# Proving observational equivalence with ProVerif

Bruno Blanchet

INRIA Paris-Rocquencourt
Bruno.Blanchet@inria.fr

based on joint work with Martín Abadi and Cédric Fournet
and with Vincent Cheval

June 2015

# Introduction

Analysis of cryptographic protocols:

- Powerful automatic tools for proving properties on behaviors (traces) of protocols (secrecy of keys, correspondences).
  - For instance, ProVerif was initially designed to prove trace properties.
- Many important properties can be formalized as process equivalences, not as properties on behaviors:
  - secrecy of a boolean $x$ in $P(x)$: $P(\mathrm{true}) \approx P(\mathrm{false})$
  - the process $P$ implements an ideal specification $Q$: $P \approx Q$

  Equivalences are usually proved by difficult, long manual proofs.
  Already much research on this topic, using in particular sophisticated bisimulation techniques (e.g., Boreale et al).
  Some recent tools for a bounded number of sessions (APTE, Akiss)

# Selected work relying on equivalences

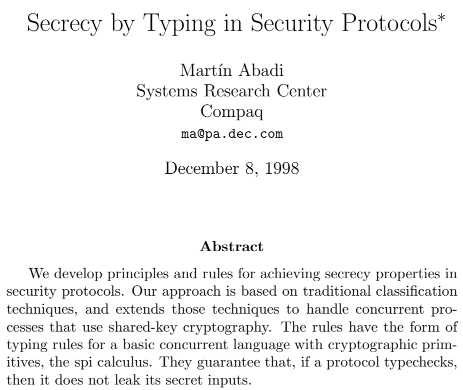
January 25, 1998

**SRC** Research Report **149**

**A Calculus for Cryptographic Protocols**
**The Spi Calculus**

Martín Abadi and Andrew D. Gordon

# Selected work relying on equivalences


Secrecy by Typing in Security Protocols*

Martín Abadi
Systems Research Center
Compaq
ma@pa.dec.com

December 8, 1998

**Abstract**

We develop principles and rules for achieving secrecy properties in security protocols. Our approach is based on traditional classification techniques, and extends those techniques to handle concurrent processes that use shared-key cryptography. The rules have the form of typing rules for a basic concurrent language with cryptographic primitives, the spi calculus. They guarantee that, if a protocol typechecks, then it does not leak its secret inputs.

# Selected work relying on equivalences

## Mobile Values, New Names, and Secure Communication

Martín Abadi
Bell Labs Research
Lucent Technologies

Cédric Fournet
Microsoft Research

**Abstract**

We study the interaction of the "new" construct with a rich but common form of (first-order) communication. This interaction is crucial in security protocols, which are the main motivating examples for our work; it also appears in other programming-language contexts. Specifically, we introduce a simple, general extension of the pi calculus with value passing, primitive functions, and equations among terms. We develop semantics and proof techniques for this extended language and apply them in reasoning about some security protocols.

### 1 A case for impurity

Purity often comes before convenience and even before faithfulness in the lambda calculus, the pi calculus, and

These difficulties are often circumvented through on-the-fly extensions. The extensions range from quick punts ("for the next example, let's pretend that we have a datatype of integers") to the laborious development of new calculi, such as the spi calculus and its variants. Generally, the extensions bring us closer to a realistic programming language or modeling language—that is not always a bad thing.

Although many of the resulting calculi are ad hoc and poorly understood, others are robust and uniform enough to have a rich theory and a variety of applications. In particular, impure extensions of the lambda calculus with function symbols and with equations among terms ("delta rules") have been developed systematically, with considerable success. Similarly, impure versions of CCS and CSP with value-passing are not always deep but often neat and convenient [31].

In this paper, we introduce, study, and use an analo-

# Equivalences as properties of behaviors (1)

Goal: extend ProVerif to the proof of process equivalences.

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

  Many interesting equivalences fall into this category.
- We introduce biprocesses to represent pairs of processes that differ only by the terms they contain.

