Automated Verification of Selected Equivalences for Security Protocols

Markus Rabe

Computer Science Department
Saarland University

January 23, 2009
Outline

1. Process Calculi
   - History
   - Basics

2. Applied $\pi$-Calculus
   - Informal Semantics
   - Constructors and Destructors

3. Observational Equivalence
   - Definition and Examples

4. Preparations for the algorithm

5. Automated Checking of Equivalences
   - Introduction
   - Demonstration of the algorithm
Milestones on the road to the Applied $\pi$-Calculus:

- 1980: CCS
- 1989: $\pi$-Calculus
- 1997: Spi-Calculus
- 2001: Applied $\pi$-Calculus
Process Calculi - Basics

Basic constructs:

- Processes $P, Q$
Process Calculi - Basics

Basic constructs:

- Processes: \( P, Q \)
- Concurrency: \( P | Q \)
Process Calculi - Basics

Basic constructs:

- Processes
  \[ P, \; Q \]

- Concurrency
  \[ P \mid Q \]

- Communication
  \[ \bar{c}\langle\text{msg}\rangle.P \mid c(x).Q \]
Basic constructs:

- Processes
- Concurrency
- Communication

Available since CCS.
Informal Semantics

New names: \((\nu k)P\)
Informal Semantics

New names: \((\nu k)P\)

Evaluation of terms: let \(x = D\) in \(P\) else \(Q\)
Informal Semantics

New names: \((\nu k)P\)

Evaluation of terms: let \(x = D\) in \(P\) else \(Q\)

How to evaluate \(D\)?
Informal Semantics

New names: \((\nu k)P\)

Evaluation of terms: \(\text{let } x = D \text{ in } P \text{ else } Q\)

Replication: \(!P := P | P | \ldots\)
Informal Semantics

New names: \((\nu k)P\)

Evaluation of terms: \(\text{let } x = D \text{ in } P \text{ else } Q\)

Replication: \(!P := P \mid P \mid \ldots\)

What is special about Applied \(\pi\)-Calculus?

Arbitrary constructors and destructors.

How to evaluate \(D\)?
Constructors and Destructors

Additional to the process in question we define an *equational theory* $\Sigma$.

\[
\Sigma \\
\downarrow \\
def\Sigma \\
\Downarrow \\
\Rightarrow \Sigma
\]
Constructors and Destructors

Additional to the process in question we define an *equational theory* $\Sigma$.

$$
\Sigma \\
\downarrow
\downarrow
\text{def}_\Sigma = \Sigma
$$

Example: We define tuples!

$$
def_{\Sigma} := \text{pair}(M, N) \rightarrow \text{pair}(M, N),\\
\text{fst}(\text{pair}(M, N)) \rightarrow M,\\
\text{snd}(\text{pair}(M, N)) \rightarrow N\\
=\Sigma := \emptyset
$$
Example:

\[ \overline{a}\langle \text{pair}(M, N) \rangle \mid a(x).\text{let } y = \text{fst}(x) \text{ in } P \text{ else } Q \]
Example:

\[ \bar{a}\langle b \rangle \mid a(x).\text{let } y = \text{fst}(x) \text{ in } P \text{ else } Q \]
Constructors and Destructors (2)

Example:

\[ \bar{a}(b) \mid a(x).\text{let } y = \text{fst}(x) \text{ in } P \text{ else } Q \]

Other functions:

- Symmetric encryption:
  \[ \text{dec}_y(\text{enc}_y(x)) \rightarrow x \]

- Public key encryption:
  \[ \text{dec}_{sk(y)}(\text{enc}_{pk(y)}(x)) \rightarrow x \]
What we want to check

Are processes $P$ and $Q$ equal?
What we want to check

Are processes $P$ and $Q$ equal?

**Definition**

*Observational equivalence* $\approx$ is the largest symmetric relation $R$ [...] such that $P R Q$ implies:

- if $P \Downarrow_M$ then $Q \Downarrow_M$,
- if $P \rightarrow P'$ then $Q \rightarrow Q'$ and $P' R Q'$ for some $Q'$,
- $C[P] R C[Q]$ for all evaluation contexts $C$. 
What we want to check

Are processes \( P \) and \( Q \) equal?

**Definition**

*Observational equivalence* \( \approx \) is the largest symmetric relation \( \mathcal{R} \) such that \( P \mathcal{R} Q \) implies:

- if \( P \Downarrow M \) then \( Q \Downarrow M \),
- if \( P \rightarrow P' \) then \( Q \rightarrow Q' \) and \( P' \mathcal{R} Q' \) for some \( Q' \),
- \( C[P] \mathcal{R} C[Q] \) for all evaluation contexts \( C \).

