Proving unlinkability using ProVerif through desynchronized bi-processes

Stéphanie Delaune
Univ Rennes, CNRS, IRISA, Spicy team

→ joint work with David Baelde and Alexandre Debant
CSF - July 10, 2023
Security protocol design is critical and error-prone as illustrated by many attacks: FREAK, Logjam, ... 

Use formal methods to improve confidence:

- prove the absence of attacks under certain assumptions; or
- identify weaknesses.

Many verification tools already exist:

- Proverif, Tamarin, AKISS, DeepSec, AVISPA, Squirrel, ...
Running example: Basic Hash protocol

- Each tag stores a secret key $k$ that is never updated.
- Readers have access to a database DB containing all the keys.
ProVerif in a nutshell

→ mainly developed by Bruno Blanchet (Prosecco team, Inria Paris)

http://proverif.inria.fr/

An automatic tool to analyse protocols in the symbolic model.

- successfully used for many large-scale case studies: TLS 1.3, ...
- protocols are modelled using a process algebra;
- both reachability and equivalence-based properties;
- security analysis done for an unbounded number of sessions;
- **No miracle**: the tool may return cannot be proved or never terminates.
Unlinkability

(ISO/IEC 15408)
“Ensuring that a user may make multiple uses of a service or resource without others being able to link these uses together.”

Informally, an observer/attacker can not observe the difference between:

1. a situation where the same device/tag may be used twice (or even more);
2. a situation where each device/tag is used at most once.
Unlinkability

(ISO/IEC 15408)
“Ensuring that a user may make multiple uses of a service or resource without others being able to link these uses together.”

Informally, an observer/attacker can not observe the difference between:

1. a situation where the same device/tag may be used twice (or even more);
2. a situation where each device/tag is used at most once.

More formally,

\[
! \text{new } k \text{. insert } \text{DB}(k) . ( ! \text{Tag}(k) \mid ! \text{Reader}) \]
\[
\approx
\]
\[
! \text{new } k \text{. insert } \text{DB}(k) . ( \text{Tag}(k) \mid ! \text{Reader})
\]

\[\rightarrow\] the notion of equivalence remains to be defined
State-of-the art

ProVerif (but also Tamarin) can only prove a restricted form of equivalence, namely **diff-equivalence**, which is too limiting to establish unlinkability.

---

State-of-the art

ProVerif (but also Tamarin) can only prove a restricted form of equivalence, namely **diff-equivalence**, which is too limiting to establish unlinkability.

Some solutions to overcome this limitation:

- Establish unlinkability using an **indirect approach** (sufficient conditions)
  
  e.g. [Solène Moreau PhD thesis, 21]

- Use **restrictions**: a feature available in Tamarin (2005), and in ProVerif (2022).

---

1[https://github.com/tamarin-prover/tamarin-prover/issues/324](https://github.com/tamarin-prover/tamarin-prover/issues/324)
State-of-the art

ProVerif (but also Tamarin) can only prove a restricted form of equivalence, namely \textit{diff-equivalence}, which is too limiting to establish unlinkability.

Some solutions to overcome this limitation:

- Establish unlinkability using an \textit{indirect approach} (sufficient conditions)
  
e.g. [Solène Moreau PhD thesis, 21]

- Use \textit{restrictions}: a feature available in Tamarin (2005), and in ProVerif (2022).
  \textbf{Tamarin}: incorrectly handled for equivalence\(^1\), now formally justify for Type-0 (very specific class) [Paradzik, 22]
  \textbf{ProVerif}: Need to be manipulated with a lot of care. \textit{Restrictions for equivalence discard \textit{bi-traces}!}

\(^1\)https://github.com/tamarin-prover/tamarin-prover/issues/324
Our contributions

We design a transformation (in 2 steps) allowing us to transform a ProVerif model $M$ into another one $M'$ such that:

If ProVerif succeeds on $M'$ then equivalence holds on $M$. 


Our contributions

We design a transformation (in 2 steps) allowing us to transform a ProVerif model $M$ into another one $M'$ such that:

If ProVerif succeeds on $M'$ then equivalence holds on $M$.

