R}_{P'_0, S}}(\text{event}(p'))$ and a substitution $\sigma$ such that $\text{event}(p'') = \sigma C$ and, for all $m\text{-event}(p) \in \sigma H$, event$(p)$ is executed in $\mathcal{T}$. 
Proof of correspondences

Closed process: $P_0$
Instrumentation of $P_0$: $P'_0$
Clauses for the protocol and the adversary: $\mathcal{R}_{P'_0,S}$.
Terms $M, M_k$, with corresponding patterns $p, p_k$ obtained by replacing $a$ with $a[]$ in $M, M_k$.

Theorem

Suppose that, for all clauses $R \in \text{solve}_{\mathcal{R}_{P'_0,S}}(\text{event}(p))$, there exist $\sigma$ and $H$ such that $R = H \land \bigwedge_{k=1}^{l} \text{m-event}(\sigma p_k) \Rightarrow \text{event}(\sigma p)$.
Then $P_0$ satisfies the correspondence $\text{event}(M) \rightsquigarrow \bigwedge_{k=1}^{l} \text{event}(M_k)$. 
The only clause \( R \in \text{solve}_{R_0, S}(\text{event}(e_B(x))) \) is:

\[
m\text{-event}(e_A(pk(sk_B[]))) \Rightarrow \text{event}(e_B(pk(sk_B[])))
\]

so we have proved \( \text{event}(e_B(x)) \leadsto \text{event}(e_A(x)) \).
Suppose that we want to prove a correspondence
\[ \text{event}(e_B(x)) \rightsquigarrow \text{inj} \ \text{event}(e_A(x)). \]

- Add a session identifier to \( e_B \): \( \text{event}(e_B(M), i) \).
  Distinguish in which session the event happens.
- Add an environment \( \rho \) to \( e_A \): \( \text{event}(e_A(M), \rho) \).
  \( \rho \) contains the variables that
  - have a single value for each execution of \( e_A \)
  - and are defined when we generate the clause.
  i.e. variables that are defined
    - above the first replication that follows \( e_A \)
    - and above the output or event that generates the clause.
- Prove \( \text{event}(e_B(x), i) \rightsquigarrow \text{event}(e_A(x), \rho) \), where \( i \) occurs in \( \rho \).
  Distinct events \( e_B(x) \) have distinct session identifiers \( i \), so they
correspond to distinct values of \( \rho \), and hence distinct events \( e_A(x) \).
Example

In the example,

\[(A) \quad \!^{i_A} \text{in}(c, x_{-pk_B}).\text{event}(e_A(x_{-pk_B})).\text{out}(c, pk_A).\text{in}(c, x_b).\text{out}(c, \text{sign}((pk_A, x_{-pk_B}, x_b), sk_A))\]

for the clause generated by the last output, \(\rho\) contains \(x_{-pk_B}\) and \(x_b\). We obtain

\[
\begin{align*}
\text{m-event}(e_A(pk(sk_B[])), \{x_b \mapsto b[pk(sk_A[]), i_B], x_{-pk_B} \mapsto pk(sk_B[])\}) \\
\Rightarrow \text{event}(e_B(pk(sk_B[])), i_B)
\end{align*}
\]

so we have the injective correspondence \(\text{event}(e_B(x)) \leadsto \text{inj} \text{event}(e_A(x))\), because the environment \(\rho\) contains \(i_B\).
Demo:

- **Woo-Lam public-key protocol**
  - examplesnd/demosimp/piwoolampk

- **Needham-Schroeder public-key protocol**
  - examplesnd/demosimp/pineedham-auth-orig
  - examplesnd/demosimp/pineedham-auth-corr-orig
## Experimental results

