Proving unlinkability using ProVerif through desynchronized bi-processes

Stéphanie Delaune
Univ Rennes, CNRS, IRISA, Spicy team
CORGIS - February 6, 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.”
(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,

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

\[ \rightarrow \text{the notion of equivalence remains to be defined} \]
ProVerif (but also Tamarin) can only prove a restricted form of equivalence, namely diff-equivalence, which is too limiting to establish unlinkability.

---

1https://github.com/tamarin-prover/tamarin-prover/issues/324
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
ProVerif (but also Tamarin) can only prove a restricted form of equivalence, namely **diff-equivalence**, which is too limiting to establishunlinkability.

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).
  
  **Tamarin**: incorrectly handled for equivalence\(^1\), now formally justify for Type-0 (very specific class) [Paradzik, 22]

  **ProVerif**: Need to be manipulated with a lot of care. **Restrictions for equivalence discard 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 $\mathcal{M}$ into another one $\mathcal{M}'$ such that:

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$. 

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$.
Our contributions

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

If ProVerif succeeds on $\mathcal{M}'$ then equivalence holds on $\mathcal{M}$.

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 sucessfully 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])

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

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

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

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

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

| !P \quad \text{replication} |

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

→ a programming language with constructs for **concurrency** and **communication**

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

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

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

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

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

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

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

\[
\quad \mid (P \mid Q) \quad \text{parallel composition}
\]

\[
\quad \mid \text{event}(e); P \quad \text{event}
\]

\[
\quad \mid \text{insert } tbl(M); P \quad \text{insertion}
\]

\[
\quad \mid \text{get } tbl(x) \text{ st. } D \text{ in } P \text{ else } Q \quad \text{lookup}
\]

\[
\quad \mid \ldots
\]
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.
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**: $\Sigma_c = \{\langle \rangle, \text{proj}_1, \text{proj}_2, \text{h}, \text{true}\}$;
- $E = \{\text{proj}_1(\langle x_1, x_2 \rangle) = x_1, \ \text{proj}_2(\langle x_1, x_2 \rangle) = x_2\}$;
- **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);
Messages/Computations as terms

Terms are built over a set of names \( N \), 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);

Let \( \Phi = \{ w \mapsto \langle n, h(n, k) \rangle \} \), and \( R = \text{eq}(h(\text{proj}_1(w), k), \text{proj}_2(w)) \). We have that

\[
R\Phi =_E \text{eq}(h(n, k), h(n, k)) \rightarrow \text{ok} \quad \text{(written } R\Phi \Downarrow = \text{ok)}
\]
We consider:

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

The real system corresponds to the following process:

\[
! R | (! \text{new } k; \text{insert keys}(k); ! T(k))
\]
Labelled transition system over configurations:

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

- multiset of processes
- frame
- knowledge of the attacker
- store
- content of the database
Semantics (some selected rules)

Labelled transition system over configurations:

\begin{align*}
  \mathcal{LTS} = (\mathcal{P}; \Phi; \mathcal{S})
\end{align*}

- multiset of processes
- frame
- knowledge of the attacker
- store
- content of the database

**OUT**

\[ \{\text{out}(c, M); P\} \uplus \mathcal{P}; \Phi; \mathcal{S} \overset{\text{out}(c,w_i)}{\longrightarrow} \{P\} \uplus \mathcal{P}; \Phi \cup \{w_i \mapsto M\}; \mathcal{S} \text{ with } i = |\Phi| \]

**IN**

\[ \{\text{in}(c, x); P\} \uplus \mathcal{P}; \Phi; \mathcal{S} \overset{\text{in}(c,R)}{\longrightarrow} \{P\{x \mapsto M\}\} \uplus \mathcal{P}; \Phi; \mathcal{S} \text{ with } R\Phi \downarrow =_{E} M \]

**GET-T**

\[ \{\text{get tbl}(x) \text{ st. } D \text{ in } P \text{ else } Q\} \uplus \mathcal{P}; \Phi; \mathcal{S} \overset{\tau}{\longrightarrow} \{P\{x \mapsto M\}\} \uplus \mathcal{P}; \Phi; \mathcal{S} \]

with \( \text{tbl}(M) \in \mathcal{S} \), and \( D\{x \mapsto M\} \downarrow =_{E} \text{true} \)
Semantics (some selected rules)

Labelled transition system over configurations:

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

- multiset of processes
- frame
- store
- knowledge of the attacker
- content of the database

**Out**

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

**In**

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

**Get-T**

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

\[\rightarrow \text{traces}(K) = \text{the set of execution traces starting from the configuration } K.\]
Trace equivalence

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

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

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

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

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

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

Example:
\[ !R | (!\text{new } k; \text{insert keys}(k); !T(k)) \approx_t !R | (\text{!new } k; \text{insert keys}(k); T(k)) \]
\[ \rightarrow \text{ an equivalence that ProVerif (and also Tamarin) is not able to prove directly.} \]
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.
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 $\text{fst}(B) \approx_t \text{snd}(B)$)

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

Example - Basic Hash protocol

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

We have that

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

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

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?

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

Let's consider a scenario with:

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

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

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

Let's consider a scenario with:

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

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.

\[ R = \text{in}(c, y); \text{get } db(k) \text{ st. eq}(h(\text{proj}_1(y), k), \text{proj}_2(y)) \text{ in out}(c, ok) \text{ else out}(c, ko). \]

\[ \rightarrow \] Thus, Proverif returns \textbf{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 \( \psi \).
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 \( \psi, \psi' = \text{true} \mid \text{false} \mid \text{event}(e) \mid M = N \mid M \neq N \mid \psi \land \psi' \mid \psi \lor \psi' \)

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 \( \psi \).