\( \rightarrow \) Resembles bisimulation in CCS.
Examples

Markus Rabe  Verification of Selected Equivalences
What is a Horn clause?
A disjunction of literals with at most one positive literal.

Simpler: An implication of the form:

$$p_1 \land p_2 \land \cdots \land p_{n-1} \rightarrow p_n$$
Additional Notation: $\text{diff}[\ldots]$

We do not aim to automatically check two arbitrary processes.
Additional Notation: \texttt{diff[...]} 

We do not aim to automatically check two arbitrary processes.

Instead we would like to check equality of two different versions of the same protocol. So we specify a \textit{biprocess}: 
We do not aim to automatically check two arbitrary processes.

Instead we would like to check equality of two different versions of the same protocol. So we specify a \textit{biprocess}:

\begin{equation}
P(\text{diff}[M, N]) \approx P(M) \equiv P(N)
\end{equation}
Additional Notation: \texttt{diff[...]} \\

We do not aim to automatically check two arbitrary processes.

Instead we would like to check equality of two different versions of the same protocol. So we specify a \textit{biprocess}:

\[ P(\text{diff}[M, N]) \]

\[ \quad \quad \quad \quad \quad P(M) \quad \approx \quad P(N) \]

E.g.:

\[ P(_\_ \_) \ := \ \ (\nu k)\tilde{a}\langle _\_ \rangle \]
\[ P(\text{diff}[h(k), h((m, k))]) \ = \ (\nu k)\tilde{a}\langle \text{diff}[h(k), h((m, k))] \rangle \]
Our example process

\[ P := \bar{c}\langle\text{enc}_k(m)\rangle \]
\[ Q := c(x)\text{.let } y = \text{dec}_k(x) \text{ in } \bar{c}\langle y\rangle \]

System to analyse:

\[(\nu k)(P | Q)\]
Our example process

\begin{align*}
P &:= \overline{c}\langle\text{enc}_k(m)\rangle \\
Q &:= c(x).\text{let } y = \text{dec}_k(x) \text{ in } \overline{c}\langle y \rangle
\end{align*}

System to analyse:

\((\nu k)(P | Q)\)

Intuition:

- Process \(P\) sends encrypted message \(m\) on channel \(c\).
- \(Q\) decrypts a received message and emits the result (if any).
Our example process

\[ P := \text{\textbar}c\langle\text{enc}_k(m)\rangle \]

\[ Q := c(x).\text{let } y = \text{dec}_k(x) \text{ in } \text{\textbar}c\langle y\rangle \]

System to analyse:

\[ (\nu k)(P \mid Q) \]

Intuition:

- Process \( P \) sends encrypted message \( m \) on channel \( c \).
- \( Q \) decrypts a received message and emits the result (if any).

We want to check whether \( Q \) always emits message \( m \).
Algorithm - Overview

Idea of the algorithm:

- Specify the property to analyse as a biprocess.
- Execute both processes ‘together’.
- If they don’t ‘do the same’, biprocess gets stuck and we have a (potential) error.
Property to analyse

We want to check whether $Q$ always emits message $m$.

We define:

$$S(\underline{ }) = (\nu k) (\bar{c}\langle\text{enc}_k(m)\rangle \mid c(x).\text{let } y = \_ \text{ in } \bar{c}\langle y\rangle)$$

Algorithm input:

$$S(\text{diff}[\text{dec}_k(x), m])$$
Property to analyse

We want to check whether Q always emits message $m$.

We define:

$$S(\_ ) = (\nu k) (\bar{c} \langle \text{enc}_k(m) \rangle \mid c(x).\text{let } y = \_ \text{ in } \bar{c}\langle y \rangle)$$

Algorithm input:

$$S(\text{diff}[\text{dec}_k(x), m])$$

with signature $\Sigma$:

$$\text{def}_{\Sigma} := \text{dec}_y(\text{enc}_y(x)) \rightarrow x$$
Algorithm - Overview

Idea of the algorithm:

- Specify the property to analyse as a biprocess.
- Execute both processes ‘together’.
- If they don’t ‘do the same’, biprocess gets stuck and we have a (potential) error.

How is this realized? By reduction to reachability.

