Proving unlinkability using ProVerif through desynchronized bi-processes

Stéphanie Delaune
Univ Rennes, CNRS, IRISA, Spicy team

CORGIS - February 6, 2023

Formal verification of cryptographic protocols

Security protocol design is critical and error-prone as illustrated by many attacks: FREAK, Logjam, ...

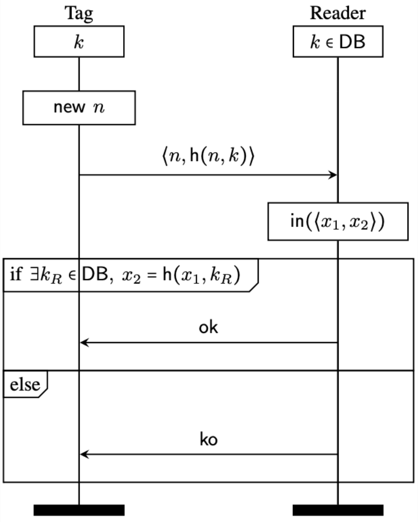
Use formal methods to improve confidence:

- 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

$\longrightarrow$ mainly developped by Bruno Blanchet (Prosecco team, Inria Paris)

`http://proverif.inria.fr/`

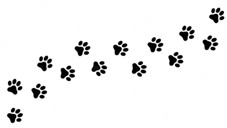
An automatic tool to analyse protocols in the symbolic model.

- 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)

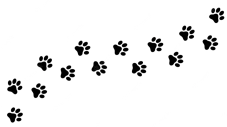
"Ensuring that a user may make multiple uses of a service or resource without others being able to link these uses together."



Unlinkability

(ISO/IEC 15408)

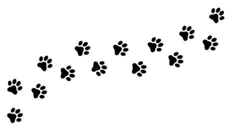
"Ensuring that a user may make multiple uses of a service or resource without others being able to link these uses together."



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.

(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,

$$!\,\mathtt{new}\ \ k.\mathtt{insert}\ \mathrm{DB}(k).\Big(!\ \mathrm{Tag}(k)\ \ |\ !\ \mathrm{Reader}\Big)$$
$$\overset{?}{\approx}$$
$$!\,\mathtt{new}\ \ k.\mathtt{insert}\ \mathrm{DB}(k).\Big(\ \mathrm{Tag}(k)\ \ |\ !\ \mathrm{Reader}\Big)$$

$\longrightarrow$ 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 limitating to establish unlinkability.

[1] 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 diff-equivalence, which is too limitating 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

State-of-the art

ProVerif (but also Tamarin) can only prove a restricted form of equivalence, namely diff-equivalence, which is too limitating 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).
  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 $\mathcal{M}$.

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

$\longrightarrow$ a programming language with constructs for concurrency and communication

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

$P, Q \quad := \quad 0$		null process
$\mid$	$\mathrm{in}(c, x); P$	input
$\mid$	$\mathrm{out}(c, M); P$	output
$\mid$	$\mathtt{new}\ n; P$	name generation
$\mid$	$\mathtt{let}\ x = D\ \mathtt{in}\ P\ \mathtt{else}\ Q$	conditional
$\mid$	$!P$	replication
$\mid$	$(P \mid Q)$	parallel composition

Protocols as processes

$\longrightarrow$ a programming language with constructs for concurrency and communication

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

$$
\begin{array}{rcll}
P, Q & := & 0 & \text{null process} \\
 & \mid & \mathrm{in}(c, x); P & \text{input} \\
 & \mid & \mathrm{out}(c, M); P & \text{output} \\
 & \mid & \mathtt{new}\ n; P & \text{name generation} \\
 & \mid & \mathtt{let}\ x = D\ \mathtt{in}\ P\ \mathtt{else}\ Q & \text{conditional} \\
 & \mid & !P & \text{replication} \\
 & \mid & (P \mid Q) & \text{parallel composition} \\
 & \mid & \mathtt{event}(e); P & \text{event} \\
 & \mid & \mathtt{insert}\ tbl(M); P & \text{insertion} \\
 & \mid & \mathtt{get}\ tbl(x)\ \mathtt{st.}\ D\ \mathtt{in}\ P\ \mathtt{else}\ Q & \text{lookup} \\
 & \mid & \ldots &
\end{array}
$$

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.