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

  The variants give a different interpretation to $\mathsf{diff}[\mathsf{true}, \mathsf{false}]$, $\mathsf{true}$ for the first variant, $\mathsf{false}$ for the second one.

# Equivalences as properties of behaviors (2)

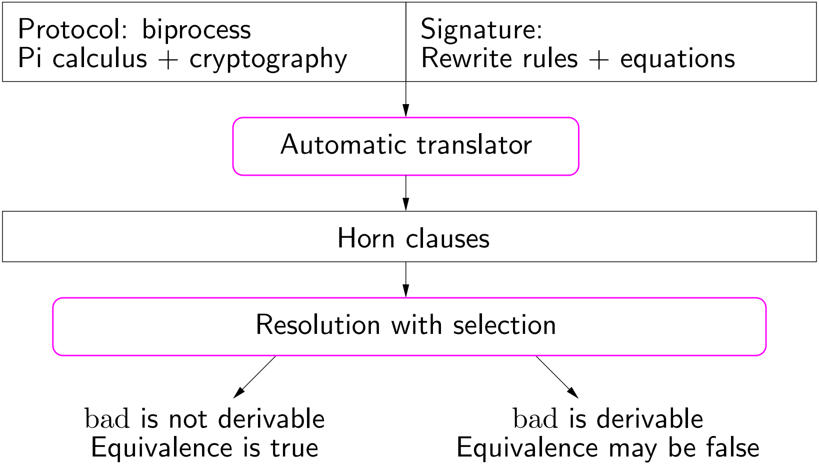
- We introduce a new operational semantics for biprocesses:

  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 diff).

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

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

# Overview of the verification method



# The process calculus

Dialect of the applied pi calculus [Abadi, Fournet, POPL'01].

| | |
|---|---|
| $M, N ::=$ | terms |
| $x, y, z$ | variable |
| $a, b, c, k, s$ | name |
| $f(M_1, \ldots, M_n)$ | constructor application |
| $D ::=$ | term evaluations |
| $M$ | term |
| fail | special failure value |
| $h(D_1, \ldots, D_n)$ | function evaluation |
| $P, Q, R ::=$ | processes |
| $\underline{M}(x).P$ | input |
| $\overline{M}\langle N \rangle.P$ | output |
| **let** $x = D$ **in** $P$ **else** $Q$ | term evaluation |
| $\mathbf{0} \quad P \mid Q \quad !P \quad (\nu a)P$ | |

**if** $M$ **then** $P$ **else** $Q$ encoded as a term evaluation.

# Representation of cryptographic primitives

Two possible representations:

- When success/failure is visible: destructors with rewrite rules

$$\begin{aligned}
&\text{constructor sencrypt}\\
&\text{destructor } \mathrm{sdecrypt}(\mathrm{sencrypt}(x, y), y) \rightarrow x \ \textbf{otherwise}\\
&\qquad\qquad\quad \mathrm{sdecrypt}(u, v) \rightarrow \mathrm{fail}
\end{aligned}$$

  The **else** branch of the term evaluation is executed when $D$ fails, that is, $D$ evaluates to fail.

- When success/failure is not visible: equations

$$\mathrm{sdecrypt}(\mathrm{sencrypt}(x, y), y) = x$$
$$\mathrm{sencrypt}(\mathrm{sdecrypt}(x, y), y) = x$$

# Representation of cryptographic primitives

Two possible representations:

- When success/failure is visible: destructors with rewrite rules

$$\begin{aligned}&\text{constructor senc}\text{rypt}\\&\text{destructor sdecrypt}(\text{sencrypt}(x, y), y) \to x \ \textbf{otherwise}\\&\qquad\qquad\quad \text{sdecrypt}(u, v) \to \text{fail}\end{aligned}$$

  The **else** branch of the term evaluation is executed when $D$ fails, that is, $D$ evaluates to fail.