- Generate Horn clauses for the attacker and the protocol,
- Use clauses to deduce *bad*.
Generate clauses

Clauses for the protocol:

$$S = (\nu k) \left( \bar{c} \langle \text{enc}_k(m) \rangle \mid c(x).\text{let } y = \text{diff}[\text{dec}_k(x), m] \text{ in } \bar{c} \langle y \rangle \right)$$

The clauses use the following primitives:

- $\text{msg}'(p_1, p_2, p'_1, p'_2)$
- $\text{input}'(p, p')$
- $\text{nounif}(p, p')$
- $\text{att}'(p, p')$
- bad
Generate clauses

Clauses for the protocol:

\[ S = (\nu k) (\bar{c}\langle\text{enc}_k(m)\rangle \mid c(x).\text{let } y = \text{diff}[\text{dec}_k(x), m] \text{ in } \bar{c}\langle y\rangle) \]

We maintain an environment and known facts:

\[ \rho_0 = a \mapsto (a[], a[]) \forall a \in \text{fn}(S) \]

\[ s = \varepsilon \]

\[ s' = \varepsilon \]

\[ H = \text{true} \]
Generate clauses

Clauses for the protocol:

\[ S = (\nu k) (\bar{c}\langle \text{enc}_k(m) \rangle \mid c(x).\text{let } y = \text{diff}[\text{dec}_k(x), m] \text{ in } \bar{c}\langle y \rangle) \]

\[ \rho = \rho_0[k \mapsto (k[], k[])] \]

\[ s = \varepsilon \]

\[ s' = \varepsilon \]

\[ H = \text{true} \]
Generate clauses

Clauses for the protocol:

\[ S = (\nu k) (\bar{c}\langle\text{enc}_k(m)\rangle \mid c(x).\text{let } y = \text{diff}[\text{dec}_k(x), m] \text{ in } \bar{c}\langle y\rangle) \]

Clauses:

\[ \rho = \rho_0[k \mapsto (k[], k[]) \mid \text{true } \rightarrow \text{msg}'(c[], \text{enc}_{k[]} (m[]), \ldots) \]

\[ s = \varepsilon \]

\[ s' = \varepsilon \]

\[ H = \text{true} \]
Generate clauses

Clauses for the protocol:

\[
S = (\nu k) (\bar{c}\langle\text{enc}_k(m)\rangle \mid c(x).\text{let } y = \text{diff}[\text{dec}_k(x), m] \text{ in } \bar{c}\langle y\rangle)
\]

Clauses:

\[
\rho = \rho_0[k \mapsto (k[], k[])]
\]

\[
s = \varepsilon
\]

\[
s' = \varepsilon
\]

\[
H = \text{msg}'(c[], x, c[], x')
\]

true \rightarrow \text{msg}'(c[], \text{enc}_{k[]}(m[]), \ldots )

true \rightarrow \text{input}'(c[], c[])
Generate clauses

Clauses for the protocol:

\[
S = (\nu k) (\bar{c}' \langle \text{enc}_k(m) \rangle \mid c(x). \text{let } y = \text{diff}[\text{dec}_k(x), m] \text{ in } \bar{c}' \langle y \rangle)
\]

Clauses:

\[
\rho = \rho_0[k \mapsto (k[], k[])] \\
\rho = \varepsilon \\
\rho' = \varepsilon \\
H = \text{msg}'(c[], x, c[], x')
\]

true \rightarrow \text{msg}'(c[], \text{enc}_k[](m[]), \ldots)

true \rightarrow \text{input}'(c[], c[])

Markus Rabe  Verification of Selected Equivalences
Analysing the let expression

\[ \text{let } y = \text{diff}[\text{dec}_k(x), m] \text{ in } \bar{c}\langle y \rangle \]

In general: In both processes the destructor could fail; 4 cases.
Analysing the let expression

\[
\text{let } y = \text{diff}[\text{dec}_k(x), m] \text{ in } \overline{\text{c}}\langle y \rangle
\]

In general: In both processes the destructor could fail; 4 cases.

- One of the two terms fails to evaluate:
  \[
  \text{msg}'(c[], x, c[], \text{enc}_{k[]}(y)) \land \text{nounif}(x, m) \rightarrow \text{bad}
  \]
  \[
  \text{msg}'(c[], \text{enc}_{k[]}(y), c[], x') \land \text{nounif}(\text{dec}_{k[]}(y), x') \rightarrow \text{bad}
  \]
Analysing the let expression

\[
\text{let } y = \text{diff}[\text{dec}_k(x), m] \text{ in } \bar{c}\langle y \rangle
\]

In general: In both processes the destructor could fail; 4 cases.

