Proving Unlinkability using ProVerif through Desynchronized Bi-Processes

David Baelde, Alexandre Debant, Stéphanie Delaune

▶ **To cite this version:**

David Baelde, Alexandre Debant, Stéphanie Delaune. Proving Unlinkability using ProVerif through Desynchronized Bi-Processes. 2022. hal-03674979v1

HAL Id: hal-03674979

https://inria.hal.science/hal-03674979v1

Preprint submitted on 25 May 2022 (v1), last revised 30 Jan 2023 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Proving Unlinkability using ProVerif through Desynchronised Bi-Processes

David Baelde[1], Alexandre Debant[2], and Stéphanie Delaune[1]

[1] Univ Rennes, CNRS, IRISA, France
[2] Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France

Abstract. Unlinkability is a privacy property of crucial importance for several systems such as mobile phones or RFID chips. Analysing this security property is very complex, and highly error-prone. Therefore, formal verification with machine support is desirable. Unfortunately, existing techniques are not sufficient to directly apply verification tools to automatically prove unlinkability. In this paper, we overcome this limitation by defining a simple transformation that will exploit several features recently introduced in the tool ProVerif. This transformation, together with some generic axioms, allow the tool to successfully conclude on several case studies. We have implemented our approach, effectively obtaining direct proofs of unlinkability on several protocols that were, until now, out of reach of automatic verification tools. Moreover, our approach is not specific to unlinkability and could, in principle, be useful in other contexts.

1 Introduction

RFID tags are popping up everywhere. They are present in credit cards enabling wireless payments, in wireless keys for opening cars, in e-passport for storing fingerprints and iris scans, and in e-ticketing systems e.g. public transport, amusement parks, ... Citizens do not expect to be traceable just because they carry an e-passport or a credit card. Many applications blatantly ignore such concerns. As a famous and widespread example, contactless credit cards reveal card numbers in clear [3]. More interestingly, subtle tracing attacks may be found even when this security goal is explicitly taken into account. For instance, the BAC protocol for e-passports can be exploited to trace citizens [13]. The attack relies on the possibility for an attacker to trace a passport by observing which error is obtained when replaying a particular message.

According to the ISO/IEC 15408-2 standard, protocols are said to provide unlinkability if they "ensure that a user may make multiple uses of [them] without others being able to link these uses together". This is often defined as the fact that an attacker should not be able to distinguish a scenario in which the same agent (i.e., the user) is involved in many sessions from one that involved different agents in each session. Following this intuition, slightly different definitions have been proposed, e.g. [21, 1, 9, 2]. A comparison between some of these definitions may

be found in [10]. In this work, we consider a variant of the definition originally proposed in [1], and revisited in [2] to make it more suitable to analyse RFID protocols manipulating a database.

Designing protocols which actually achieve the required security goals is well-known to be error-prone. A common good practice is to formally analyse protocols using symbolic techniques and tools (e.g. TLS 1.3 [16, 5]). Aiming at machine support is really relevant since manual proofs are error-prone, tedious and hardly verifiable. Moreover, new protocols are developed quite frequently and need to be verified quickly. These symbolic techniques are mature for reachability properties like confidentiality or authentication, and some tools exist, e.g. Tamarin [20] and ProVerif [6], to ease the verification process. This approach has been extended to deal with privacy properties that are usually expressed through equivalences [7, 4]. Nevertheless, verifying a security property in such a setting, and especially unlinkability as defined above, remains a difficult problem. Actually, ProVerif and Tamarin can only prove a restricted form of equivalence, namely *diff-equivalence*, which (in its standard form) is too limiting to prove unlinkability.

Our contributions. We show that this limitation can be overcome by defining a simple transformation that exploits several features of ProVerif that have been recently introduced. More specifically, our transformation relies on a recent extension of the notion of biprocess, which ProVerif uses to establish equivalences. Roughly, a biprocess embeds two processes in it, which are declared equivalent if they can always evolve simultaneously, even for actions that are not observable by the attacker. We propose to dissociate the two processes that form the biprocess when executing some specific unobservable instructions on which the two processes may diverge. This is possible thanks to a recent extension of the tool, which was initially proposed to establish frame opacity in [19]. This transformation alone is not sufficient for the tool to conclude on our case studies because of internal over-approximations. However, adding some generic axioms, as authorised by the latest version of ProVerif presented in [8], improves the accuracy enough to successfully conclude on all our examples. We have implemented our approach and applied it on several case studies. We have obtained, for the first time, direct proofs of unlinkability for several protocols.

2 Modelling protocols

We recall first how cryptographic messages are modelled as terms in ProVerif. Building on this, we shall see how protocols are modelled as processes, and how protocol indistinguishability is formally expressed through process equivalences.

2.1 Term algebra

As usual in the symbolic setting, messages are modelled through a term algebra. We assume an infinite set $\mathcal{N}$ of *names* used to represent atomic data such as keys, or nonces, as well as an infinite set Σ_0 of *constants* used to represent atomic data

known by the attacker; and two infinite and disjoint sets of *variables* $\mathcal{X}$ and $\mathcal{W}$. Variables in $\mathcal{X}$ are used to refer e.g. to input messages, and variables in $\mathcal{W}$, called *handles*, are used as pointers to messages learned by the attacker. We assume a finite *signature* Σ of *function symbols* with their arity partitioned into two disjoint subsets Σ_c, and Σ_d representing respectively the *constructors*, and *destructors*, and we denote $\Sigma_c^+ = \Sigma_c \uplus \Sigma_0$, and $\Sigma^+ = \Sigma \uplus \Sigma_0$.

We note $\mathcal{T}(\mathcal{F}, D)$ the set of terms built from elements of the set of atomic data D by applying function symbols in the signature $\mathcal{F}$. We refer to elements of $\mathcal{T}(\Sigma^+, \mathcal{N} \cup \mathcal{X})$ as *expressions* and elements of $T(\Sigma_c^+, \mathcal{N} \cup \mathcal{X})$ as (constructor) terms. We define $\mathsf{vars}(M)$ as the set of variables that occur in an expression M. A term (resp. an expression) is *ground* when it contains no variable. We finally define $\mathsf{names}(M)$ as the set of names occurring in M. We will sometimes use $\mathsf{names}(M)$ and $\mathsf{vars}(D)$ as vectors, meaning that we select an arbitrary ordering over these finite sets.

In order to provide a meaning to constructor symbols, we equip (constructor) terms with an equational theory. We assume a set E of equations over $\mathcal{T}(\Sigma_c, \mathcal{X})$ and we define $=_{\mathsf{E}}$ as the smallest congruence containing E that is closed under substitutions and under bijective renaming. In addition, the semantics of destructor symbols is given by a set R of *ordered rewriting rules* of the form $\mathsf{g}(M_1, \ldots, M_n) \to M_0$ with $M_0, M_1, \ldots, M_n \in \mathcal{T}(\Sigma_c, \mathcal{X})$. A ground expression D can be rewritten in D' if there is a position p in D, a rewrite rule $\mathsf{g}(M_1, \ldots, M_n) \to M_0$ and a substitution θ from variables to ground terms such that $D|_p =_{\mathsf{E}} \mathsf{g}(M_1\theta, \ldots, M_n\theta)$, and $D' =_{\mathsf{E}} D[M_0\theta]_p$, i.e. D in which the subterm at position p has been replaced by $M_0\theta$. In the case where more than one rule may be applied at position p, only the first such rule can be effectively used. Given a ground expression D, it may be possible to rewrite it (in an arbitrary number of steps) into a ground (constructor) term M: in that case, this term is noted $D{\Downarrow}$, and we say that D *evaluates to* $D{\Downarrow}$. We write $D{\Downarrow\!\!\!\!/}$ when no such term exists, and say that the *computation fails*.

For modelling purposes, we split the signature $\Sigma_c \uplus \Sigma_d$ into two parts, Σ_{pub} and Σ_{priv}. An attacker builds his own messages by applying public function symbols to terms he already knows, and which are available to him through variables in $\mathcal{W}$. Formally, a computation done by the attacker is a *recipe*, i.e. a term in $\mathcal{T}(\Sigma_{\mathsf{pub}}^+, \mathcal{W})$ where $\Sigma_{\mathsf{pub}}^+ = \Sigma_{\mathsf{pub}} \uplus \Sigma_0$.

Example 1. We consider $\Sigma_c = \{\langle\,\rangle,\ \mathsf{proj}_1,\ \mathsf{proj}_2,\ \mathsf{h}\}$, $\Sigma_d = \emptyset$, and we assume that all the function symbols are public. The symbol $\langle\,\rangle$ of arity 2 is used to model the pairing operator, and we have projection symbols proj_1 and proj_2 (both of arity 1) to access the components of a pair. The symbol h of arity 2 represents a keyed hash function. Given $n_T, k \in \mathcal{N}$, we have that $\langle n_T, \mathsf{h}(n_T, k)\rangle$ is a term.

The standard equational theory $\mathsf{E}_{\mathsf{pair}}$ modelling the pairing operator contains the two following equations: $\mathsf{proj}_1(\langle x_1, x_2\rangle) = x_1$, and $\mathsf{proj}_2(\langle x_1, x_2\rangle) = x_2$.