Example

\[ \text{event(once}(x_{id}, x_{sid})) \land \text{event(once}(x_{id}, y_{sid})) \Rightarrow x_{sid} = y_{sid} \]
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 \( \psi, \psi' = \text{true} \mid \text{false} \mid \text{event}(e) \mid M = N \mid M \neq N \mid \psi \land \psi' \mid \psi \lor \psi' \)

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

Example

\[ \text{event}(\text{once}(x_{\text{id}}, x_{\text{sid}})) \land \text{event}(\text{once}(x_{\text{id}}, y_{\text{sid}})) \Rightarrow x_{\text{sid}} = y_{\text{sid}} \]

Warning! When used on a biprocess, a (bi)restriction will discard bi-execution.

\[ \text{event}(\text{once}(\text{choice}[\_\_\_, x_{\text{id}}], \text{choice}[\_\_\_, x_{\text{sid}}])) \land \text{event}(\text{once}(\text{choice}[\_\_\_, x_{\text{id}}], \text{choice}[\_\_\_, y_{\text{sid}}])) \Rightarrow x_{\text{sid}} = y_{\text{sid}} \]
We consider an extension of standard bi-processes using the `allowDiffPatterns` option available in ProVerif since 2018.

→ systematic use of `choice[x^L, x^R]` for variable bindings in let, get, and input.

Example:

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

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

→ systematic use of choice[$x^L$, $x^R$] for variable bindings in let, get, and input.

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

→ a standard bi-process can be written as a separated bi-process, i.e. $\text{vars(fst}(B)) \cap \text{vars(snd}(B)) = \emptyset$. 
Desynchronized bi-processes

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

\[ \rightarrow \text{systematic use of } \text{choice}[x^L, x^R] \text{ for variable bindings in let, get, and input.} \]

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

\[ \rightarrow \text{a standard bi-process can be written as a separated bi-process, i.e. } \]
\[ \text{vars}(\text{fst}(B)) \cap \text{vars}(\text{snd}(B)) = \emptyset. \]

Example: \( B \) is not separated. Actually, \( \text{fst}(B) \) is not closed, and makes no sense.

Non-separated and closed bi-processes have a well-defined semantics in ProVerif and we can study whether diff-equivalence holds on them. However, this does not imply:

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

Main Goal

Transform a ProVerif model $\mathcal{M}$ of unlinkability into another model $\mathcal{M}'$ such that:

- diff-equivalence is verified with ProVerif on the transformed model $\mathcal{M}'$; and
- diff-equivalence on $\mathcal{M}'$ implies trace equivalence for the original model $\mathcal{M}$.
Main Goal

Transform a ProVerif model $\mathcal{M}$ of unlinkability into another model $\mathcal{M}'$ such that:

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

Two main steps

1. duplicate the get instructions in $\mathcal{M}$ 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:

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

else

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

We illustrate this on a very simple example.

Before, . . .

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

. . . and ProVerif can not proved equivalence (whereas it holds).
We illustrate this on a very simple example.

After, ... 

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

... together with the following axiom:

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

\[ \rightarrow \] 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, ... 

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

...together with the following axiom:

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

\[ \rightarrow \text{On this model, ProVerif is able to conclude that equivalence holds.} \]

Going back to the Basic Hash protocol

\[ \text{event(FailL(x_L))} \land \text{event(Inserted(diff[y_L, y_R]))} \Rightarrow \text{proj}_2(x_L) \neq h(\text{proj}_1(x_L), y_L) \]
\[ \text{event(FailR(x_R))} \land \text{event(Inserted(diff[y_L, y_R]))} \Rightarrow \text{proj}_2(x_R) \neq h(\text{proj}_1(x_R), y_R) \]
**Main result**

<table>
<thead>
<tr>
<th>Theorem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Let ( M = (B_0, \emptyset, Ax, L) ) be a ProVerif standard model (( B_0 ) is separated), and ( M' = (B'_0, \emptyset, Ax \cup Ax', L) ) be the model obtained after applying our transformation. Moreover, we assume that:</td>
</tr>
<tr>
<td>• for all ( \varrho \in Ax ), we have that ( \text{traces}(B_0) \vdash \varrho );</td>
</tr>
<tr>
<td>• for all ( \varrho \in Ax ), we have that ( \text{traces}(B'_0) \vdash \varrho );</td>
</tr>
<tr>
<td>• ProVerif returns diff-equivalence is true on ( M' ).</td>
</tr>
<tr>
<td>We conclude that ( \text{fst}(B_0) \approx_t \text{snd}(B_0) ).</td>
</tr>
</tbody>
</table>
Implementation
The two steps of the transformation have been implemented ($\approx 2k$ Ocaml LoC).

Case studies
Basic Hash, Hash-Lock, Feldhofer, a variant of LAK, OSK.
$\rightarrow$ ProVerif is able to conclude on all these examples!
Implementation

The two steps of the transformation have been implemented ($\approx 2k$ Ocaml LoC).

Case studies

Basic Hash, Hash-Lock, Feldhofer, a variant of LAK, OSK.

$\rightarrow$ ProVerif is able to conclude on all these examples!
Conclusion & Future Work

Our approach significantly improves automation regarding unlinkability.

Future Work

- better integration in ProVerif;
- beyond unlinkability;
- Other difficulty: dealing with mutable states.