Our transformation contains two main steps:

1. We dissociate the two processes that forms that bi-process. Possible using the option: allowDiffPatterns
2. We generate some axioms (and prove them correct) to help the analysis.

The transformation has been implemented and successfully used on several case studies.
High-level view of ProVerif
Protocols as processes

→ a programming language with constructs for concurrency and communication

(applied-pi calculus [Abadi & Fournet, 01])

\[
P, Q \ := \ 0 \quad \text{null process}
\]

\[
| \quad \text{in}(c, x); P \quad \text{input}
\]

\[
| \quad \text{out}(c, M); P \quad \text{output}
\]

\[
| \quad \text{new } n; P \quad \text{name generation}
\]

\[
| \quad \text{let } x = D \text{ in } P \text{ else } Q \quad \text{conditional}
\]

\[
| \quad !P \quad \text{replication}
\]

\[
| \quad (P \mid Q) \quad \text{parallel composition}
\]
Protocols as processes

→ a programming language with constructs for concurrency and communication

(applied-pi calculus [Abadi & Fournet, 01])

\[
P, Q ::= 0 \quad \text{null process}
\]
\[
| \quad \text{input}
\]
\[
| \quad \text{output}
\]
\[
| \quad \text{name generation}
\]
\[
| \quad \text{conditional}
\]
\[
| \quad \text{replication}
\]
\[
| \quad \text{parallel composition}
\]
\[
| \quad \text{event}
\]
\[
| \quad \text{insertion}
\]
\[
| \quad \text{lookup}
\]
\[
| \quad \text{...}
\]
Messages/Computations as terms

Terms are built over a set of names $\mathcal{N}$, and function symbols $\Sigma_c \cup \Sigma_d$ equipped with an equational theory $E$ and rewriting rules for destructors.

Example:

- **constructor symbols**: $\Sigma_c = \{\langle \rangle, \text{proj}_1, \text{proj}_2, h, \text{true}\}$;
- $E = \{\text{proj}_1(\langle x_1, x_2 \rangle) = x_1, \; \text{proj}_2(\langle x_1, x_2 \rangle) = x_2\}$;
- **destructor symbols**: $\Sigma_d = \{\text{eq}\}$;
- **rewriting rule**: $\text{eq}(x, x) \rightarrow \text{true}$.
- all the function symbols are public (available to the attacker);
Going back to Basic Hash

We consider:

- \( T(k) = \text{new } n; \text{out}(c, \langle n, h(n, k) \rangle) \).
- \( R = \text{in}(c, y); \text{get } db(k) \text{ st. } \text{eq}(h(\text{proj}_1(y), k), \text{proj}_2(y)) \text{ in out}(c, ok) \text{ else out}(c, ko) \).

The real system corresponds to the following process:

\[
! R \mid (! \text{new } k; \text{insert keys}(k); ! T(k))
\]
Semantics (some selected rules)

Labelled transition system over configurations:

\[(\mathcal{P}; \Phi; S)\]

- Multiset of processes
- Frame
- Store
- Knowledge of the attacker
- Content of the database
Semantics (some selected rules)

Labelled transition system over configurations:

\[(P; \Phi; S)\]

- **Out** \((\{\text{out}(c, M); P\} \cup P; \Phi; S) \xrightarrow{\text{out}(c, w_i)} (\{P\} \cup P; \Phi \cup \{w_i \mapsto M\}; S)\) with \(i = |\Phi|\)

- **In** \((\{\text{in}(c, x); P\} \cup P; \Phi; S) \xrightarrow{\text{in}(c, R)} (\{P\{x \mapsto M\}\} \cup P; \Phi; S)\) with \(R\Phi \Downarrow =_E M\)

- **Get-T** \((\{\text{get tbl}(x) \text{ st. } D \text{ in } P \text{ else } Q\} \cup P; \Phi; S) \xrightarrow{\tau} (\{P\{x \mapsto M\}\} \cup P; \Phi; S)\) with \(\text{tbl}(M) \in S\), and \(D\{x \mapsto M\} \Downarrow =_E \text{true}\)

...
Trace equivalence

Static equivalence between frames: $\Phi \sim_s \Phi'$.
Any test that holds in $\Phi$ also holds in $\Phi'$ (and conversely).