- When success/failure is not visible: equations

$$\text{sdecrypt}(\text{sencrypt}(x, y), y) = x$$
$$\text{sencrypt}(\text{sdecrypt}(x, y), y) = x$$

# Semantics

- $D \Downarrow M$ when the term evaluation $D$ evaluates to $M$.
  $D \Downarrow \mathrm{fail}$ when the term evaluation $D$ fails.
  Uses rewrite rules of destructors and equations.
- $\equiv$ transforms processes so that reduction rules can be applied.
- Main reduction rules:

$$\overline{N}\langle M\rangle.Q \mid N'(x).P \;\to\; Q \mid P\{M/x\} \qquad \text{(Red I/O)}$$
$$\text{if } \Sigma \vdash N = N'$$

$$\textbf{let } x = D \textbf{ in } P \textbf{ else } Q \to P\{M/x\} \qquad \text{(Red Fun 1)}$$
$$\text{if } D \Downarrow M$$

$$\textbf{let } x = D \textbf{ in } P \textbf{ else } Q \to Q \qquad \text{(Red Fun 2)}$$
$$\text{if } D \Downarrow \mathrm{fail}$$

# 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.

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

$P$ satisfies observational equivalence when $\mathsf{fst}(P) \approx \mathsf{snd}(P)$.

# Semantics of biprocesses

A biprocess reduces when both variants of the process reduce in the same way.

$$\overline{N}\langle M\rangle.Q \mid N'(x).P \;\rightarrow\; Q \mid P\{M/x\} \qquad \text{(Red I/O)}$$
$$\text{if } \Sigma \vdash \mathrm{fst}(N) = \mathrm{fst}(N') \text{ and } \Sigma \vdash \mathrm{snd}(N) = \mathrm{snd}(N')$$

$$\textbf{let } x = D \textbf{ in } P \textbf{ else } Q \rightarrow P\{\mathrm{diff}[M_1, M_2]/x\} \qquad \text{(Red Fun 1)}$$
$$\text{if } \mathrm{fst}(D) \Downarrow M_1 \text{ and } \mathrm{snd}(D) \Downarrow M_2$$

$$\textbf{let } x = D \textbf{ in } P \textbf{ else } Q \rightarrow Q \qquad \text{(Red Fun 2)}$$
$$\text{if } \mathrm{fst}(D) \Downarrow \mathrm{fail} \text{ and } \mathrm{snd}(D) \Downarrow \mathrm{fail}$$

$$\begin{tikzcd}
P \arrow[rr] \arrow[dr, dash] \arrow[dd, dash] & & Q \arrow[dr, dash] \arrow[dd, dash] & \\
& \mathrm{snd}(P) \arrow[rr, dashed] & & \mathrm{snd}(Q) \\
\mathrm{fst}(P) \arrow[rr, dashed] & & \mathrm{fst}(Q) &
\end{tikzcd}$$

# 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.*

# 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.*

# Formalizing “reduce in the same way”

The biprocess $P$ is uniform when
$\mathsf{fst}(P) \rightarrow Q_1$ implies $P \rightarrow Q$ for some biprocess $Q$ with $\mathsf{fst}(Q) \equiv Q_1$,
and symmetrically for $\mathsf{snd}(P) \rightarrow Q_2$.

$$
\begin{tikzcd}
P \arrow[rr, dashed] \arrow[dd, dash] & & Q \arrow[d, dash, dotted] \\
& & \mathsf{fst}(Q) \arrow[d, phantom, "\equiv"] \\
\mathsf{fst}(P) \arrow[rr] & & Q_1
\end{tikzcd}
\qquad
\begin{tikzcd}
P \arrow[rr, dashed] \arrow[dd, dash] & & Q \arrow[d, dash, dotted] \\
& & \mathsf{snd}(Q) \arrow[d, phantom, "\equiv"] \\
\mathsf{snd}(P) \arrow[rr] & & Q_2
\end{tikzcd}
$$

If, for all plain evaluation contexts $C$ and reductions $C[P_0] \rightarrow^* P$, the biprocess $P$ is uniform,
then $P_0$ satisfies observational equivalence.