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

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

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

$$R\Phi =_{\mathsf{E}} \mathsf{eq}(\mathsf{h}(n, k), \mathsf{h}(n, k)) \to \mathsf{ok} \text{ (written } R\Phi\Downarrow = \mathsf{ok})$$

Going back to Basic Hash

We consider:

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

The real system corresponds to the following process:

$$! \,\mathsf{R} \mid (!\, \mathtt{new}\ k; \mathtt{insert}\ keys(k);\ !\, \mathsf{T}(k))$$

Semantics (some selected rules)

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:

$$(\mathcal{P}; \Phi; \mathcal{S})$$

multiset of processes

frame
knowledge of the attacker

store
content of the database

$$\textsc{Out} \quad (\{\mathrm{out}(c, M); P\} \uplus \mathcal{P}; \Phi; \mathcal{S}) \xrightarrow{\mathrm{out}(c,\mathrm{w}_i)} (\{P\} \uplus \mathcal{P}; \Phi \cup \{\mathrm{w}_i \mapsto M\}; \mathcal{S}) \text{ with } i = |\Phi|$$

$$\textsc{In} \quad (\{\mathrm{in}(c, x); P\} \uplus \mathcal{P}; \Phi; \mathcal{S}) \xrightarrow{\mathrm{in}(c,R)} (\{P\{x \mapsto M\}\} \uplus \mathcal{P}; \Phi; \mathcal{S}) \quad \text{with } R\Phi{\Downarrow} =_{\mathsf{E}} M$$

$$\textsc{Get-T} \quad (\{\mathtt{get}\ \mathit{tbl}(x)\ \mathtt{st.}\ D\ \mathtt{in}\ P\ \mathtt{else}\ Q\} \uplus \mathcal{P}; \Phi; \mathcal{S}) \xrightarrow{\tau} (\{P\{x \mapsto M\}\} \uplus \mathcal{P}; \Phi; \mathcal{S})$$

$$\text{with } \mathit{tbl}(M) \in \mathcal{S}, \text{ and } D\{x \mapsto M\}{\Downarrow} =_{\mathsf{E}} \mathsf{true}$$

Semantics (some selected rules)

Labelled transition system over configurations:

$$(\mathcal{P}; \Phi; \mathcal{S})$$

multiset of processes

frame
knowledge of the attacker

store
content of the database

$$\textsc{Out} \quad (\{\mathrm{out}(c, M); P\} \uplus \mathcal{P}; \Phi; \mathcal{S}) \xrightarrow{\mathrm{out}(c,\mathrm{w}_i)} (\{P\} \uplus \mathcal{P}; \Phi \cup \{\mathrm{w}_i \mapsto M\}; \mathcal{S}) \text{ with } i = |\Phi|$$

$$\textsc{In} \quad (\{\mathrm{in}(c, x); P\} \uplus \mathcal{P}; \Phi; \mathcal{S}) \xrightarrow{\mathrm{in}(c,R)} (\{P\{x \mapsto M\}\} \uplus \mathcal{P}; \Phi; \mathcal{S}) \quad \text{with } R\Phi\Downarrow =_\mathsf{E} M$$

$$\textsc{Get-T} \quad (\{\mathrm{get}\ tbl(x)\ \mathtt{st.}\ D\ \mathtt{in}\ P\ \mathtt{else}\ Q\} \uplus \mathcal{P}; \Phi; \mathcal{S}) \xrightarrow{\tau} (\{P\{x \mapsto M\}\} \uplus \mathcal{P}; \Phi; \mathcal{S})$$
$$\text{with } tbl(M) \in \mathcal{S}, \text{ and } D\{x \mapsto M\}\Downarrow =_\mathsf{E} \mathrm{true}$$

$\longrightarrow$ $\mathtt{traces}(K)$ = 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 Φ also holds in Φ' (and conversely).