Example: $\{w_1 \mapsto \langle n, h(n, k) \rangle; w_2 \mapsto k\} \not\sim_s \{w_1 \mapsto \langle n, h(n, k) \rangle; w_2 \mapsto k'\}$

$\rightarrow$ with the test $h(\text{proj}_1(w_1), w_2) = \text{proj}_2(w_1)$.
Trace equivalence

Static equivalence between frames: $\Phi \sim_s \Phi'$.
Any test that holds in $\Phi$ also holds in $\Phi'$ (and conversely).

Example: $\{w_1 \mapsto \langle n, h(n, k) \rangle; w_2 \mapsto k \} \not\sim_s \{w_1 \mapsto \langle n, h(n, k) \rangle; w_2 \mapsto k' \}$
$\rightarrow$ with the test $h(\text{proj}_1(w_1), w_2) \not= \text{proj}_2(w_1)$.

Trace equivalence between configurations: $K \approx_t K'$.
For any execution trace $K \xrightarrow{\text{tr}} (P; \Phi; S)$ there exists an execution $K' \xrightarrow{\text{tr}} (P'; \Phi'; S')$
such that $\Phi \sim_s \Phi'$ (and conversely)

Example:

$\begin{align*}
! R & | (! \text{new} \ k; \text{insert keys}(k); ! T(k)) \approx_t ! R & | (! \text{new} \ k; \text{insert keys}(k); T(k)) \\
\rightarrow & \text{an equivalence that ProVerif (and also Tamarin) is not able to prove directly.}
\end{align*}$
Going back to diff-equivalence

How it works (or not)?

• form a bi-process $B$ using the operator $\text{choice}[M_L, M_R]$;
• both sides of the bi-processes have to evolve \textit{simultaneously} to be declared in
  diff-equivalence (and this implies $\text{fst}(B) \approx_t \text{snd}(B)$)

$\rightarrow$ the semantics is given by a labelled transition system over bi-configurations
($\mathcal{P}; \Phi; S$) where messages and computations may contain the \text{choice} operator.
Going back to diff-equivalence

How it works (or not)?

- form a bi-process $B$ using the operator $\text{choice} [M_L, M_R]$;
- both sides of the bi-processes have to evolve simultaneously to be declared in diff-equivalence (and this implies $\text{fst}(B) \approx_t \text{snd}(B)$)

$\Rightarrow$ the semantics is given by a labelled transition system over bi-configurations $(\mathcal{P}; \Phi; S)$ where messages and computations may contain the $\text{choice}$ operator.

Example - Basic Hash protocol

$$B = \text{!}R \mid (\text{!new } k; \text{!new } kk; \text{insert } db(\text{choice}[k, kk]); T(\text{choice}[k, kk]))$$

We have that:

- $\text{fst}(B) = \text{!}R \mid \text{!new } k; \text{!insert } db(k); T(k) \quad \text{(* real situation *)}$
- $\text{snd}(B) = \text{!}R \mid \text{!!new } kk; \text{insert } db(kk); T(kk) \quad \text{(* ideal situation *)}$
Why diff-equivalence is too strong?

\[ B = \text{!}R \mid (! \text{new } k; \text{!new } kk; \text{insert } db(\text{choice}[k, kk]); T(\text{choice}[k, kk])) \]

Let’s consider a scenario with:

- 1 reader;
- 2 tags: \( T(\text{choice}[k, kk_1]) \), and \( T(\text{choice}[k, kk_2]) \).
Why diff-equivalence is too strong?

\[ B = !R \mid (\text{!new } k; \text{!new } kk; \text{insert } db(\text{choice}[k, kk]); \text{T(choice}[k, kk])) \]

Let's consider a scenario with:

- 1 reader;
- 2 tags: \(T(\text{choice}[k, kk_1])\),
  and \(T(\text{choice}[k, kk_2])\).

The frame contains: \(w_1 = \langle n_1, h(n, \text{choice}[k, kk_1]) \rangle\).
Why diff-equivalence is too strong?

\[ B = !R | (!\text{new } k; \!\text{new } kk; \text{insert } db(\text{choice}[k, kk]); \text{T}(\text{choice}[k, kk])) \]

Let's consider a scenario with:

- 1 reader;
- 2 tags: \( T(\text{choice}[k, kk_1]) \), and \( T(\text{choice}[k, kk_2]) \).