# Result [Blanchet, Abadi, Fournet, JLAP, 2008]

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 $P \equiv C'[\overline{N}\langle M\rangle.Q \mid N'(x).R]$, then
   $\Sigma \vdash \mathsf{fst}(N) = \mathsf{fst}(N')$ if and only if $\Sigma \vdash \mathsf{snd}(N) = \mathsf{snd}(N')$,

2. the (Red Fun) rules apply in the same way on both variants.

   if $P \equiv C'[\mathbf{let}\ x = D\ \mathbf{in}\ Q\ \mathbf{else}\ R]$, then
   $\mathsf{fst}(D) \Downarrow \mathsf{fail}$ if and only if $\mathsf{snd}(D) \Downarrow \mathsf{fail}$.

Then $P_0$ satisfies observational equivalence.

# Application

- The previous result reduces the proof of observational equivalence to the proof of trace properties for biprocesses.
- We extend the Horn clause approach of ProVerif from processes to biprocesses, to prove these properties.
  - Basically, each argument of the predicates is doubled, with one argument for each side.

# Example: “probabilistic” encryption

Probabilistic public-key encryption is modeled by an equation:

$$\mathrm{dec}(\mathrm{enc}(x, \mathrm{pk}(s), a), s) = x$$

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

Ciphertexts are indistinguishable from fresh names:

$$(\nu s)(\overline{c}\langle \mathrm{pk}(s)\rangle \mid !c'(x).(\nu a)\overline{c}\langle \mathrm{diff}[\mathrm{enc}(x, \mathrm{pk}(s), a), a]\rangle)$$

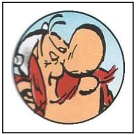
satisfies equivalence.

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

# Example: private authentication

Simplified version of the private authentication protocol [Abadi, Fournet, TCS, 2004].

$\xrightarrow{\{N_A, pk_A\}_{pk_B}}$ $\xrightarrow{\{x, y\}_{pk_B}}$ $y = pk_A?$

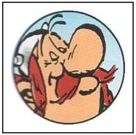


# Example: private authentication

Simplified version of the private authentication protocol [Abadi, Fournet, TCS, 2004].

$$\xrightarrow{\{N_A, pk_A\}_{pk_B}} \qquad \xrightarrow{\{x, y\}_{pk_B}} \quad y = pk_A?$$

$$\xleftarrow{\{x, N_B, pk_B\}_y} \qquad \xleftarrow{\text{yes}}$$



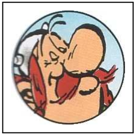
# Example: private authentication

Simplified version of the private authentication protocol [Abadi, Fournet, TCS, 2004].

$\xrightarrow{\{N_A, pk_A\}_{pk_B}}$ $\xrightarrow{\{x, y\}_{pk_B}}$ $y = pk_A?$

$\xleftarrow{\{x, N_B, pk_B\}_y}$ $\xleftarrow{\text{yes}}$

$\xleftarrow{\{N_B\}_{pk_B}}$ $\xleftarrow{\text{no}}$

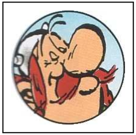


# Proving anonymity

$\xrightarrow{\{x,y\}_{pk_B}}$ $y = pk_A?$

$\xleftarrow{\{x,N_B,pk_B\}_y}$ $\xleftarrow{\text{yes}}$

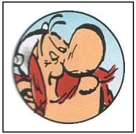
$\xleftarrow{\{N_B\}_{pk_B}}$ $\xleftarrow{\text{no}}$



$\xrightarrow{\{x,y\}_{pk_B}}$ $y = pk_{A'}?$

$\xleftarrow{\{x,N_B,pk_B\}_y}$ $\xleftarrow{\text{yes}}$

$\xleftarrow{\{N_B\}_{pk_B}}$ $\xleftarrow{\text{no}}$



Caesar should not be able to know who Obelix wants to talk to.