Example 2. To illustrate the notion of destructor symbols and ordered rewriting systems, we consider the signature $\Sigma = \Sigma_c \cup \Sigma_d$ such that $\Sigma_c = \{\mathtt{true}, \mathtt{false}\}$ and $\Sigma_d = \{\mathtt{eq}\}$ together with the rewriting rules $\mathtt{eq}(x, x) \to \mathtt{true}$, and $\mathtt{eq}(x, y) \to$

`false`, in this order. We assume again that all these function symbols are public. Let $\mathsf{w} \in \mathcal{W}$, and $\mathsf{k}_0 \in \Sigma_0$. We have that $R = \mathsf{eq}(\mathsf{h}(\mathsf{proj}_1(\mathsf{w}), \mathsf{k}_0), \mathsf{proj}_2(\mathsf{w}))$ is a recipe, and $D = R\{\mathsf{w} \mapsto \langle n_T, \mathsf{h}(n_T, \mathsf{k}_0)\rangle\}$, i.e. R in which the occurrences of w are replaced by $\langle n_T, \mathsf{h}(n_T, \mathsf{k}_0)\rangle$, is a ground expression such that $D{\Downarrow} = \mathtt{true}$.

2.2 Process algebra

We consider two disjoint sets $\mathcal{C}_{\mathsf{pub}}$ and $\mathcal{C}_{\mathsf{priv}}$ of *channel names*. The channels in $\mathcal{C}_{\mathsf{pub}}$ are assumed to be public, whereas those in $\mathcal{C}_{\mathsf{priv}}$ are not. We denote $\mathcal{C} = \mathcal{C}_{\mathsf{pub}} \uplus \mathcal{C}_{\mathsf{priv}}$. We also consider a set Σ_e of *event* symbols and a set Σ_t of *table* symbols. An event symbol *evt* of arity k may be applied to k terms to form an event $evt(M_1, \ldots, M_k)$. We assume that all table symbols are of arity 1. Events will be used to record particular points in executions, while tables are used to dynamically store data across sessions.

Protocols will be modelled as processes given by the grammar below, where $c \in \mathcal{C}$, $M \in \mathcal{T}(\Sigma_c^+, \mathcal{N} \cup \mathcal{X})$ is a term, $x \in \mathcal{X}$, $n \in \mathcal{N}$, $c_{\mathsf{priv}} \in \mathcal{C}_{\mathsf{priv}}$, $tbl \in \Sigma_t$, $D \in \mathcal{T}(\Sigma^+, \mathcal{N} \cup \mathcal{X})$ is an expression and e is an event:

$$
\begin{array}{lll}
P, Q := 0 & & \mid \mathtt{new}\ n; P \\
\quad \mid\ \mathsf{out}(c, M); P & & \mid \mathtt{new}\ c_{\mathsf{priv}}; P \\
\quad \mid\ \mathsf{in}(c, x); P & & \mid \mathtt{let}\ x = D\ \mathtt{in}\ P\ \mathtt{else}\ Q \\
\quad \mid\ \mathtt{event}(e); P & & \mid \mathtt{insert}\ tbl(M); P \\
\quad \mid\ !P & & \mid \mathtt{get}\ tbl(x)\ \mathtt{st}.\ D\ \mathtt{in}\ P\ \mathtt{else}\ Q \\
\quad \mid\ (P \mid Q) & & \mid i : P
\end{array}
$$

We will not comment on the standard constructs of this grammar (i.e. null process, name and channel restrictions, input, output, parallel composition, replication). The process $\mathtt{insert}\ tbl(M); P$ inserts a record in the table *tbl* then runs P, and $\mathtt{get}\ tbl(x)\ \mathtt{st}.\ D\ \mathtt{in}\ P\ \mathtt{else}\ Q$ looks for a record M in the table *tbl* such that the expression D evaluates to $\mathtt{true}$ when x is bound to M. When such a record is found, it runs P in which x has been replaced by M. Otherwise, it runs Q. The construct $\mathtt{let}\ x = D\ \mathtt{in}\ P\ \mathtt{else}\ Q$ combines a computation with a conditional: it attempts to evaluate D, executes P with x bound to the resulting message upon success, and Q otherwise. The construct $i : P$ models that the process P must be executed in phase i only. Finally, the last construct simply records the event e before executing P.

The constructs in and $\mathtt{let}$ bind their variables x in their sub-processes P. The get construct binds x in D and P. The free variables $\mathsf{fv}(P)$ of a process P are defined accordingly. A process P is *closed* when $\mathsf{fv}(P) = \emptyset$.

The operational semantics of processes is given by a labelled transition system over configurations (denoted by K) representing the current state of the process.

Definition 1. *A* configuration *is a tuple* $(\mathcal{P}; \Phi; \mathcal{S}; i)$ *with* $i \in \mathbb{N}$ *and such that:*

- $\mathcal{P}$ *is a multiset of closed processes;*
- $\Phi = \{\mathsf{w}_1 \mapsto M_1, \ldots, \mathsf{w}_n \mapsto M_n\}$ *is a* frame *representing the knowledge of the adversary, i.e. a substitution where* $\mathsf{w}_1, \ldots, \mathsf{w}_n$ *are handles and* $M_1, \ldots, M_n$ *are ground terms;*

– *$\mathcal{S}$ is a set of elements of the form $tbl(M)$ with $tbl \in \Sigma_t$, and M a ground term. This set $\mathcal{S}$ represents the content of all the tables.*

We sometimes write P for the configuration $(\{0 : P\}; \emptyset; \emptyset; 0)$, where we explicitly indicate that the process is intended to execute in phase 0. The labelled transitions are defined using usual rules (see e.g. [15, **?**]). For instance, an adversary can send any message of her knowledge on a public channel through the In rule, while communications are synchronous on private channels:

$$\textsc{In}\quad (\{i : \mathsf{in}(c, x); P\} \uplus \mathcal{P}; \Phi; \mathcal{S}; i) \xrightarrow{\mathsf{in}(c,R)} (\{i : P\{x \mapsto M\}\} \uplus \mathcal{P}; \Phi; \mathcal{S}; i)$$
$$\text{if } c \in \mathcal{C}_{\mathsf{pub}} \text{ and there exists a recipe } R \text{ such that } R\Phi\Downarrow =_{\mathsf{E}} M$$

$$\textsc{IO}\quad (\{i : \mathsf{in}(c, x); P\} \uplus \{i : \mathsf{out}(c, M); Q\} \uplus \mathcal{P}; \Phi; \mathcal{S}; i)$$
$$\xrightarrow{\tau} (\{i : P\{x \mapsto M\}\} \uplus \{i : Q\} \uplus \mathcal{P}; \Phi; \mathcal{S}; i) \text{ if } c \in \mathcal{C}_{\mathsf{priv}}$$

Regarding tables, the semantics is given by the rules Insert, Get-Then, Get-Else described below:

$$(\{i : \texttt{insert}\ tbl(M); P\} \uplus \mathcal{P}\ \Phi\ \mathcal{S}; i) \xrightarrow{\tau} (\{i : P\} \uplus \mathcal{P}; \Phi; \mathcal{S} \cup \{tbl(M)\}; i)$$

$(\{i : \texttt{get}\ tbl(x)\ \texttt{st.}\ D\ \texttt{in}\ P\ \texttt{else}\ Q\} \uplus \mathcal{P}; \Phi; \mathcal{S}; i) \xrightarrow{\tau} (\{i : P\{x \mapsto M\}\} \uplus \mathcal{P}; \Phi; \mathcal{S}; i)$
if there exists M such that $tbl(M) \in \mathcal{S}$, and $D\{x \mapsto M\}$ evaluates to true

$(\{i : \texttt{get}\ tbl(x)\ \texttt{st.}\ D\ \texttt{in}\ P\ \texttt{else}\ Q\} \uplus \mathcal{P}; \Phi, \mathcal{S}; i) \xrightarrow{\tau} (\{i : Q\} \uplus \mathcal{P}; \Phi; \mathcal{S}; i)$
if for any M such that $tbl(M) \in \mathcal{S}$, $D\{x \mapsto M\}$ does not evaluate to true

We also have rules for the other constructs, and some rules to deal with phases:

$$\textsc{Move}\qquad (\mathcal{P}; \Phi; \mathcal{S}; i) \xrightarrow{\mathsf{phase}\ (i+1)} (\mathcal{P}; \Phi; \mathcal{S}; i + 1)$$

$$\textsc{Phase}\qquad (\{i : i' : P\} \cup \mathcal{P}; \Phi; \mathcal{S}; i) \xrightarrow{\tau} (\{i' : P\} \cup \mathcal{P}; \Phi; \mathcal{S}; i)$$

An *execution trace* is a finite sequence $K_0 \xrightarrow{\ell_1} K_1 \xrightarrow{\ell_2} \ldots \xrightarrow{\ell_n} K_n$. The associated sequence of observables of such a trace is the sequence $\ell_1, \ldots, \ell_n$ from which τ actions have been removed. We denote $\texttt{traces}(K_0)$ the set of execution traces from K_0. Given an integer j such that $0 \leq j \leq n$, we denote $T[j]$ the configuration K_j, i.e the configuration at step j in trace T.