### Pentium III 1GHz

<table>
<thead>
<tr>
<th></th>
<th>Time (ms)</th>
<th>Cases with attacks</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NS</strong>=Needham-Schroeder</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>WL</strong>=Woo-Lam</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NS public key</td>
<td>25</td>
<td>None</td>
</tr>
<tr>
<td>NS public key corrected</td>
<td>16</td>
<td>None</td>
</tr>
<tr>
<td>WL public key</td>
<td>4</td>
<td>All[Lowe]</td>
</tr>
<tr>
<td>WL public key corrected</td>
<td>6</td>
<td>None</td>
</tr>
<tr>
<td>WL shared key (with tags)</td>
<td>6</td>
<td>All[Durante]</td>
</tr>
<tr>
<td>WL shared key corrected (tags)</td>
<td>5</td>
<td>None</td>
</tr>
<tr>
<td>Simpler Yahalom, unidirectional</td>
<td>29</td>
<td>Key</td>
</tr>
<tr>
<td>Simpler Yahalom, bidirectional</td>
<td>101</td>
<td>All[Syverson]</td>
</tr>
<tr>
<td>Otway-Rees</td>
<td>62</td>
<td>Key[BAN]</td>
</tr>
<tr>
<td>Simpler Otway-Rees</td>
<td>10</td>
<td>All[Paulson]</td>
</tr>
<tr>
<td>Main mode of Skeme</td>
<td>67</td>
<td>None</td>
</tr>
</tbody>
</table>

Bruno Blanchet (Inria) ProVerif Year 2016-17
Overview

1. A dialect of the applied pi calculus
2. Undecidability
3. The Horn clause representation
4. The resolution algorithm
5. Experimental results
6. Formal translation from the applied pi calculus
7. Extension to correspondences
8. Extension to observational equivalence
9. Conclusion
Joint work with Martín Abadi and Cédric Fournet.

Goal: extend tools designed for proving properties of behaviors (here ProVerif) to the proof of process equivalences.

- We focus on equivalences between processes that differ only by the terms they contain, e.g., $P(\text{true}) \approx P(\text{false})$.

Many interesting equivalences fall into this category.

- We introduce biprocesses to represent pairs of processes that differ only by the terms they contain.

$P(\text{true})$ and $P(\text{false})$ are variants of a biprocess $P(\text{diff[true, false]})$.

The variants give a different interpretation to $\text{diff[true, false]}$, true for the first variant, false for the second one.
We introduce a new operational semantics for biprocesses:

A biprocess reduces when both variants reduce in the same way and after reduction, they still differ only by terms (so can be written using \text{diff}).

We establish $P(\text{true}) \approx P(\text{false})$ by reasoning on behaviors of $P(\text{diff[true, false]})$:

If, for all reachable configurations, both variants reduce in the same way, then we have equivalence.
The process calculus

Extension of the pi-calculus with function symbols for cryptographic primitives. We allow several function symbols to be evaluated in one instruction.

\[ M, N ::= \]
\[ x, y, z \]
\[ a, b, c, k, s \]
\[ f(M_1, \ldots, M_n) \]

\[ D ::= \]
\[ M \]
\[ g(D_1, \ldots, D_n) \]

\[ P, Q, R ::= \]
\[ \text{in}(M, x).P \]
\[ \text{out}(M, N).P \]
\[ \text{let } x = D \text{ in } P \text{ else } Q \]
\[ 0 \] \[ P \parallel Q \]
\[ !P \]
\[ \text{new } a.P \]

terms
variable
name
constructor application
term evaluations
term
destructor evaluation
processes
input
output
term evaluation
Semantics

We use a semantics based on structural equivalence and reduction.

\[ D \Downarrow M \text{ when the term evaluation } D \text{ evaluates to } M. \]

Uses rewrite rules of destructors.

Main reduction rules:

\[ \text{out}(N, M).Q \parallel \text{in}(N', x).P \rightarrow Q \parallel P\{M/x\} \quad \text{(Red I/O)} \]

\[ \text{if } N = N' \]

\[ \text{let } x = D \text{ in } P \text{ else } Q \rightarrow P\{M/x\} \quad \text{(Red Fun 1)} \]

\[ \text{if } D \Downarrow M \]

\[ \text{let } x = D \text{ in } P \text{ else } Q \rightarrow Q \quad \text{(Red Fun 2)} \]

\[ \text{if there is no } M \text{ such that } D \Downarrow M \]
Observational equivalences and biprocesses

Two processes $P$ and $Q$ are observationally equivalent ($P \approx Q$) when the adversary cannot distinguish them.

A biprocess $P$ is a process with diff.