- One of the two terms fails to evaluate:
  \[
  \text{msg}'(c[], x, c[], \text{enc}_k\langle y \rangle) \wedge \text{nounif}(x, m) \rightarrow \text{bad}
  \]
  \[
  \text{msg}'(c[], \text{enc}_k\langle y \rangle, c[], x') \wedge \text{nounif}(\text{dec}_k\langle y \rangle, x') \rightarrow \text{bad}
  \]

- Both terms succeed to evaluate:
  Check \( \bar{c}\langle y \rangle \) with \( \rho = \rho_0[y \mapsto (z, m[])] \).
  \[
  \text{msg}'(c[], \text{enc}_k\langle z \rangle, c[], \text{enc}_k\langle z' \rangle) \rightarrow \text{msg}'(c[], z, c[], m[])
  \]
Analysing the let expression

\[ \text{let } y = \text{diff}[\text{dec}_k(x), m] \text{ in } \bar{c}\langle y \rangle \]

In general: In both processes the destructor could fail; 4 cases.

- One of the two terms fails to evaluate:
  \[ \text{msg}'(c[], x, c[], \text{enc}_k(y)) \land \text{nounif}(x, m) \rightarrow \text{bad} \]
  \[ \text{msg}'(c[], \text{enc}_k(y), c[], x') \land \text{nounif}(\text{dec}_k(y), x') \rightarrow \text{bad} \]

- Both terms succeed to evaluate:
  Check \( \bar{c}\langle y \rangle \) with \( \rho = \rho_0[y \mapsto (z, m[])] \).
  \[ \text{msg}'(c[], \text{enc}_k(z), c[], \text{enc}_k(z')) \rightarrow \text{msg}'(c[], z, c[], m[]) \]

- Both terms fail to evaluate: have to create clauses for else branch, which is empty here.
Protocol clauses

All protocol clauses:

\[ true \rightarrow \text{msg}'(c[], \text{enc}_{k[]} (m[]), \ldots) \]
\[ true \rightarrow \text{input}'(c[], c[]) \]
\[ \text{msg}'(c[], x, c[], \text{enc}_{k[]} (y)) \land \text{nounif}(x, m) \rightarrow \text{bad} \]
\[ \text{msg}'(c[], \text{enc}_{k[]} (y), c[], x') \land \text{nounif}(\text{dec}_{k[]} (y), x') \rightarrow \text{bad} \]
\[ \text{msg}'(c[], \text{enc}_{k[]} (z), c[], \text{enc}_{k[]} (z')) \rightarrow \text{msg}'(c[], z, c[], m[]) \]
Clauses for the attacker

The attacker ...

- ... knows all free names \( a \): \( \text{att}'(a[], a[]) \)
- ... can read from known channels:
  \[
  \text{msg}'(x, y, x', y') \land \text{att}'(x, x') \rightarrow \text{att}'(y, y')
  \]
- ... can generate messages:
  \[
  \text{att}'(x, x') \land \text{att}'(y, y') \rightarrow \text{msg}'(x, y, x', y')
  \]
- ... can evaluate constructors and destructors
Clauses for the attacker

The attacker ... 

- ... knows all free names $a$: att'(a[], a[])
- ... can read from known channels: 
  $\text{msg}'(x, y, x', y') \land \text{att}'(x, x') \rightarrow \text{att}'(y, y')$
- ... can generate messages: 
  $\text{att}'(x, x') \land \text{att}'(y, y') \rightarrow \text{msg}'(x, y, x', y')$
- ... can evaluate constructors and destructors

The attacker has found a flaw if ... 

- ... he can distinguish channels: 
  $\text{input}'(x, x') \land \text{msg}'(x, z, y', z') \land \text{nounif}(x', y') \rightarrow \text{bad}$
- ... a constructor succeeds in one process but fails in the other.
Resolution algorithm and theorem

We use the clauses and try to deduce bad:

\[ H \rightarrow C \quad F \land H' \rightarrow C' \]
\[ \sigma H \land \sigma H' \rightarrow \sigma C' \]

Given that \( C \) and \( F \) are unifiable.
Resolution algorithm and theorem

We use the clauses and try to deduce bad:

\[
\begin{align*}
H \rightarrow C & \quad F \land H' \rightarrow C' \\
\sigma H \land \sigma H' & \rightarrow \sigma C'
\end{align*}
\]

Given that \( C \) and \( F \) are unifiable.

**Theorem 3**

If bad is not a logical consequence of \( \mathcal{R}_{P_0} \), then \( P_0 \) satisfies observational equivalence.

Method is sound but incomplete.
Summary

What we have seen

- Process calculi
- Basics of applied $\pi$-calculus
- Observational equivalence
- Heart of the algorithm
Summary

What we have seen

• Process calculi
• Basics of applied $\pi$-calculus
• Observational equivalence
• Heart of the algorithm

What I left out

• Handling of equational theories
• Unifiability
• Solving algorithm
• Optimizations for protocols with multiple stages
Thank you for your attention!