Example 3. For illustration purposes, we consider the Basic Hash protocol as described in [9]. Each tag stores a secret key that is never updated, and the readers have access to a database containing all these keys. We have that:

$$T \to R : \langle n_T, \mathsf{h}(n_T, k)\rangle$$

where n_T is a fresh nonce and k is the secret key. When receiving a message, the reader checks that it is a pair whose second element is a hash of the first element of the pair with one of the keys from the database. The protocol can be modelled in our syntax as follows:

$$P_{\mathsf{BH}} = (0 : !\ \texttt{new}\ k;\ \texttt{insert}\ keys(k); 1 : !\ P_{\mathsf{T}})\ \mid\ (1 : !\ P_{\mathsf{R}})$$

where $k, n_T \in \mathcal{N}$, $y, z \in \mathcal{X}$, $\mathsf{ok}, \mathsf{error} \in \Sigma_0$, $keys \in \Sigma_t$, and $c_R, c_T \in \mathcal{C}_{\mathsf{pub}}$. Moreover, we have that:

– $P_{\mathsf{R}} = \mathsf{in}(c_R, x)$; $\mathtt{get}\ keys(y)\ \mathtt{st}.\ D\ \mathtt{in}\ \mathsf{out}(c_R, \mathsf{ok})\ \mathtt{else}\ \mathsf{out}(c_R, \mathsf{error})$;
– $D = \mathsf{eq}(\mathsf{proj}_2(x), \mathsf{h}(\mathsf{proj}_1(x), y))$; and
– $P_{\mathsf{T}} = \mathtt{new}\ n_T$; $\mathsf{out}(c_T, \langle n_T, \mathsf{h}(n_T, k)\rangle)$.

2.3 Trace equivalence

We now define the notion of *trace equivalence* on which our definition of unlinkability is based. Intuitively, two processes are trace equivalent if for each trace of one process, there is an indistinguishable trace of the other process. To define this formally, we first introduce *static equivalence* between frames. Intuitively, an attacker can distinguish two frames Φ and Φ' if there exists a test that fails in Φ and succeeds in Φ' (or the contrary).

Definition 2. *Two frames Φ and Φ' are in* static equivalence*, written $\Phi \sim_s \Phi'$, when $\mathrm{dom}(\Phi) = \mathrm{dom}(\Phi')$ and for any recipes R_1 and R_2, we have that:*

$$R_1\Phi\Downarrow =_{\mathsf{E}} R_2\Phi\Downarrow,\ \textit{if, and only if,}\ R_1\Phi'\Downarrow =_{\mathsf{E}} R_2\Phi'\Downarrow.$$

Example 4. Continuing with the Basic Hash protocol, we consider the two following frames ($n_T, n'_T, k, k' \in \mathcal{N}$):

– $\Phi_{\mathsf{diff}} = \{\mathsf{w}_1 \mapsto \langle n_T, \mathsf{h}(n_T, k)\rangle;\ \mathsf{w}_2 \mapsto \langle n'_T, \mathsf{h}(n'_T, k')\rangle\}$; and
– $\Phi_{\mathsf{same}} = \{\mathsf{w}_1 \mapsto \langle n_T, \mathsf{h}(n_T, k)\rangle;\ \mathsf{w}_2 \mapsto \langle n'_T, \mathsf{h}(n'_T, k)\rangle\}$.

We have that $\Phi_{\mathsf{diff}} \sim_s \Phi_{\mathsf{same}}$. Intuitively, an attacker can not distinguish two outputs from the same tag from two outputs performed by different tags.

Given two execution traces T and T' leading respectively to configurations $K = (\mathcal{P}; \Phi; \mathcal{S}; i)$ and $K' = (\mathcal{P}'; \Phi'; \mathcal{S}'; i')$, we write $T \cong T'$ when $i = i'$, T and T' have the same sequence of observables, and $\Phi \sim_s \Phi'$.

Definition 3. *Let K and K' be two configurations. We say that K and K' are in* trace equivalence*, written $K \approx_t K'$, when, for any execution trace $T \in \mathtt{traces}(K)$, there exists $T' \in \mathtt{traces}(K')$ such that $T \cong T'$, and conversely.*

We rely on this notion of trace equivalence to model unlinkability following the definition originally proposed in [1] (using a stronger notion of equivalence) and used also in some other papers e.g. [18]. The purpose of this work is not to provide a discussion on how to model unlinkability but to show how to overcome the limitations of the existing tool ProVerif to automate proofs of unlinkability. Therefore, we simply illustrate the definition on the Basic Hash example.

Example 5. Going back to our running example, unlinkability will be modelled considering the following equivalence:

$$P_{\mathsf{BH}} \stackrel{?}{\approx}_t (0 :!\ \mathtt{new}\ k;\ \mathtt{insert}\ keys(k); 1 :\ P_{\mathsf{T}})\ \mid\ (1 :!\ P_{\mathsf{R}})$$

This equivalence states that an attacker cannot distinguish the situation where many tags play multiple sessions from the situation where each tag can play only once. Note that the replication ! in front of process P_{T} has been removed on the right hand side of the equivalence.

Tools like Tamarin or ProVerif cannot currently prove this equivalence, as observed in [2]. We will explain how ProVerif works and why it fails on our example in Section 3, before explaining our approach to solve this issue in Section 4.

3 Proving equivalences using ProVerif

ProVerif can prove an equivalence between two processes P and Q that differ only by the terms and expressions they contain. In Section 3.1, we explain this notion of biprocesses, and the notion of diff-equivalence that ProVerif considers. Then, we introduce some features that have been recently introduced in ProVerif [8] and on which our analysis will rely.

3.1 Biprocesses and diff-equivalence

Two processes that only differ by terms and expressions can be given by a single process in which the special function symbol $\mathtt{diff}[\cdot,\cdot]$ is used. Intuitively, the first argument corresponds to the term used in the first process, whereas the second one is the term used in the other process. More formally, the grammar we consider for biprocesses is the same as the one introduced previously with the addition of $\mathtt{diff}[M, M']$ for terms and the systematic use of $\mathtt{diff}[x, y]$ for variable bindings: for instance an input biprocess is of the form $\mathsf{in}(c, \mathtt{diff}[x, y]); P$. The latter addition means that we do not consider ProVerif's standard biprocesses but their extension allowed by the `allowDiffPatterns` option.

When P is a biprocess we define $\mathsf{fst}(P)$ as the process obtained by replacing any subterm $\mathtt{diff}[M, M']$ by M in P. We define similarly $\mathsf{fst}(M)$ when M is a biterm, and define symmetrically $\mathsf{snd}(P)$ and $\mathsf{snd}(M)$. ProVerif's standard biprocesses, without diff operators on variable bindings, can equivalently be characterised in our setting as *separated* biprocesses: assuming a partition $\mathcal{X} = \mathcal{X}^{\mathsf{L}} \uplus \mathcal{X}^{\mathsf{R}}$, a biprocess is separated when its variable bindings are of the form $\mathtt{diff}[x^{\mathsf{L}}, x^{\mathsf{R}}]$ with $x^{\mathsf{L}} \in \mathcal{X}^{\mathsf{L}}$ and $x^{\mathsf{R}} \in \mathcal{X}^{\mathsf{R}}$, and all its subterms M are such that $\mathsf{vars}(\mathsf{fst}(M)) \subseteq \mathcal{X}^{\mathsf{L}}$ and $\mathsf{vars}(\mathsf{snd}(M)) \subseteq \mathcal{X}^{\mathsf{R}}$. When a closed biprocess P is separated, $\mathsf{fst}(P)$ and $\mathsf{snd}(P)$ are closed processes too.

Example 6. The biprocess $P = \mathsf{in}(c, \mathtt{diff}[x^{\mathsf{L}}, x^{\mathsf{R}}]).\mathsf{out}(c, \langle x^{\mathsf{L}}, x^{\mathsf{R}}\rangle)$ is a closed biprocess since the variables x^{L} and x^{R} are bound by the input construct, but it is not separated. Moreover, $\mathsf{fst}(P) = \mathsf{in}(c, x^{\mathsf{L}}).\mathsf{out}(c, \langle x^{\mathsf{L}}, x^{\mathsf{R}}\rangle)$ is not closed.

The semantics of biprocesses will be defined by a labelled transition system over biconfigurations of the form $(\mathcal{P}; \Phi; \mathcal{S}; i)$ where $\mathcal{P}$ is a multiset of biprocesses, Φ is a substitution mapping variables to biterms called a *biframe*, $\mathcal{S}$ is a *bistore*, i.e. a set of items of the form $tbl(M)$ where tbl is a table identifier and M is a biterm, and i an integer. When Φ is a biframe, we write Φ_{fst} for the frame of domain $\mathrm{dom}(\Phi)$ such that $\Phi_{\mathsf{fst}}(\mathsf{w}) = \mathsf{fst}(\Phi(\mathsf{w}))$. For a bistore $\mathcal{S}$, we define $\mathcal{S}_{\mathsf{fst}} = \{tbl(\mathsf{fst}(M)) \mid tbl(M) \in \mathcal{S}\}$. We define similarly Φ_{snd} and $\mathcal{S}_{\mathsf{snd}}$.