Example: $\{\mathsf{w}_1 \mapsto \langle n, \mathsf{h}(n, k)\rangle;\ \mathsf{w}_2 \mapsto k\} \not\sim_s \{\mathsf{w}_1 \mapsto \langle n, \mathsf{h}(n, k)\rangle;\ \mathsf{w}_2 \mapsto k'\}$

$\longrightarrow$ with the test $\mathsf{h}(\mathsf{proj}_1(\mathsf{w}_1), \mathsf{w}_2) \stackrel{?}{=} \mathsf{proj}_2(\mathsf{w}_1)$.

Trace equivalence

Static equivalence between frames: $\Phi \sim_s \Phi'$.

Any test that holds in Φ also holds in Φ' (and conversely).

Example: $\{\mathsf{w}_1 \mapsto \langle n, \mathsf{h}(n, k)\rangle;\ \mathsf{w}_2 \mapsto k\} \not\sim_s \{\mathsf{w}_1 \mapsto \langle n, \mathsf{h}(n, k)\rangle;\ \mathsf{w}_2 \mapsto k'\}$

$\longrightarrow$ with the test $\mathsf{h}(\mathsf{proj}_1(\mathsf{w}_1), \mathsf{w}_2) \stackrel{?}{=} \mathsf{proj}_2(\mathsf{w}_1)$.

Trace equivalence between configurations: $K \approx_t K'$.

For any execution trace $K \xrightarrow{\mathsf{tr}} (\mathcal{P}; \Phi; \mathcal{S})$ there exists an execution $K' \xrightarrow{\mathsf{tr}} (\mathcal{P}'; \Phi'; \mathcal{S}')$ such that $\Phi \sim_s \Phi'$ (and conversely)

Example:

$$!\,\mathrm{R} \mid (!\,\mathtt{new}\ k; \mathtt{insert}\ keys(k); !\,\mathrm{T}(k)) \approx_t !\,\mathrm{R} \mid (!\,\mathtt{new}\ k; \mathtt{insert}\ keys(k);\ \mathrm{T}(k))$$

$\longrightarrow$ 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 $\text{choice}[M_{\mathsf{L}}, M_{\mathsf{R}}]$;
- both sides of the bi-processes have to evolve simulatenously to be declared in diff-equivalence (and this implies $\text{fst}(B) \approx_t \text{snd}(B)$)

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

Going back to diff-equivalence

How it works (or not)?

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

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

Example - Basic Hash protocol

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

We have that

- $\mathsf{fst}(B) = \,!\mathrm{R} \mid !\,\mathtt{new}\ k;\ !\mathtt{insert}\ db(k); \mathrm{T}(k)$ `(* real situation *)`
- $\mathsf{snd}(B) = !\mathrm{R} \mid !\,!\mathtt{new}\ kk; \mathtt{insert}\ db(kk); \mathrm{T}(kk)$ `(* ideal situation *)`

Why diff-equivalence is too strong?

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

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?

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

Let's consider a scenario with:

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

DB	left	right
line 1	k	kk_1
line 2	k	kk_2

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

Why diff-equivalence is too strong?

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

Let's consider a scenario with:

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

DB	left	right
line 1	k	kk_1
line 2	k	kk_2

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

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

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

$\longrightarrow$ Thus, Proverif returns `cannot be proved` on this example.

Beyond ProVerif 2.00 [Blanchet et al., 2022]

$\longrightarrow$ support for axioms, lemmas, and restrictions as in Tamarin.

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

$$\mathtt{event}(e_1) \wedge \ldots \wedge \mathtt{event}(e_n) \Rightarrow \psi$$

with $\psi, \psi' = \text{true} \mid \text{false} \mid \mathtt{event}(e) \mid M = N \mid M \neq N \mid \psi \wedge \psi' \mid \psi \vee \psi'$

Semantics: An execution trace T satisfies ρ (noted $T \vdash \rho$) if whenever T contains instances of $\mathtt{event}(e_i)$ at some timepoint τ_i for each i, then T also satisfies ψ.