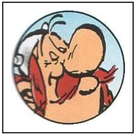
# False attack

$\xrightarrow{\{N_A, pk_A\}_{pk_B}}$

$\xrightarrow{\{x, y\}_{pk_B}}$ $y = pk_A?$

$\xleftarrow{\{x, N_B, pk_B\}_y}$ $\xleftarrow{\text{yes}}$

$\xleftarrow{\{N_B\}_{pk_B}}$ $\xleftarrow{\text{no}}$

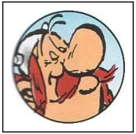


$\xrightarrow{\{N_A, pk_A\}_{pk_B}}$

$\xrightarrow{\{x, y\}_{pk_B}}$ $y = pk_{A'}?$

$\xleftarrow{\{x, N_B, pk_B\}_y}$ $\xleftarrow{\text{yes}}$

$\xleftarrow{\{N_B\}_{pk_B}}$ $\xleftarrow{\text{no}}$

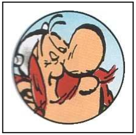


# False attack

$\xrightarrow{\{N_A, pk_A\}_{pk_B}}$

$\xrightarrow{\{x, y\}_{pk_B}}$ $y = pk_A?$

$\xleftarrow{\{x, N_B, pk_B\}_y}$ $\xleftarrow{\text{yes}}$

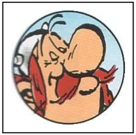


$\xrightarrow{\{N_A, pk_A\}_{pk_B}}$

$\xrightarrow{\{x, y\}_{pk_B}}$ $y = pk_{A'}?$

$\xleftarrow{\{x, N_B, pk_B\}_y}$ $\xleftarrow{\text{yes}}$

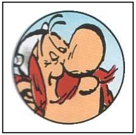
$\xleftarrow{\{N_B\}_{pk_B}}$ $\xleftarrow{\text{no}}$



# False attack

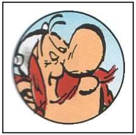
$\xrightarrow{\{N_A, pk_A\}_{pk_B}}$ $\xrightarrow{\{x, y\}_{pk_B}}$ $y = pk_A?$

$\xleftarrow{\{x, N_B, pk_B\}_y}$ $\xleftarrow{\text{yes}}$



$\xrightarrow{\{N_A, pk_A\}_{pk_B}}$ $\xrightarrow{\{x, y\}_{pk_B}}$ $y = pk_{A'}?$

$\xleftarrow{\{N_B\}_{pk_B}}$ $\xleftarrow{\text{no}}$



The test takes a different branch: the proof of equivalence fails.

# Demo

- File `private_auth_falseattack.pv`

# Solution: encode tests inside terms [Cheval, Blanchet, POST'13]

The if-then-else destructor:

$$\begin{aligned}
&\text{ifthenelse}(x, x, u, v) \rightarrow u \ \textbf{otherwise} \\
&\text{ifthenelse}(x, y, u, v) \rightarrow v
\end{aligned}$$

# Solution: encode tests inside terms [Cheval, Blanchet, POST'13]

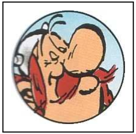
The if-then-else destructor:

$$\mathrm{ifthenelse}(x, x, u, v) \to u \ \textbf{otherwise}$$
$$\mathrm{ifthenelse}(x, y, u, v) \to v$$

$$\xrightarrow{\{x,y\}_{pk_B}} \quad y = pk_A?$$

$$\xleftarrow{\{x,N_B,pk_B\}_y} \quad \xleftarrow{\text{yes}}$$

$$\xleftarrow{\{N_B\}_{pk_B}} \quad \xleftarrow{\text{no}}$$



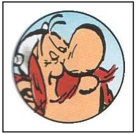
# Solution: encode tests inside terms [Cheval, Blanchet, POST'13]

The if-then-else destructor:

$$\mathrm{ifthenelse}(x, x, u, v) \rightarrow u \textbf{ otherwise}$$
$$\mathrm{ifthenelse}(x, y, u, v) \rightarrow v$$

$\xrightarrow{\{x,y\}_{pk_B}}$

$\xleftarrow{M}$



$$M = \mathrm{ifthenelse}(y, \mathrm{pk}_A, \{x, N_B, pk_B\}_y, \{N_B\}_{pk_B})$$

# Demo

- File `private_auth_direct.pv`

## Merging two processes

In order to prove that $P$ and $Q$ are observationally equivalent using this approach, we need to merge $P$ and $Q$ into a biprocess $R$:

$$\mathsf{fst}(R) \approx P \qquad \mathsf{snd}(R) \approx Q$$

- Trivial when $P$ and $Q$ have exactly the same structure.
- There is a general merging:

$$R = \textbf{if } \mathsf{diff}[\mathsf{true}, \mathsf{false}] \textbf{ then } P \textbf{ else } Q$$

  does not allow ProVerif to prove observational equivalence.
- Do something more clever!