We now define the transitions $\rightarrow_b$ over biconfigurations, as an adaptation of the transitions for regular configurations. For separated biprocesses, $(\mathcal{P}; \Phi; \mathcal{S}; i) \rightarrow_b$

$(\mathcal{P}';\Phi';\mathcal{S}';i)$ implies $(\mathsf{fst}(\mathcal{P});\Phi_{\mathsf{fst}};\mathcal{S}_{\mathsf{fst}};i) \to (\mathsf{fst}(\mathcal{P}');\Phi'_{\mathsf{fst}};\mathcal{S}'_{\mathsf{fst}};i)$ and similarly for the second projection. This does not hold in general and, as illustrated by Example 6, it may not even make sense.

The input and internal communication rules become:

$$\textsc{In}_b \quad (\{i : \mathsf{in}(c, \mathtt{diff}[x^{\mathsf{L}}, x^{\mathsf{R}}]); P\} \uplus \mathcal{P}; \Phi; \mathcal{S}; i) \xrightarrow{\mathsf{in}(c,R)} (\{i : P\{x^{\mathsf{L}} \mapsto M, x^{\mathsf{R}} \mapsto M'\}\} \uplus \mathcal{P}; \Phi; \mathcal{S}; i)$$
$$\text{if } c \in \mathcal{C}_{\mathsf{pub}} \text{ and } R\Phi_{\mathsf{fst}}\Downarrow =_{\mathsf{E}} M \text{ and } R\Phi_{\mathsf{snd}}\Downarrow =_{\mathsf{E}} M'$$

$$\textsc{IO}_b \quad (\{i : \mathsf{in}(c, \mathtt{diff}[x^{\mathsf{L}}, x^{\mathsf{R}}]); P\} \uplus \{i : \mathsf{out}(c, M); Q\} \uplus \mathcal{P}; \Phi; \mathcal{S}; i) \xrightarrow{\tau} (\{i : P\{x^{\mathsf{L}} \mapsto \mathsf{fst}(M), x^{\mathsf{R}} \mapsto \mathsf{snd}(M)\}\} \uplus \{i : Q\} \uplus \mathcal{P}; \Phi; \mathcal{S}; i) \text{ if } c \in \mathcal{C}_{\mathsf{priv}}$$

When executing a let instruction, the left and right processes have to progress in the same way:

$$(\{i : \mathtt{let}\ \mathtt{diff}[x^{\mathsf{L}}, x^{\mathsf{R}}] = D\ \mathtt{in}\ P\ \mathtt{else}\ Q\} \uplus \mathcal{P}; \Phi; \mathcal{S}; i) \xrightarrow{\tau}_b (\{i : P\{x^{\mathsf{L}} \mapsto \mathsf{fst}(D)\Downarrow, x^{\mathsf{R}} \mapsto \mathsf{snd}(D)\Downarrow\}\} \uplus \mathcal{P}; \Phi; \mathcal{S}; i)$$

$$(\{i : \mathtt{let}\ \mathtt{diff}[x^{\mathsf{L}}, x^{\mathsf{R}}] = D\ \mathtt{in}\ P\ \mathtt{else}\ Q\} \uplus \mathcal{P}; \Phi; \mathcal{S}; i) \xrightarrow{\tau}_b (\{i : Q\} \uplus \mathcal{P}; \Phi; \mathcal{S}; i) \text{ if } \mathsf{fst}(D)\not\Downarrow \text{ and } \mathsf{snd}(D)\not\Downarrow$$

When executing a get instruction, the left and right processes have to execute the same branch *and* access to the same element:

$$(\{i : \mathtt{get}\ tbl(\mathtt{diff}[x^{\mathsf{L}}, x^{\mathsf{R}}])\ \mathtt{st}.\ D\ \mathtt{in}\ P\ \mathtt{else}\ Q\} \uplus \mathcal{P}; \Phi; \mathcal{S}; i) \xrightarrow{\tau}_b (\{i : P\{x^{\mathsf{L}} \mapsto \mathsf{fst}(M), x^{\mathsf{R}} \mapsto \mathsf{snd}(M)\}\} \uplus \mathcal{P}; \Phi; \mathcal{S}; i)$$
$$\text{if there exists } tbl(M) \in \mathcal{S} \text{ such that } \mathsf{fst}(D)\{x^{\mathsf{L}} \mapsto \mathsf{fst}(M), x^{\mathsf{R}} \mapsto \mathsf{snd}(M)\}\Downarrow =_{\mathsf{E}} \mathsf{true} \text{ and } \mathsf{snd}(D)\{x^{\mathsf{L}} \mapsto \mathsf{fst}(M), x^{\mathsf{R}} \mapsto \mathsf{snd}(M)\}\Downarrow =_{\mathsf{E}} \mathsf{true}$$

$$(\{i : \mathtt{get}\ tbl(\mathtt{diff}[x^{\mathsf{L}}, x^{\mathsf{R}}])\ \mathtt{st}.\ D\ \mathtt{in}\ P\ \mathtt{else}\ Q\} \uplus \mathcal{P}; \Phi; \mathcal{S}; i) \xrightarrow{\tau}_b (\{i : Q\} \uplus \mathcal{P}; \Phi; \mathcal{S}; i)$$
$$\text{if for all } tbl(M) \in \mathcal{S},\ \mathsf{fst}(D)\{x^{\mathsf{L}} \mapsto \mathsf{fst}(M), x^{\mathsf{R}} \mapsto \mathsf{snd}(M)\} \text{ does not evaluate to } \mathsf{true} \text{ and } \mathsf{snd}(D)\{x^{\mathsf{L}} \mapsto \mathsf{fst}(M), x^{\mathsf{R}} \mapsto \mathsf{snd}(M)\} \text{ does not evaluate to } \mathsf{true}$$

An execution trace of a biprocess is called a *bi-execution trace*. The notion of convergence defined next captures the fact that the left and right processes always agree along a trace, in the sense that one of the above rules is always applicable.

Definition 4. *A bi-execution trace T* converges*, denoted $T{\downarrow}{\uparrow}$, when for all steps τ, $T[\tau] = K = (\mathcal{P}; \Phi; \mathcal{S}; i)$ implies:*

1. *if $(i : \mathtt{get}\ tbl(\mathtt{diff}[x^{\mathsf{L}}, x^{\mathsf{R}}])\ \mathtt{st}.\ D\ \mathtt{in}\ P\ \mathtt{else}\ Q) \in \mathcal{P}$ then for all $tbl(M) \in \mathcal{S}$, $\mathsf{fst}(D)\{x^{\mathsf{L}} \mapsto \mathsf{fst}(M), x^{\mathsf{R}} \mapsto \mathsf{snd}(M)\}\Downarrow =_{\mathsf{E}} \mathtt{true}$ if and only if $\mathsf{snd}(D)\{x^{\mathsf{L}} \mapsto \mathsf{fst}(M), x^{\mathsf{R}} \mapsto \mathsf{snd}(M)\}\Downarrow =_{\mathsf{E}} \mathtt{true}$;*
2. *$(i : \mathtt{let}\ \mathtt{diff}[x^{\mathsf{L}}, x^{\mathsf{R}}] = D\ \mathtt{in}\ P\ \mathtt{else}\ Q) \in \mathcal{P}$ then $\mathsf{fst}(D)$ evaluates if and only if $\mathsf{snd}(D)$ evaluates;*
3. *$\mathsf{fst}(\Phi) \sim_s \mathsf{snd}(\Phi)$.*

We extend this notion to sets of bitraces as expected, i.e. $\mathcal{T}\downarrow\uparrow$ if, and only if, $T\downarrow\uparrow$ for all $T \in \mathcal{T}$. In [7], Blanchet *et. al.* proved an important result for separated biprocesses: if $\mathtt{traces}(B)$ converges then $\mathsf{fst}(B)$ is observationally equivalent to $\mathsf{snd}(B)$. In short, and because observational equivalence implies trace equivalence [11], we have:

$$\text{if } B \text{ is separated and } \mathtt{traces}(B)\downarrow\uparrow \text{ then } \mathsf{fst}(B) \approx_t \mathsf{snd}(B)$$

Example 7. Going back to our running example, we form the biprocess B_{BH}:

$$(0 : !\,\mathtt{new}\ k^{\mathsf{L}};\ !\,\mathtt{new}\ k^{\mathsf{R}};\ \mathtt{insert}\ keys(\mathtt{diff}[k^{\mathsf{L}}, k^{\mathsf{R}}]); 1 : !\ B_{\mathsf{T}})\ \mid\ (1 : !\ B_{\mathsf{R}})$$

where $B_{\mathsf{T}} = P_{\mathsf{T}}\{k \mapsto \mathtt{diff}[k^{\mathsf{L}}, k^{\mathsf{R}}]\}$, and B_{R} is obtained from P_{R} by changing x (resp. y) into $\mathtt{diff}[x^{\mathsf{L}}, x^{\mathsf{R}}]$ (resp. $\mathtt{diff}[y^{\mathsf{L}}, y^{\mathsf{R}}]$). This is a separated biprocess. The projection $\mathsf{fst}(B_{\mathsf{BH}})$ represents the situation with multiple tags playing multiple sessions, whereas $\mathsf{snd}(B_{\mathsf{BH}})$ represents tags that play only once.