$fst(P) =$ the process obtained by replacing $\text{diff}[M, M']$ with $M$.

$snd(P) =$ the process obtained by replacing $\text{diff}[M, M']$ with $M'$.

$P$ satisfies observational equivalence when $fst(P) \approx snd(P)$.
A biprocess reduces when both variants of the process reduce in the same way.

\[
\text{out}(N, M).Q \parallel \text{in}(N', x).P \rightarrow Q \parallel P\{M/x\} \quad \text{(Red I/O)}
\]

if \(\text{fst}(N) = \text{fst}(N')\) and \(\text{snd}(N) = \text{snd}(N')\)

let \(x = D\) in \(P\) else \(Q \rightarrow P\{\text{diff}[M_1, M_2]/x\}\) \quad \text{(Red Fun 1)}

if \(\text{fst}(D) \Downarrow M_1\) and \(\text{snd}(D) \Downarrow M_2\)

let \(x = D\) in \(P\) else \(Q \rightarrow Q\) \quad \text{(Red Fun 2)}

if there is no \(M_1\) such that \(\text{fst}(D) \Downarrow M_1\) and there is no \(M_2\) such that \(\text{snd}(D) \Downarrow M_2\)

\[
\begin{align*}
P & \quad \rightarrow \quad Q \\
\text{snd}(P) & \quad \longrightarrow \quad \text{snd}(Q) \\
\text{fst}(P) & \quad \longrightarrow \quad \text{fst}(Q)
\end{align*}
\]
Let $P_0$ be a closed biprocess.

If for all configurations $P$ reachable from $P_0$ (in the presence of an adversary), both variants of $P$ reduce in the same way, then $P_0$ satisfies observational equivalence.
Proof of observational equivalence using biprocesses

Let $P_0$ be a closed biprocess.

*If for all configurations $P$ reachable from $P_0$ (in the presence of an adversary), both variants of $P$ reduce in the same way, then $P_0$ satisfies observational equivalence.*

An adversary is represented by a plain evaluation context (evaluation context without diff), so:

*If, for all plain evaluation contexts $C$ and reductions $C[P_0] \rightarrow^* P$, both variants of $P$ reduce in the same way, then $P_0$ satisfies observational equivalence.*
Let $P_0$ be a closed biprocess.

Suppose that, for all plain evaluation contexts $C$, all evaluation contexts $C'$, and all reductions $C[P_0] \rightarrow^* P$,

1. the **(Red I/O) rules** apply in the same way on both variants.
   
   If $P \equiv C'[\text{out}(N, M) \parallel \text{in}(N', x).R]$, then $\text{fst}(N) = \text{fst}(N')$ if and only if $\text{snd}(N) = \text{snd}(N')$,

2. the **(Red Fun) rules** apply in the same way on both variants.
   
   If $P \equiv C'[\text{let } x = D \text{ in } Q \text{ else } R]$, then there exists $M_1$ such that $\text{fst}(D) \downarrow M_1$ if and only if there exists $M_2$ such that $\text{snd}(D) \downarrow M_2$.

Then $P_0$ satisfies observational equivalence.
Example: Non-deterministic encryption

Non-deterministic public-key encryption is modeled by an equation:

\[ \text{adec}(\text{aenc}(x, \text{pk}(s), a), s) = x \]

Without knowledge of the decryption key, ciphertexts appear to be unrelated to the plaintexts.

Ciphertexts are indistinguishable from fresh names:

\[ \text{new } s \cdot (\text{out}(c, \text{pk}(s)) \parallel !\text{in}(c', x).\text{new } a.\text{out}(c, \text{diff}[\text{aenc}(x, \text{pk}(s), a), a])) \]

satisfies equivalence.

This equivalence can be proved using the previous result, and verified automatically by ProVerif.
Translation into clauses

As in our previous work, we translate the protocol and the adversary into a set of Horn clauses.