# A related problem: simplifying biprocesses

- In the private authentication example, we moved a test from the process level to the term level.
- Doing the same on the process

$$R = \textbf{if } \mathrm{diff}[\mathrm{true}, \mathrm{false}] \textbf{ then } P \textbf{ else } Q$$

could allow ProVerif to prove observational equivalence.

# A related problem: simplifying biprocesses

- In the private authentication example, we moved a test from the process level to the term level.
- Doing the same on the process

$$R = \textbf{if } \mathsf{diff}[\mathsf{true}, \mathsf{false}] \textbf{ then } P \textbf{ else } Q$$

could allow ProVerif to prove observational equivalence.

- ... but moving this test implies merging the structures of $P$ and $Q$!

# Merging algorithm

- Merge branches of biprocesses:
  - Automatically transforms the private authencation example
  - Allows merging $P$ and $Q$ by applying it to
    **if** diff[true, false] **then** $P$ **else** $Q$
- Two steps:
  1. "Normalize" the process
     (Try to reorganize processes that send the same messages so that they have the same syntactic form.)
  2. Merge branches on the normalized process

# Normalization algorithm

1. Make sure term evaluations always succeed
   - New destructors:

$$\mathrm{catchfail}(x) \rightarrow x \ \textbf{otherwise}$$
$$\mathrm{catchfail}(\mathrm{fail}) \rightarrow \mathrm{c_{fail}}$$

$$\mathrm{not\_c_{fail}}(\mathrm{c_{fail}}) \rightarrow \mathrm{false}\ \textbf{otherwise}$$
$$\mathrm{not\_c_{fail}}(x) \rightarrow \mathrm{true}$$

   - **let** $x = D$ **in** $P$ **else** $Q$
     $\rightarrow$ **let** $x = \mathrm{catchfail}(D)$ **in if** $\mathrm{not\_c_{fail}}(x)$ **then** $P$ **else** $Q$
     (We allow destructors outside **let**; they can be encoded using an additional **let**.)
2. Remove unused restrictions and lets; $!0 \rightarrow 0$

# Normalization algorithm

3. Move new, let, if before outputs

$$\overline{M}\langle N\rangle.(\nu a)P \rightarrow (\nu a)\overline{M}\langle N\rangle.P$$
$$\overline{M}\langle N\rangle.\textbf{let } x = D \textbf{ in } P \rightarrow \textbf{let } x = D \textbf{ in } \overline{M}\langle N\rangle.P$$
$$\overline{M}\langle N\rangle.\textbf{if } M \textbf{ then } P \textbf{ else } Q \rightarrow \textbf{if } M \textbf{ then } \overline{M}\langle N\rangle.P \textbf{ else } \overline{M}\langle N\rangle.Q$$

   (Avoid capture of names and variables.)

4. Move new, let, if before parallel composition

$$((\nu a).P) \mid Q \rightarrow (\nu a).(P \mid Q)$$
$$(\textbf{let } x = D \textbf{ in } P) \mid Q \rightarrow \textbf{let } x = D \textbf{ in } (P \mid Q)$$
$$(\textbf{if } M \textbf{ then } P \textbf{ else } P') \mid Q \rightarrow \textbf{if } M \textbf{ then } P \mid Q \textbf{ else } P' \mid Q$$