Unfortunately, the set $\mathtt{traces}(B_{\mathsf{BH}})$ of bi-execution traces is not convergent. Indeed, as explained in [2], we can reach a biconfiguration $(\mathcal{P}; \Phi; \mathcal{S}; i)$ such that $\mathcal{P}$ contains the biprocess

$$i : \mathtt{get}\ keys(\mathtt{diff}[y^{\mathsf{L}}, y^{\mathsf{R}}])\ \mathtt{st}.\ D\{x^{\mathsf{L}} \mapsto \mathsf{fst}(M), x^{\mathsf{R}} \mapsto \mathsf{snd}(M)\}\ \mathtt{in}\ \ldots\ \mathtt{else}\ \ldots$$

where $M = \langle n_T, \mathsf{h}(n_T, \mathtt{diff}[k^{\mathsf{L}}, k_1^{\mathsf{R}}])\rangle$, and the associated $\mathcal{S}$ contains the two elements $keys(\mathtt{diff}[k^{\mathsf{L}}, k_1^{\mathsf{R}}])$ and $keys(\mathtt{diff}[k^{\mathsf{L}}, k_2^{\mathsf{R}}])$. Considering the element $keys(\mathtt{diff}[k^{\mathsf{L}}, k_2^{\mathsf{R}}]) \in \mathcal{S}$, we can see that item 1 of Definition 4 is not satisfied.

3.2 Analysing biprocesses with restrictions, lemmas and axioms

An extension of ProVerif with restrictions, lemmas, and axioms has recently been introduced [8]. As we shall see, these features are useful to establish unlinkability.

We consider *formulas* and *correspondence queries* given by the grammar below, where M and N are (constructor) terms:

$$\psi, \psi' := \mathsf{true} \mid \mathsf{false} \mid \mathtt{event}(e) \mid M = N \mid M \neq N \mid \psi \wedge \psi' \mid \psi \vee \psi'$$
$$\rho := \mathtt{event}(e_1) \wedge \ldots \wedge \mathtt{event}(e_n) \Rightarrow \psi$$

Intuitively, a trace T satisfies the correspondence query ρ (noted $T \vdash \rho$) if whenever T contains instances of $\mathtt{event}(e_i)$ at τ_i for each $1 \leq i \leq n$, then T also satisfies ψ where each event in ψ is found in T at some step τ that occurs before some τ_i. A configuration K_0 satisfies ρ when $T \vdash \rho$ for any $T \in \mathtt{traces}(K_0)$.

This notion of correspondence queries is extended naturally to bi-execution traces by considering that events contain bi-terms.

Correspondence queries may be used in several ways in the verification of a biprocess. We call *restriction* a correspondence query that is going to be used to restrict the set of bi-execution traces that we want to consider. We call *axiom* a correspondence query that holds for all bi-execution traces, and we call it a *lemma* when this fact should be verified by ProVerif instead of just being assumed. The next theorem sums up what it means to verify a biprocess

in presence of axioms, lemmas and restrictions using ProVerif's semi-decision procedure prove' for equivalence queries — axioms and lemmas are similarly handled in this theorem, but in practice axioms would be trusted while lemmas would be verified. This result has been stated and proved in [8, Theorem 1] for separated biprocesses only but its authors are confident that the result holds for the general biprocesses considered here (personal communication).

Theorem 1 ([8]). *Let K be an (initial) biconfiguration, $\mathcal{Ax}$, $\mathcal{L}$, $\mathcal{R}$ be respectively sets of axioms, lemmas, and restrictions. If*

- *for all $\varrho \in \mathcal{Ax} \cup \mathcal{L}$, $\mathtt{traces}(K) \cap \{T \mid T \vdash \rho, \forall \rho \in \mathcal{R}\} \vdash \varrho$*
- *and $\mathsf{prove}'(K, \mathcal{Ax}, \mathcal{L}, \mathcal{R})$ returns* true

then $\mathtt{traces}(K) \cap \{T \mid T \vdash \rho, \forall \rho \in \mathcal{R}\}\downarrow\uparrow$.

4 Our approach

As we have seen in Example 7, ProVerif is not able to conclude on the biprocess modelling our unlinkability property on the Basic Hash protocol. We propose to transform such biprocesses so that diff-equivalence may be achieved (and verified with ProVerif) on the transformed bi-processes, and that this diff-equivalence implies trace equivalence for the original process. We proceed in two steps: (1) We duplicate the get instructions to dissociate the process of the left and the one on the right. (2) We add some axioms to help ProVerif to reason on our biprocess. These axioms are used to improve the accuracy of the verification procedure by avoiding over-approximations. Before formally defining these two steps in Sections 4.2 and 4.3, we illustrate the approach on our running example.

4.1 Going back to our running example

On our running example, our transformations will only change the biprocess modelling the reader since it is the only one that features a get instruction. Instead of performing a get instruction to access one (bi)record in the keys table, it will perform two get instructions in a row to access two records in the keys table. This will allow to choose two different records: one for the left and one for the right. Of course, the test performed remains the same and has to be satisfied by the element chosen on the left, and the one chosen on the right. We show the transformed biprocess in Figure 1, where the symbol _ represents irrelevant names for variables or public channels, and bad_L and bad_R are distinct public constants. The reader can ignore the added events at this stage.

This transformation alone is not enough to allow ProVerif to conclude. We thus add some axioms that emphasise some properties that are necessarily satisfied, to make sure that these properties are not lost in the over-approximations performed in ProVerif's verification procedure. To this end, we make use of the three events introduced in the transformed bi-process: Inserted, FailL, and FailR. The first one indicates when an element is inserted in the table *keys*, whereas the

$$
\begin{array}{l}
\mathtt{in}(c, \mathtt{diff}[x^{\mathsf{L}}, x^{\mathsf{R}}]);\\
\mathtt{get}\ \mathit{keys}(\mathtt{diff}[y^{\mathsf{L}}, \_])\ \mathtt{st.}\ \mathsf{eq}(\mathsf{proj}_2(x^{\mathsf{L}}), \mathsf{h}(\mathsf{proj}_1(x^{\mathsf{L}}, y^{\mathsf{L}})))\ \mathtt{in}\\
\quad \mathtt{get}\ \mathit{keys}(\mathtt{diff}[\_, y^{\mathsf{R}}])\ \mathtt{st.}\ \mathsf{eq}(\mathsf{proj}_2(x^{\mathsf{R}}), \mathsf{h}(\mathsf{proj}_1(x^{\mathsf{R}}, y^{\mathsf{R}})))\ \mathtt{in}\\
\quad\ \mathtt{out}(c, \mathsf{ok})\\
\quad \mathtt{else}\ \mathtt{event}(\mathsf{FailR}(x^{\mathsf{R}})); \mathsf{out}(\_, \mathtt{diff}[\mathsf{bad}_{\mathsf{L}}, \mathsf{bad}_{\mathsf{R}}])\\
\mathtt{else}\\
\quad \mathtt{event}(\mathsf{FailL}(x^{\mathsf{L}}));\\
\quad \mathtt{get}\ \mathit{keys}(\mathtt{diff}[\_, y^{\mathsf{R}}])\ \mathtt{st.}\ \mathsf{eq}(\mathsf{proj}_2(x^{\mathsf{R}}), \mathsf{h}(\mathsf{proj}_1(x^{\mathsf{R}}, y^{\mathsf{R}})))\ \mathtt{in}\\
\quad\ \mathsf{out}(\_, \mathtt{diff}[\mathsf{bad}_{\mathsf{L}}, \mathsf{bad}_{\mathsf{R}}])\\
\quad \mathtt{else}\ \mathtt{out}(c, \mathsf{error})
\end{array}
$$

Fig. 1. Transformed reader biprocess for Basic Hash

two others will be used to track the outcome of lookups in that table: FailL indicates a failure of the get instruction regarding the left hand side of the process, whereas FailR indicates a failure of the get instruction w.r.t. the right process. Relying on these events, we state two axioms:

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

Let us focus on the first one. In case the FailL event is executed with a value M, it means that there is no entry $\mathtt{diff}[N, N']$ in the table allowing one to evaluate the expression $\mathsf{eq}(\mathsf{proj}_2(M), \mathsf{h}(\mathsf{proj}_1(M), N))$ to true. Otherwise, we should have chosen this entry to execute the $\mathtt{then}$ branch. Thus, it is safe to say that for any entry $\mathtt{diff}[N, N']$ in the table key, we have $\mathsf{proj}_2(M) \neq \mathsf{h}(\mathsf{proj}_1(M), N)$. The same applies on the right.

We may note that such an axiom is indeed satisfied since we know that insertions in the table can be done at the very beginning of the execution traces. This means that the element allowing one to perform the reasoning above can be assumed to be present before the execution of the FailL or FailR event.

4.2 Desynchronising the two parts of the biprocess

We now define the first step of the transformation informally described above.