<table>
<thead>
<tr>
<th>DB</th>
<th>left</th>
<th>right</th>
</tr>
</thead>
<tbody>
<tr>
<td>line 1</td>
<td>( k )</td>
<td>( kk_1 )</td>
</tr>
<tr>
<td>line 2</td>
<td>( k )</td>
<td>( kk_2 )</td>
</tr>
</tbody>
</table>

The frame contains: \( w_1 = \langle n_1, h(n, \text{choice}[k, kk_1]) \rangle \).

On line 2, with \( w_1 \) in input for process \( R \), the bi-process \( B \) will diverge.

\[ \rightarrow \] Thus, Proverif returns \textit{cannot be proved} on this example.
Beyond ProVerif 2.00 [Blanchet et al., 2022]

×→ support for axioms, lemmas, and restrictions as in Tamarin.

**Syntax:** This gives the user the possibility to write correspondence queries of the form:

\[
\text{event}(e_1) \land \ldots \land \text{event}(e_n) \Rightarrow \psi
\]

with \( \psi, \psi' = \text{true} | \text{false} | \text{event}(e) | M = N | M \neq N | \psi \land \psi' | \psi \lor \psi' \)

**Semantics:** An execution trace \( T \) satisfies \( \rho \) (noted \( T \vdash \rho \)) if whenever \( T \) contains instances of \( \text{event}(e_i) \) at some timepoint \( \tau_i \) for each \( i \), then \( T \) also satisfies \( \psi \).
Desynchronized bi-processes

We consider an extension of standard bi-processes using the allowDiffPatterns option available in ProVerif since 2018.

→ this allow us to use choice[$x^L, x^R$] for variable bindings in let, get, and input.

Example:

\[
B = \text{in}(c, \text{choice}[^L, ^R]); \text{out}(c, \langle ^L, ^R \rangle).
\]
Desynchronized bi-processes

We consider an extension of standard bi-processes using the `allowDiffPatterns` option available in ProVerif since 2018.

→ this allow us to use `choice[x^L, x^R]` for variable bindings in `let`, `get`, and `input`.

Example: \( B = \text{in}(c, \text{choice}[x^L, x^R]); \text{out}(c, \langle x^L, x^R \rangle) \).

**Warning !**

Closed bi-processes of this form have a well-defined semantics in ProVerif and we can study diff-equivalence (convergence of all the bi-traces). However, this does **not** imply:

\[ \text{fst}(B) \approx_t \text{snd}(B) \]
Our transformation
In a nutshell

Main Goal

Transform a ProVerif model $M$ of unlinkability into another model $M'$ such that:

- diff-equivalence is verified with ProVerif on the transformed model $M'$; and
- diff-equivalence on $M'$ implies trace equivalence for the original model $M$. 

Two main steps

1. duplicate the get instructions in $M$ to dissociate the two parts of the bi-process;
2. add some axioms to help ProVerif to reason on our new model.
Main Goal

Transform a ProVerif model $M$ of unlinkability into another model $M'$ such that:

- diff-equivalence is verified with ProVerif on the transformed model $M'$; and
- diff-equivalence on $M'$ implies trace equivalence for the original model $M$.

Two main steps

1. duplicate the get instructions in $M$ to dissociate the two parts of the bi-process;
2. add some axioms to help ProVerif to reason on our new model.
Desynchronizing the two parts of the biprocess

Instead of performing a get instruction to access a bi-record in the keys table, we perform two get instructions in a row to access two records in the keys table. → This allows us to choose two different records for the left and for the right.
Desynchronizing the two parts of the biprocess

Instead of performing a get instruction to access a bi-record in the keys table, we perform two get instructions in a row to access two records in the keys table.

→ This allows us to choose two different records for the left and for the right.