Beyond ProVerif 2.00 [Blanchet et al., 2022]

$\longrightarrow$ support for axioms, lemmas, and restrictions as in Tamarin.

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

$$\mathtt{event}(e_1) \wedge \ldots \wedge \mathtt{event}(e_n) \Rightarrow \psi$$

with $\psi, \psi' = \mathsf{true} \mid \mathsf{false} \mid \mathtt{event}(e) \mid M = N \mid M \neq N \mid \psi \wedge \psi' \mid \psi \vee \psi'$

Semantics: An execution trace T satisfies ρ (noted $T \vdash \rho$) if whenever T contains instances of $\mathtt{event}(e_i)$ at some timepoint τ_i for each i, then T also satisfies ψ.

Example

$$\mathtt{event}(once(x_{id}, x_{sid})) \wedge \mathtt{event}(once(x_{id}, y_{sid})) \implies x_{sid} = y_{sid}$$

Beyond ProVerif 2.00 [Blanchet et al., 2022]

$\longrightarrow$ support for axioms, lemmas, and restrictions as in Tamarin.

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

$$\mathtt{event}(e_1) \wedge \ldots \wedge \mathtt{event}(e_n) \Rightarrow \psi$$

with $\psi, \psi' = \text{true} \mid \text{false} \mid \mathtt{event}(e) \mid M = N \mid M \neq N \mid \psi \wedge \psi' \mid \psi \vee \psi'$

Semantics: An execution trace T satisfies ρ (noted $T \vdash \rho$) if whenever T contains instances of $\mathtt{event}(e_i)$ at some timepoint τ_i for each i, then T also satisfies ψ.

Example

$$\mathtt{event}(once(x_{id}, x_{sid})) \wedge \mathtt{event}(once(x_{id}, y_{sid})) \implies x_{sid} = y_{sid}$$

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

$$\begin{aligned} & \mathtt{event}(once(\text{choice}[\_, x_{id}], \text{choice}[\_, x_{sid}])) \\ \wedge\ & \mathtt{event}(once(\text{choice}[\_, x_{id}], \text{choice}[\_, y_{sid}])) \implies x_{sid} = y_{sid} \end{aligned}$$

Desynchronized bi-processes

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

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

Desynchronized bi-processes

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

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

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

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

Desynchronized bi-processes

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

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

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

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

Example: B is not separated. Actually, $\mathsf{fst}(B)$ 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:

$$\mathsf{fst}(B) \approx_t \mathsf{snd}(B)$$

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 $\mathcal{M}$.

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 $\mathcal{M}$.

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.

$\longrightarrow$ 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.

$\longrightarrow$ This allows us to choose two different records for the left and for the right.

Example:

$$
\begin{array}{l}
\mathtt{in}(c, \mathtt{diff}[x^{\mathsf{L}}, x^{\mathsf{R}}]);\\
\mathtt{get}\ db(\mathtt{diff}[y^{\mathsf{L}}, \_])\ \mathtt{st.}\ \mathsf{eq}(\mathsf{proj}_2(x^{\mathsf{L}}), \mathsf{h}(\mathsf{proj}_1(x^{\mathsf{L}}, y^{\mathsf{L}})))\ \mathtt{in}\\
\quad \mathtt{get}\ db(\mathtt{diff}[\_, y^{\mathsf{R}}])\ \mathtt{st.}\ \mathsf{eq}(\mathsf{proj}_2(x^{\mathsf{R}}), \mathsf{h}(\mathsf{proj}_1(x^{\mathsf{R}}, y^{\mathsf{R}})))\ \mathtt{in}\ \mathsf{out}(c, \mathsf{choice}[\mathsf{ok}, \mathsf{ok}])\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{else}\ \mathsf{out}(c, \mathsf{choice}[\mathsf{ok}, \mathsf{ko}])\\
\mathtt{else}\\
\quad \mathtt{get}\ db(\mathtt{diff}[\_, y^{\mathsf{R}}])\ \mathtt{st.}\ \mathsf{eq}(\mathsf{proj}_2(x^{\mathsf{R}}), \mathsf{h}(\mathsf{proj}_1(x^{\mathsf{R}}, y^{\mathsf{R}})))\ \mathtt{in}\ \mathsf{out}(c, \mathsf{choice}[\mathsf{ko}, \mathsf{ok}])\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{else}\ \mathsf{out}(c, \mathsf{choice}[\mathsf{ko}, \mathsf{ko}])
\end{array}
$$

Refining the analysis in the failure branches

We illustrate this on a very simple example.