Definition 5. *Let B be a separated biprocess. We define $\mathcal{T}_1(B)$ as follows:*

- $\mathcal{T}_1(\mathtt{get}\ \mathit{tbl}(\mathtt{diff}[x^{\mathsf{L}}, x^{\mathsf{R}}])\ \mathtt{st.}\ D\ \mathtt{in}\ B_{\mathsf{then}}\ \mathtt{else}\ B_{\mathsf{else}})$ *is the following biprocess, where* $\_$ *represents irrelevant names for variables or public channels, and* $\mathsf{bad} = \mathtt{diff}[\mathsf{bad}_{\mathsf{L}}, \mathsf{bad}_{\mathsf{R}}]$ *for two arbitrary distinct public constants* $\mathsf{bad}_{\mathsf{L}}$ *and* $\mathsf{bad}_{\mathsf{R}}$*:*

$$
\begin{array}{l}
\mathtt{get}\ \mathit{tbl}(\mathtt{diff}[x^{\mathsf{L}}, \_])\ \mathtt{st.}\ \mathsf{fst}(D)\ \mathtt{in}\\
\quad \mathtt{get}\ \mathit{tbl}(\mathtt{diff}[\_, x^{\mathsf{R}}])\ \mathtt{st.}\ \mathsf{snd}(D)\ \mathtt{in}\ \mathcal{T}_1(B_{\mathsf{then}})\ \mathtt{else}\ \mathsf{out}(\_, \mathsf{bad})\\
\mathtt{else}\\
\quad \mathtt{get}\ \mathit{tbl}(\mathtt{diff}[\_, x^{\mathsf{R}}])\ \mathtt{st.}\ \mathsf{snd}(D)\ \mathtt{in}\ \mathsf{out}(\_, \mathsf{bad})\ \mathtt{else}\ \mathcal{T}_1(B_{\mathsf{else}})
\end{array}
$$

– *The transformation is homomorphic in all other cases: for instance, we define* $\mathcal{T}_1(\mathtt{out}(c, M); B') = \mathtt{out}(c, M); \mathcal{T}_1(B')$ *and* $\mathcal{T}_1(\mathtt{in}(c, \mathtt{diff}[x^{\mathsf{L}}, x^{\mathsf{R}}]); B') = \mathtt{in}(c, \mathtt{diff}[x^{\mathsf{L}}, x^{\mathsf{R}}]); \mathcal{T}_1(B')$.

Note that, because B is separated, $\mathsf{fst}(D)$ cannot have x^{R} as a free variable, which would render the above transformation semantically dubious. In the above definition, when $\mathsf{fst}(D)$ and $\mathsf{snd}(D)$ disagree, we use the biprocess $\mathtt{out}(\_, \mathtt{diff}[\mathsf{bad}_{\mathsf{L}}, \mathsf{bad}_{\mathsf{R}}])$ to signal that we want diff-equivalence to fail. To conclude in more cases, a variant could slightly relax this, under the assumption that B_{then} and B_{else} have the same structure. We do not encounter this situation in our case studies, and we left this extension as future work.

We establish the following result so that the "equivalence is true" returned by ProVerif on $\mathcal{T}_1(B_0)$ will allow one to conclude that trace equivalence holds between the two projections of B_0, and thus our unlinkability property holds.

Theorem 2. *Let* B_0 *be a separated biprocess such that* $\mathcal{T}_1(B_0)\downarrow\uparrow$. *We have* $\mathsf{fst}(B_0) \approx_t \mathsf{snd}(B_0)$.

This theorem will be proved relying on two lemmas that we now introduce. In these lemmas, when T is a bi-execution trace of $\mathcal{T}_1(B_0)$, we allow ourselves to write $\mathsf{fst}(T)$ meaning that the first projection has been applied on each configuration of T. Note that this does not imply that $\mathsf{fst}(T)$ is an execution of $\mathsf{fst}(\mathcal{T}_1(B_0))$ as $\mathcal{T}_1(B_0)$ is not separated. Actually, we only use $\mathsf{fst}(T)$ to assess its indistinguishability wrt. another trace (e.g., in $T_{\mathsf{fst}} \cong \mathsf{fst}(T)$): we thus only care about the observables, and the first projection of the last frame of T. Since $\mathcal{T}_1(B_0)$ duplicates the `get` instruction, to establish the connection between $\mathsf{fst}(B_0)$ and $\mathcal{T}_1(B_0)$, it is easier to consider a notion of execution trace where the two `get` instructions corresponding to the same `get` instruction in the original biprocess B_0 are triggered in a row. We denote $\overline{\mathtt{traces}}$ this notion of bi-execution trace.

Lemma 1. *Let* B_0 *be a separated biprocess such that* $\mathcal{T}_1(B_0)\downarrow\uparrow$, *and* $T_{\mathsf{fst}} \in \mathtt{traces}(\mathsf{fst}(B_0))$. *There exists* $T \in \overline{\mathtt{traces}}(\mathcal{T}_1(B_0))$ *such that* $T_{\mathsf{fst}} \cong \mathsf{fst}(T)$. *A similar result holds for* snd.

Proof (Sketch). We will find T such that $\mathsf{fst}(T)$ and T_{fst} differ only on the duplicated τ actions corresponding to transformed `get` operations. Although the lemma is stated in terms of static equivalence, frames are actually equal. The convergence hypothesis guarantees that we avoid the $\mathtt{diff}[\mathsf{bad}_{\mathsf{L}}, \mathsf{bad}_{\mathsf{R}}]$ which can artificially appear in the traces of the transformed biprocess. □

Lemma 2. *Let* B_0 *be a separated biprocess such that* $\mathcal{T}_1(B_0)\downarrow\uparrow$, *and let* $T \in \overline{\mathtt{traces}}(\mathcal{T}_1(B_0))$. *There exists* $T_{\mathsf{fst}} \in \mathtt{traces}(\mathsf{fst}(B_0))$ *such that* $T_{\mathsf{fst}} \cong \mathsf{fst}(T)$. *A similar result holds for* snd.

Proof (Sketch). The projected traces are simply obtained by de-duplicating the τ actions corresponding to `get` subprocesses. Again, the convergence hypothesis guarantees that we avoid the $\mathtt{diff}[\mathsf{bad}_{\mathsf{L}}, \mathsf{bad}_{\mathsf{R}}]$. □

4.3 Refining the analysis in failure branches

We transform a *model*, i.e. a bi-process $\mathcal{C}$ together with restrictions, axioms and lemmas, to a new model that is easier to verify by ProVerif.

Definition 6. *We have* $(B, \mathcal{R}, \mathcal{A}x, \mathcal{L}) \rightsquigarrow_{\mathsf{else}} (B', \mathcal{R}, \mathcal{A}x \cup \{ax\}, \mathcal{L})$ *when there exists a context* C *and two terms* M_1 *and* M_2 *without diff operators such that*

$B = C[\mathtt{get}\ tbl(\mathtt{diff}[x^{\mathsf{L}}, x^{\mathsf{R}}])\ \mathtt{st}.\ \mathsf{eq}(M_1, M_2)\ \mathtt{in}\ P\ \mathtt{else}\ Q]$

$B' = C[\mathtt{get}\ tbl(\mathtt{diff}[x^{\mathsf{L}}, x^{\mathsf{R}}])\ \mathtt{st}.\ \mathsf{eq}(M_1, M_2)\ \mathtt{in}\ P\ \mathtt{else}\ \mathtt{event}(\mathsf{Fail}(\vec{n}, \vec{y})).Q]$

where $\vec{n} = \mathsf{names}(\{M_1, M_2\}), \vec{y} = \mathsf{vars}(\{M_1, M_2\}) \setminus \{x^{\mathsf{L}}, x^{\mathsf{R}}\}$ *and all operations* $\mathtt{insert}\ tbl(M)$ *have been replaced by* $\mathtt{event}(\mathsf{Inserted}(M)).\mathtt{insert}\ tbl(M)$

$ax = \{\mathtt{event}(\mathsf{Fail}(\vec{z}, \vec{y})) \wedge \mathtt{event}(\mathsf{Inserted}(\mathtt{diff}[x^{\mathsf{L}}, x^{\mathsf{R}}])) \Rightarrow (M_1 \neq M_2)\{\vec{n} \mapsto \vec{z}\}\}$

where Fail *and* $\mathsf{Inserted}$ *are distinct events that do not occur in* $(B, \mathcal{R}, \mathcal{A}x, \mathcal{L})$*, and* $\vec{z}$ *is a vector of fresh variables of the same length as* $\vec{n}$*.*

Intuitively, the transformation instruments the process with events in order to express, in the added axiom, that we can only execute the else branch of the biprocess if no value inserted in *tbl* satisfies $M_1 = M_2$. This makes sense even if x^{L} and x^{R} do not occur in $\mathsf{eq}(M_1, M_2)$: in that case we are simply saying that we can only visit the else branch if either $\mathsf{eq}(M_1, M_2)$ is false, or no value has even been inserted in the table.

The above intuition is not entirely exact, though. Indeed, the semantics of the axiom is that if at some point in the trace the events Fail and $\mathsf{Inserted}$ have happened (for some values of the free variables $\vec{z}, \vec{y}, x^{\mathsf{L}}, x^{\mathsf{R}}$), then $\mathsf{eq}(M_1, M_2)$ evaluates to true. The semantics of biprocesses enforces a weaker version of this statement, where the $\mathsf{Inserted}$ must happen before Fail. It is not possible to express such precise temporal properties in restrictions with ProVerif but, as we shall see, we can live with this limitation.

The transformation can be iterated. Obviously, the insert events of several transformations relying on the same table can be merged. We justify next a single step of the transformation; the result extends naturally for iterations.