The predicates differ in order to translate behaviors of biprocesses instead of processes:

\[ F ::= \]

- \( \text{attacker}'(p, p') \)  the attacker has \( p \) (resp. \( p' \))
- \( \text{mess}'(p_1, p_2, p'_1, p'_2) \)  message \( p_2 \) is sent on \( p_1 \) (resp. \( p'_2 \) on \( p'_1 \))
- \( \text{input}'(p, p') \)  input on \( p \) (resp. \( p' \))
- \( \text{nounif}(p, p') \)  \( p \) and \( p' \) do not unify
- \( \text{bad} \)  the property may be false

Magenta arguments for the first version of the biprocess, blue ones for the second version.
Example: some generated clauses

The biprocess of the non-deterministic encryption example:

\[
\text{new } s. (\text{out}(c, \text{pk}(s)) \parallel \text{!in}(c', x). \text{new } a. \text{out}(c, \text{diff}[\text{aenc}(x, \text{pk}(s), a), a]))
\]

yields the clauses:

\[
\text{mess}'(c, \text{pk}(s), c, \text{pk}(s))
\]
\[
\text{mess}'(c', x, c', x') \Rightarrow \text{mess}'(c, \text{aenc}(x, \text{pk}(s), a[i, x]), c, a[i, x'])
\]

The first clause corresponds to the output of the public key \( \text{pk}(s) \).
The second clause corresponds to the other output.
Resolution algorithm

**Theorem**

*If* bad *is not a logical consequence of the clauses, then* \( P_0 \) *satisfies observational equivalence.*

We determine whether bad is a logical consequence of the clauses using a resolution-based algorithm.

This algorithm uses domain-specific simplification steps (for predicate nounif in particular).
**Applications**

- **Weak secrets**: We can express that a password is protected against off-line guessing attacks by an equivalence, and prove it using our technique (done for 4 versions of EKE).

- **Authenticity**: We can formalize authenticity as an equivalence and prove it (for the Wide-Mouth Frog protocol).

- **JFK**: We can show that the encrypted messages of JFK are equivalent to fresh names, with our technique plus the property that observational equivalence is contextual.

Total runtime: 45 s on a Pentium M 1.8 GHz.
Overview

1. A dialect of the applied pi calculus
2. Undecidability
3. The Horn clause representation
4. The resolution algorithm
5. Experimental results
6. Formal translation from the applied pi calculus
7. Extension to correspondences
8. Extension to observational equivalence
9. Conclusion
Conclusion: Some other results

- Automatic proof of strong secrecy [Blanchet, Oakland’04]
- Reconstruction of attacks from derivations [Allamigeon, Blanchet, CSFW’05]
- Case studies:
  - Certified email protocol [Abadi, Blanchet, SAS’03],
  - JFK [Abadi, Blanchet, Fournet, ESOP’04],
  - Plutus [Blanchet, Chaudhuri, S&P’08]

Software and papers at http://proverif.inria.fr
Usage by others

Tools that use ProVerif:
- Web service verifier TulaFale (Microsoft Research)
- Verification of F# implementations, including TLS (Microsoft Research and MSR-INRIA)
- Spi2Java includes verification by ProVerif via Spi2ProVerif (Pozza et al)
- E-voting protocols (Kremer and Ryan, Backes et al)
- Zero-knowledge protocols, DAA (Backes et al)
- Dolev-Yao proof implying computational security (Canetti and Herzog)
- ...

Bruno Blanchet (Inria) ProVerif Year 2016-17 156 / 158
Conclusion: Advantages of this technique

- A particularly efficient verifier
- Can handle complex protocols (JFK, ...)
- Unbounded number of runs of the protocol
  Unbounded message size
  \[\Rightarrow\] Can be used for certification of protocols
- Can prove various properties: secrecy, correspondences, observational equivalence
- Can handle a wide range of cryptographic primitives, specified by rewrite rules or by equations.
Conclusion: Limitations

- The proofs are done in the Dolev-Yao model. We would like automatic proof of protocols in a computational setting (Computational soundness; EasyCrypt, CryptoVerif) (see David Pointcheval’s course for manual computational proofs)

- The proofs are done on a model of the protocol. We would like automatic proof of implementations of protocols
  - Translation from implementation to model: FS2PV, FS2CV, ...
  - Generation of implementations from models: Spi2Java, CV2ML, ...
  - Direct verification of implementation: F*, miTLS, ...

(see Karthik Bhargavan and Catalin Hritcu’s course)