Before, ...

$$\begin{aligned} B = \ & \texttt{insert}\ \mathit{tbl}(\mathsf{ok}); \\ & \mathsf{get}\ \mathit{tbl}(x)\ \mathsf{st.\ true\ in}\ \texttt{out}(c, \mathsf{ok}) \\ & \qquad\qquad \mathsf{else}\ \texttt{out}(c, \mathsf{choice}[\mathsf{ok_L}, \mathsf{ok_R}]) \end{aligned}$$

... 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, ...

$$
\begin{aligned}
B = {} & \texttt{event}(\mathsf{Inserted}(\mathsf{ok}));\ \texttt{insert}\ tbl(\mathsf{ok});\\
& \quad \mathsf{get}\ tbl(x)\ \mathsf{st.\ true\ in}\ \texttt{out}(c, \mathsf{ok})\\
& \qquad\qquad \mathsf{else}\ \texttt{event}(\mathsf{Fail}());\texttt{out}(c, \mathsf{choice}[\mathsf{ok}_\mathsf{L}, \mathsf{ok}_\mathsf{R}])
\end{aligned}
$$

... together with the following axiom:

$$\texttt{event}(\mathsf{Fail}()) \wedge \texttt{event}(\mathsf{Inserted}(\texttt{diff}[y^\mathsf{L}, y^\mathsf{R}])) \Rightarrow \mathsf{false}.$$

$\longrightarrow$ 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, …

$$B = \mathtt{event}(\mathsf{Inserted}(\mathsf{ok}));\ \mathtt{insert}\ tbl(\mathsf{ok});$$
$$\text{get } tbl(x) \text{ st. true in } \mathtt{out}(c, \mathsf{ok})$$
$$\text{else } \mathtt{event}(\mathsf{Fail}());\mathtt{out}(c, \mathsf{choice}[\mathsf{ok}_\mathsf{L}, \mathsf{ok}_\mathsf{R}])$$

… together with the following axiom:

$$\mathtt{event}(\mathsf{Fail}()) \wedge \mathtt{event}(\mathsf{Inserted}(\mathtt{diff}[y^\mathsf{L}, y^\mathsf{R}])) \Rightarrow \mathsf{false}.$$

$\longrightarrow$ On this model, ProVerif is able to conclude that equivalence holds.

Going back to the Basic Hash protocol

$$\mathtt{event}(\mathsf{FailL}(x^\mathsf{L})) \wedge \mathtt{event}(\mathsf{Inserted}(\mathtt{diff}[y^\mathsf{L}, y^\mathsf{R}])) \Rightarrow \mathsf{proj}_2(x^\mathsf{L}) \neq \mathsf{h}(\mathsf{proj}_1(x^\mathsf{L}), y^\mathsf{L})$$
$$\mathtt{event}(\mathsf{FailR}(x^\mathsf{R})) \wedge \mathtt{event}(\mathsf{Inserted}(\mathtt{diff}[y^\mathsf{L}, y^\mathsf{R}])) \Rightarrow \mathsf{proj}_2(x^\mathsf{R}) \neq \mathsf{h}(\mathsf{proj}_1(x^\mathsf{R}), y^\mathsf{R})$$

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

We conclude that $\mathsf{fst}(B_0) \approx_t \mathsf{snd}(B_0)$.

Case studies

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.

$\longrightarrow$ 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.

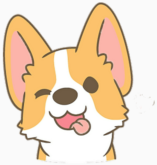
$\longrightarrow$ ProVerif is able to conclude on all these examples !

Conclusion & Future Work

Our approach significantly improves automation regarding unlinkability.

Future Work

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



Questions ?