Proposition 1. *Let* $(B, \mathcal{R}, \mathcal{A}x, \mathcal{L})$*, and* $(B', \mathcal{R}, \mathcal{A}x', \mathcal{L})$ *be two models such that* $(B, \mathcal{R}, \mathcal{A}x, \mathcal{L}) \rightsquigarrow_{\mathsf{else}} (B', \mathcal{R}, \mathcal{A}x', \mathcal{L})$*. We have that* $\mathcal{A}x' = \mathcal{A}x \cup \{ax\}$*, and assuming that all insert instructions are performed in a phase that precedes the phase of the modified get instruction, we have that*

$$\mathtt{traces}(B') \cap \{T \mid T \vdash \rho, \forall \rho \in \mathcal{R}\} \vdash ax.$$

Proof. Consider a bitrace $T' \in \mathtt{traces}(B') \cap \{T \mid T \vdash \rho, \forall \rho \in \mathcal{R}\}$, and assume that T' violates ax. Then for some substitution θ, we have $\mathsf{Inserted}(\mathtt{diff}[x^{\mathsf{L}}, x^{\mathsf{R}}])\theta$ in the bitrace, as well as $\mathsf{Fail}(\vec{z}, \vec{y})\theta$, and $(M_1 =_{\mathsf{E}} M_2)\{\vec{n} \mapsto \vec{z}\}\theta$ is true. In other words, the execution of the $\mathtt{get}$ action, for the values given by θ, has stepped to the $\mathtt{else}$ branch, when there existed a biterm $\mathtt{diff}[x^{\mathsf{L}}, x^{\mathsf{R}}]\theta$ in the table that satisfies the condition. This contradicts the semantics of biprocesses. □

This result, together with Theorem 1, shows that if ProVerif concludes on the transformed model, then diff-equivalence holds for the original one.

5 Case studies

In this section, we explain how our approach can be used to analyse unlinkability on several existing protocols.

5.1 Implementation

In order to analyse our protocols of interest, we have developed a tool that implements the two transformations described in the previous section. This tool makes it easy to test our approach on various protocols, and avoids mistakes during transformations. It takes as input a ProVerif model containing a biprocess specifying the equivalence to be verified and returns another ProVerif file. If ProVerif concludes that diff-equivalence holds on the output file then our result allows one to conclude that the left and right processes written in the input file are in trace equivalence.

In practice, we implemented a ProVerif front-end tool to automatically apply our two transformations ($\approx 2k$ OCaml LoC). It takes as input a description of the protocol as presented in Example 7 and outputs a new ProVerif model in which the two parts of the biprocess have been desynchronised and the axioms corresponding to else branches have been added. ProVerif can then be run on this output to automatically prove the security of the protocol. The source code of our tool and material to reproduce results can be found at [17].

5.2 Results

We first study four examples: Basic Hash (used as a running example), Hash-Lock, Feldhofer, and a variant of the LAK protocol (LAK-v1). All these examples are described, e.g., in [2]. They are similar in the sense that they involve monotonic states, i.e. states that are never updated. On all these examples, it was already well-known that ProVerif is unable to conclude on the model encoding unlinkability relying on standard biprocess and diff-equivalence. Applying our tool, we obtain a file on which ProVerif is able to conclude in a few seconds, thus we conclude that these protocols ensure unlinkability.

We then consider a fixed version of the OSK protocol, described as OSK-v2 in [2]. The protocol can be described informally as follows:

$$T \to R: \quad \mathsf{g}(\mathsf{h}(k_T))$$
$$T \text{ updates } k_T \text{ with } \mathsf{h}(k_T)$$

Here, g and h are two hash functions. Each tag has a secret key k_T, whose initial value is stored in the reader's database. The tag updates its key by applying the h function symbol at each session on its current key. The reader expects a message of the form $\mathsf{g}(\mathsf{h}^n(x))$ for some database entry x (h^n means that h is applied n times). This check is modelled relying on a private destructor $\mathsf{retrievekey}$ allowing one to extract k from a term of the form $\mathsf{h}(n, k)$ used to model $\mathsf{h}^n(k)$. As usual in ProVerif, we encode the state of the tag relying on a private channel. On this model ProVerif is unable to conclude, as expected.

Strictly speaking, our first transformation does not apply on the OSK-v2 reader biprocess, as Definition 6 only handles tests of the form $\mathsf{eq}(M_1, M_2)$ where M_1 and M_2 do not contain destructors, whereas our reader features a test $\mathsf{eq}(\mathsf{retrievekey}(x), k)$. This hypothesis is actually important to be able to write the corresponding axiom respecting the ProVerif syntax. However, we can express a similar condition performing a more involved transformation which essentially consists in computing the variants of an expression to get rid of destructor symbols [14]. On this specific example, we can write instead the following axiom:

$$\mathtt{event}(\mathsf{FailL}(\mathsf{g}(\mathsf{h}(z, x^{\mathsf{L}})))) \wedge \mathtt{event}(\mathsf{Inserted}(\mathtt{diff}[x^{\mathsf{L}}, x^{\mathsf{R}}])) \Longrightarrow \mathsf{false}.$$

Intuitively, we want to express that whenever a trace contains $\mathsf{FailL}(M)$ (i.e. the first get operation has failed) we have $\mathsf{retrievekey}(M) \neq \mathsf{fst}(K)$ for all $keys(K)$ in the store. Actually, this can happen because $\mathsf{retrievekey}(M)$ does not reduce, or because it reduces but not to $\mathsf{fst}(K)$. The axiom above only considers the second case, which is obviously sound and sufficient for our purpose.

One last issue remains after this manual application of our second transformation, that is orthogonal to the problem we consider in this paper: ProVerif is not precise enough when considering private channels. This is an issue that has been identified and solved in [12] for trace-based properties. Applying similar ideas, we come up with an axiom that solves the remaining issue and allows one to successfully conclude on this example. Justifying this axiom for OSK-v2 is not difficult, but a general treatment of this further transformation is beyond the scope of this paper. We expect that this final step will be superseded by a future extension of GSVerif to equivalence properties.

5.3 Comparison with earlier work

All the results presented above are in line with the security analysis performed in [2] where unlinkability has been established relying on three sufficient conditions proved using the tool Tamarin. Clearly, the approach proposed in [2] requires more coding effort as it is less direct than the one we described here. In particular, to analyse the three conditions, two encodings have to be written for each protocol. One to encode their *well-authentication* and *non-desynchronisation* conditions, and another one to encode their equivalence property, namely *frame opacity*. Each time, this requires to write several lemmas corresponding to their conditions (between two and four depending on the protocol). In some cases, several intermediate lemmas are needed to help Tamarin prove the conditions.

Regarding OSK-v2, the security analysis conducted in [2] has been possible thanks to several simplifications and using some modelling tricks. For instance, frame opacity has been established discarding the reader role, and the attacker is not given the power to apply h on top of a term. Again, the coding effort has been relatively important to conduct the security analysis. Thus, even if this example is not fully handled by our theoretical results and implementation, we consider that our approach brings a significant improvement in this case as well.

6 Conclusion

We have shown that unlinkability can be automatically proved in ProVerif, by applying two generic transformations on the standard bi-process expressing unlinkability. Our approach significantly improves over existing automated proofs, avoiding several over-approximations as well as manual work. It builds on several recent features of ProVerif. In particular, we make crucial use of extended bi-processes with diff operators on bound variables, that have seen only limited use so far (and which would not make sense in e.g. Tamarin).

As illustrated by our study of the OSK-v2 protocol, it would be natural to extend our theoretical development as well as our tool to extend our second transformation beyond equality tests. In light of the usefulness and simplicity of this transformation, a promising perspective would be to integrate it by default in ProVerif, but this requires the addition of temporal constraints in correspondence queries. Meanwhile, this work introduces a new approach that can be manually applied and/or adapted to improve ProVerif accuracy on many examples.

Our contribution in this paper, namely relaxing diff-equivalence to be able to prove unlinkability in ProVerif, does not address other difficult aspects that might arise in this task. Handling the tag's mutable state in the OSK-v2 case study has proved difficult, but there is hope that advances with GSVerif will soon solve this issue. It is also natural to consider protocols with state updates on the reader's side, i.e. updates of database entries. Although such operations are not directly supported in ProVerif, there is hope to model them naturally enough using restrictions – which calls for a (slight) generalisation of our results.

References

1. M. Arapinis, T. Chothia, E. Ritter, and M. Ryan. Analysing unlinkability and anonymity using the applied pi calculus. In *Proc. 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010*, pages 107–121. IEEE Computer Society, 2010.
2. D. Baelde, S. Delaune, and S. Moreau. A method for proving unlinkability of stateful protocols. In *Proc. of the 33rd IEEE Computer Security Foundations Symposium (CSF'20)*. IEEE Computer Society Press, July 2020.
3. D. Basin, R. Sasse, and J. Toro-Pozo. The EMV standard: Break, fix, verify. In *2021 IEEE Symposium on Security and Privacy (S&P'21)*, pages 1766–1781. IEEE, 2021.
4. D. A. Basin, J. Dreier, and R. Sasse. Automated symbolic proofs of observational equivalence. In I. Ray, N. Li, and C. Kruegel, editors, *Proc. 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*, pages 1144–1155. ACM, 2015.
5. K. Bhargavan, B. Blanchet, and N. Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *2017 IEEE Symposium on Security and Privacy, (S&P'17), San Jose, CA, USA, May 22-26, 2017*, pages 483–502. IEEE Computer Society, 2017.
6. B. Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *14th IEEE Computer Security Foundations Workshop (CSFW'01), 11-13 June 2001, Cape Breton, Nova Scotia, Canada*, pages 82–96, 2001.