5. Group replicated processes inside a parallel composition

$$!P \mid P' \mid !P'' \rightarrow P' \mid !(P \mid P'')$$

# Normalization algorithm

6. Move new, let before if

$$\textbf{if } M \textbf{ then } (\nu a)P \textbf{ else } Q \rightarrow (\nu a)\textbf{if } M \textbf{ then } P \textbf{ else } Q$$

$$\textbf{if } M \textbf{ then let } x = D \textbf{ in } P \textbf{ else } Q \rightarrow \textbf{let } x = D \textbf{ in if } M \textbf{ then } P \textbf{ else } Q$$

7. Move new before let

$$\textbf{let } x = D \textbf{ in } (\nu a)P \rightarrow (\nu a)\textbf{let } x = D \textbf{ in}$$

8. Move if (with its new/let) before replication

$$!s_\nu s_{\text{let}}\textbf{if } M \textbf{ then } P \textbf{ else } Q \rightarrow s_\nu s_{\text{let}}\textbf{if } M \textbf{ then } !s_\nu s_{\text{let}} P \textbf{ else } !s_\nu s_{\text{let}} Q$$

The equational theory is closed under one-to-one renaming, so the test $M$ will always yield the same result independently of the considered fresh names.

9. Remove double replication

$$!s_{\text{let}}!s_\nu s'_{\text{let}} Q \rightarrow !s_\nu s_{\text{let}} s'_{\text{let}} Q$$

# Grammar of normalized processes

$P ::= s_\nu s_{\text{let}} T$

$s_\nu$ consists of a sequence of $(\nu a)$

$s_{\text{let}}$ consists of a sequence of **let** $x = D$ **in** such that $D$ always succeeds.

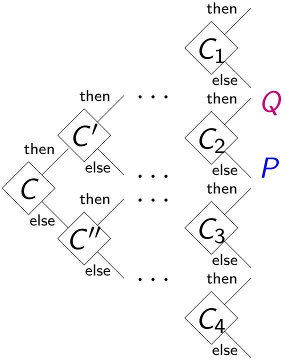
| | |
|---|---|
| $T ::=$ | decision tree |
| $\quad Q$ | |
| $\quad$ **if** $M$ **then** $T_1$ **else** $T_2$ | test |
| $Q ::= R_1 \mid \ldots \mid R_n \mid S$ | parallel composition $(n \geq 0)$ |
| $R ::=$ | communication |
| $\quad M(x).P$ | input |
| $\quad \overline{M}\langle N\rangle.Q$ | output |
| $S ::=$ | optional replicated process |
| $\quad !s_\nu s_{\text{let}} Q$ | replicated process |
| $\quad \mathbf{0}$ | no replicated process |

# Some explanation

- When two processes $Q$ have the same structure, we will be able to merge them.
  The structure
  - ignores the precise channels and messages of inputs and outputs and
  - considers parallel composition up to associativity and commutativity.
- It seems difficult to go further with general syntactic rules.

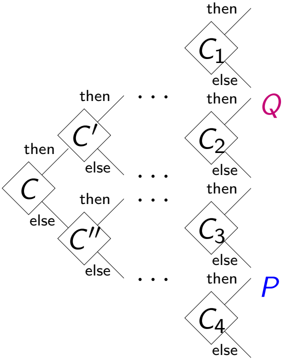
# Merging

- Merge leaves of decision trees



# Merging

- Merge leaves of decision trees




- If necessary, reorganize the decision tree so that the merged leaves are the two branches of an **if**.

# Merging inputs and outputs ($R$)

- New destructor

$$
\begin{aligned}
&\mathrm{ite}(\mathrm{true}, u, v) \rightarrow u \textbf{ otherwise} \\
&\mathrm{ite}(x, u, v) \rightarrow v
\end{aligned}
$$

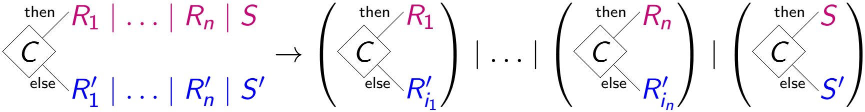
- Outputs

$$
C \begin{cases} \text{then} & \overline{M}\langle N\rangle.Q \\ \text{else} & \overline{M'}\langle N'\rangle.Q' \end{cases} \rightarrow \overline{\mathrm{ite}(C, M, M')}\langle \mathrm{ite}(C, N, N')\rangle.\, C \begin{cases} \text{then} & Q \\ \text{else} & Q' \end{cases}
$$