Example:

\[
in(c, \text{choice}[x^L, x^R]);
\]

\[
\text{get } db(\text{choice}[y^L, _]) \text{ st. eq}(\text{proj}_2(x^L), h(\text{proj}_1(x^L, y^L))) \text{ in}
\]

\[
\text{get } db(\text{choice}[_, y^R]) \text{ st. eq}(\text{proj}_2(x^R), h(\text{proj}_1(x^R, y^R))) \text{ in out}(c, \text{choice}[\text{ok, ok}])
\]

else

\[
\text{get } db(\text{choice}[_, y^R]) \text{ st. eq}(\text{proj}_2(x^R), h(\text{proj}_1(x^R, y^R))) \text{ in out}(c, \text{choice}[\text{ko, ok}])
\]

else out(c, choice[ko, ko])

Refining the analysis in the failure branches

We illustrate this on a very simple example.

Before, . . .

\[
B = \text{insert \ } tbl(\text{ok}); \\
\text{get \ } tbl(x) \text{ st. true in out}(c, \text{ok}) \\
\text{else out}(c, \text{choice}[\text{ok}_L, \text{ok}_R])
\]

. . . and ProVerif can not proved equivalence (whereas it holds).
Refining the analysis in the failure branches

We illustrate this on a very simple example.

After, . . .

\[ B = \text{event(Inserted(ok)); insert tbl(ok)}; \]
\[ \text{get tbl(x) st. true in out(c, ok)} \]
\[ \text{else event(Fail()); out(c, choice[ok_L, ok_R])} \]

. . . together with the following axiom:

\[ \text{event(Fail()) \land event(Inserted(choice[y_L, y_R]))} \Rightarrow \text{false.} \]

\[ \rightarrow \text{On this model, ProVerif is able to conclude that equivalence holds.} \]
Refining the analysis in the failure branches

We illustrate this on a very simple example.

After, . . .

\[ B = \text{event(Inserted(ok))}; \text{insert \ tbl(ok)}; \]
\[ \text{get \ tbl}(x) \text{ st. true in out}(c, \text{ok}) \]
\[ \text{else } \text{event(Fail())}; \text{out}(c, \text{choice}[\text{ok}_L, \text{ok}_R]) \]

. . . together with the following axiom:

\[ \text{event(Fail())} \land \text{event(Inserted(\text{choice}[y^L, y^R]))} \Rightarrow \text{false}. \]

→ On this model, ProVerif is able to conclude that equivalence holds.

Going back to the Basic Hash protocol

\[ \text{event(FailL}(x^L)) \land \text{event(Inserted(\text{choice}[y^L, y^R]))} \Rightarrow \text{proj}_2(x^L) \neq h(\text{proj}_1(x^L), y^L) \]
\[ \text{event(FailR}(x^R)) \land \text{event(Inserted(\text{choice}[y^L, y^R]))} \Rightarrow \text{proj}_2(x^R) \neq h(\text{proj}_1(x^R), y^R) \]
Main result

**Theorem**

Let $\mathcal{M} = (B_0, \emptyset, \mathcal{A}_x, \mathcal{L})$ be a ProVerif standard model ($B_0$ is separated), and $\mathcal{M}' = (B', \emptyset, \mathcal{A}_x \cup \mathcal{A}_x', \mathcal{L})$ be the model obtained after applying our transformation. Moreover, we assume that:

- for all $\varrho \in \mathcal{A}_x$, we have that $\text{traces}(B_0) \vdash \varrho$;
- for all $\varrho \in \mathcal{A}_x$, we have that $\text{traces}(B') \vdash \varrho$;
- ProVerif returns diff-equivalence is true on $\mathcal{M}'$.

We conclude that $\text{fst}(B_0) \approx_t \text{snd}(B_0)$. 
Case studies

Implementation
The two steps of the transformation have been implemented (≈ 2k Ocaml LoC).

Case studies
Basic Hash, Hash-Lock, Feldhofer, a variant of LAK, OSK.
→ ProVerif is able to conclude on all these examples!
Our approach significantly improves automation regarding unlinkability.

Future Work

- better integration in ProVerif;
- beyond unlinkability (e.g. privacy in e-voting);
- other difficulty: dealing with mutable states (GSVerif).

→ if you are interested in this subject (going beyond diff-equivalence using ProVerif), you should attend Vincent’s talk (tomorrow afternoon).

Questions ?