- Inputs

$$
C \begin{cases} \text{then} & M(x).Q \\ \text{else} & M'(x').Q' \end{cases} \rightarrow \mathrm{ite}(C, M, M')(x'').\, C \begin{cases} \text{then} & Q\{^{x''}/_{x}\} \\ \text{else} & Q'\{^{x''}/_{x'}\} \end{cases}
$$

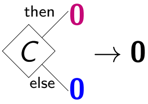
# Merging parallel compositions ($Q$)



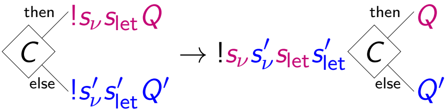

$i_1, \dots i_n$ is a permutation of $1, \dots, n$.

# Merging replications ($S$)

- Nil

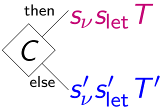



- Replication (naive version)




- $!s_{\nu 1} s_{\mathrm{let}1} !s_{\nu 2} s_{\mathrm{let}2} Q$ can actually be merged with $!s'_\nu s'_{\mathrm{let}} Q'$ by merging $Q$ and $Q'$, because $!s'_\nu s'_{\mathrm{let}} Q' \approx !!s'_\nu s'_{\mathrm{let}} Q'$.
  - Separate the processes in $s_\nu s_{\mathrm{let}} Q$ and $s'_\nu s'_{\mathrm{let}} Q'$ into independent groups.
  - Possibly add a replication in front of some of these groups.

# Merging general processes ($P$)




$\rightarrow s_\nu s'_\nu s_{\mathrm{let}} s'_{\mathrm{let}}$ **if** $C$ **then** $T$ **else** $T'$

# Properties of this algorithm

- Soundness: these transformations preserve observational equivalence
- Incomplete.
- Open question: partial completeness?
  - All channels public
  - Other conditions?

# Demo

- File `private_auth_biprocess.pv`
- File `private_auth_processes.pv`

# Conclusion

- ProVerif is the only tool that can prove process equivalences for an unbounded number of sessions.
- Restricted but useful class of equivalences:
  - First restricted to processes that differ only by the terms they contain.
  - Extended by automatic merging of processes.
- Tool available at `http://proverif.inria.fr`.

# Back to the applied pi calculus

## Mobile Values, New Names, and Secure Communication

Martín Abadi
Bell Labs Research
Lucent Technologies

Cédric Fournet
Microsoft Research

**Abstract**

We study the interaction of the "new" construct with a rich but common form of (first-order) communication. This interaction is crucial in security protocols, which are the main motivating examples for our work; it also appears in other programming-language contexts. Specifically, we introduce a simple, general extension of the pi calculus with value passing, primitive functions, and equations among terms. We develop semantics and proof techniques for this extended language and apply them in reasoning about some security protocols.

### 1 A case for impurity

Purity often comes before convenience and even before faithfulness in the lambda calculus, the pi calculus, and

These difficulties are often circumvented through on-the-fly extensions. The extensions range from quick punts ("for the next example, let's pretend that we have a datatype of integers") to the laborious development of new calculi, such as the spi calculus and its variants. Generally, the extensions bring us closer to a realistic programming language or modeling language—that is not always a bad thing.

Although many of the resulting calculi are ad hoc and poorly understood, others are robust and uniform enough to have a rich theory and a variety of applications. In particular, impure extensions of the lambda calculus with function symbols and with equations among terms ("delta rules") have been developed systematically, with considerable success. Similarly, impure versions of CCS and CSP with value-passing are not always deep but often neat and convenient [31].

In this paper, we introduce, study, and use an analo-

# Advertisement

With Martín Abadi and Cédric Fournet, we recently revisited their applied pi calculus paper [POPL'01]:

- Minor changes to the language to make it closer to ProVerif
- Detailed proofs of all results
- Revised examples
    - New example on indifferentiability