7. B. Blanchet, M. Abadi, and C. Fournet. Automated verification of selected equivalences for security protocols. *Journal of Logic and Algebraic Programming*, 75(1):3–51, 2008.
8. B. Blanchet, V. Cheval, and V. Cortier. Proverif with lemmas, induction, fast subsumption, and much more. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (S&P'22)*. IEEE Computer Society Press, 2022.
9. M. Brusò, K. Chatzikokolakis, and J. den Hartog. Formal verification of privacy for RFID systems. In *Proc. 23rd IEEE Computer Security Foundations Symposium, (CSF'10)*, pages 75–88. IEEE Computer Society, 2010.
10. M. Brusò, K. Chatzikokolakis, S. Etalle, and J. den Hartog. Linking unlinkability. In C. Palamidessi and M. D. Ryan, editors, *Trustworthy Global Computing - 7th International Symposium, TGC 2012, Newcastle upon Tyne, UK, September 7-8, 2012, Revised Selected Papers*, volume 8191 of *LNCS*, pages 129–144. Springer, 2012.
11. V. Cheval, V. Cortier, and S. Delaune. Deciding equivalence-based properties using constraint solving. *Theor. Comput. Sci.*, 492:1–39, 2013.
12. V. Cheval, V. Cortier, and M. Turuani. A little more conversation, a little less action, a lot more satisfaction: Global states in proverif. In *Proceedings of the 31st IEEE Computer Security Foundations Symposium (CSF'18)*, pages 344–358, 2018.
13. T. Chothia and V. Smirnov. A traceability attack against e-passports. In R. Sion, editor, *Financial Cryptography and Data Security, 14th International Conference, FC 2010, Tenerife, Canary Islands, Spain, January 25-28, 2010, Revised Selected Papers*, volume 6052 of *LNCS*, pages 20–34. Springer, 2010.
14. H. Comon-Lundh and S. Delaune. The finite variant property: How to get rid of some algebraic properties. In *Proceedings of the 16th International Conference on Rewriting Techniques and Applications (RTA'05)*, volume 3467 of *LNCS*, pages 294–307, Nara, Japan, Apr. 2005. Springer.
15. V. Cortier, A. Dallon, and S. Delaune. Efficiently deciding equivalence for standard primitives and phases. In J. López, J. Zhou, and M. Soriano, editors, *Proc. 23rd European Symposium on Research in Computer Security, (ESORICS'18)*, volume 11098 of *LNCS*, pages 491–511. Springer, 2018.
16. C. Cremers, M. Horvat, J. Hoyland, S. Scott, and T. van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, editors, *Proc. ACM SIGSAC Conference on Computer and Communications Security, (CCS'17)*, pages 1773–1788. ACM, 2017.
17. A. Debant, S. Delaune, and D. Baelde. Proving Unlinkability using ProVerif through Desynchronized Bi-Processes. working paper or preprint available at https://hal.inria.fr/hal-03674979, May 2022.
18. J. Dreier, L. Hirschi, S. Radomirovic, and R. Sasse. Automated unbounded verification of stateful cryptographic protocols with exclusive OR. In *Proc. 31st IEEE Computer Security Foundations Symposium (CSF'18)*, pages 359–373. IEEE Computer Society, 2018.
19. L. Hirschi, D. Baelde, and S. Delaune. A method for unbounded verification of privacy-type properties. *Journal of Computer Security*, 27(3):277–342, 2019.
20. S. Meier, B. Schmidt, C. Cremers, and D. A. Basin. The TAMARIN prover for the symbolic analysis of security protocols. In *Proc. 25th International Conference on Computer Aided Verification (CAV'13)*, pages 696–701, 2013.
21. T. van Deursen, S. Mauw, and S. Radomirovic. Untraceability of RFID protocols. In *Information Security Theory and Practices. Smart Devices, Convergence and Next Generation Networks, Second IFIP WG 11.2 International Workshop, WISTP*

2008, Seville, Spain, May 13-16, 2008. Proceedings, volume 5019 of *LNCS*, pages 1–15. Springer, 2008.

A Proofs

We define $\mathcal{T}_1(\mathcal{P})$ for a multiset $\mathcal{P}$ as the multiset of all transformed elements of $\mathcal{P}$. We extend this notion to bi-configurations as follows: $\mathcal{T}_1(K) = (\mathcal{T}_1(\mathcal{P}); \Phi; \mathcal{S}; i)$ when $K = (\mathcal{P}; \Phi; \mathcal{S}; i)$. Similarly, we extend the notion fst and snd to multiset of processes and to configurations as follows: $\mathsf{fst}(K) = (\mathsf{fst}(\mathcal{P}); \Phi_{\mathsf{fst}}; \mathcal{S}_{\mathsf{fst}}; i)$ when $K = (\mathcal{P}; \Phi; \mathcal{S}; i)$, and similarly for snd.

Lemma 1. *Let B_0 be a separated biprocess such that $\mathcal{T}_1(B_0)\downarrow\uparrow$, and $T_{\mathsf{fst}} \in \mathtt{traces}(\mathsf{fst}(B_0))$. There exists $T \in \overline{\mathtt{traces}}(\mathcal{T}_1(B_0))$ such that $T_{\mathsf{fst}} \cong \mathsf{fst}(T)$. A similar result holds for* snd.

Proof. We prove the result for fst, the case of snd being symmetric. We start by establishing the following claim.

Claim. Let K_0 be a separated bi-configuration, and $K = \mathcal{T}_1(K_0)$ such that $K\downarrow\uparrow$. If $\mathsf{fst}(K_0) \xrightarrow{\alpha} K'_L$ for some α and some K'_L, then there exist a separated bi-configuration K'_0 and K' such that $K'\downarrow\uparrow$, $K' = \mathcal{T}_1(K'_0)$, and $\mathsf{fst}(K'_0) = K'_L$. Moreover, either this step corresponds to the execution of a $\mathtt{get}$ instruction, and we have that $\alpha = \tau$, and $K \xrightarrow{\tau\tau} K'$ (with the execution of two $\mathtt{get}$ instructions in a row); otherwise we have that $K \xrightarrow{\alpha} K'$.

Note that the result stated above allows us to prove the statement in the lemma by induction on the length of the execution trace T_{fst}. Now, it remains to establish this claim. We do a case analysis depending on the rule used to perform $\mathsf{fst}(K_0) \xrightarrow{\alpha} K'_L$.

Case In. In such a case, we have that:

- $K_0 = (\{i : \mathtt{in}(c, \mathtt{diff}[x^{\mathsf{L}}, x^{\mathsf{R}}]); P\} \cup \mathcal{P}; \Phi; \mathcal{S}; i)$;
- $\alpha = \mathtt{in}(c, R)$ for some recipe R such that $R\Phi_{\mathsf{fst}}\Downarrow =_{\mathsf{E}} M$, and $R\Phi_{\mathsf{snd}}\Downarrow =_{\mathsf{E}} M'$;
- $K'_L = (\{i : \mathsf{fst}(P)\{x^{\mathsf{L}} \mapsto M\}\} \cup \mathsf{fst}(\mathcal{P}); \Phi_{\mathsf{fst}}; \mathcal{S}_{\mathsf{fst}}; i)$; and
- $K = \mathcal{T}_1(K_0) = (\{i : \mathtt{in}(c, \mathtt{diff}[x^{\mathsf{L}}, x^{\mathsf{R}}]); \mathcal{T}_1(P)\} \cup \mathcal{T}_1(\mathcal{P}); \Phi; \mathcal{S}; i)$.

We consider the two following bi-configurations:

- $K'_0 = (\{i : P\{x^{\mathsf{L}} \mapsto M, x^{\mathsf{R}} \mapsto M'\}\} \cup \mathcal{P}; \Phi; \mathcal{S}; i)$; and
- $K' = (\{i : \mathcal{T}_1(P)\{x^{\mathsf{L}} \mapsto M, x^{\mathsf{R}} \mapsto M'\}\} \cup \mathcal{T}_1(\mathcal{P}); \Phi; \mathcal{S}; i)$.

Since K_0 is separated, we have that K'_0 is separated too, and we have that $\mathsf{fst}(K'_0) = K'_L$. Relying on the fact that our transformation is compatible with the substitution of diff-free terms, i.e., $\mathcal{T}(B\sigma) = \mathcal{T}(B)\sigma$ for all biprocess B and substitution σ which do not introduce terms with diff operators, we deduce that $\mathcal{T}_1(K'_0) = K'$. Then, it is easy to see that $K \xrightarrow{\mathtt{in}(c,R)} K'$, and thus $K'\downarrow\uparrow$.

Most of the cases can be done in a similar way. We focus now on the more interesting cases, i.e. the ones corresponding to $\mathtt{get}$ instructions.

Case Get-T. In such a case, we have that $\alpha = \tau$, and:

– $K_0 = (\{i : \mathtt{get}\ tbl(\mathtt{diff}[x^{\mathsf{L}}, x^{\mathsf{R}}])\ \mathtt{st}.\ D\ \mathtt{in}\ B_{\mathsf{then}}\ \mathtt{else}\ B_{\mathsf{else}}\} \cup \mathcal{P}; \Phi; \mathcal{S}; i)$;
– $K'_L = (\{i : \mathsf{fst}(B_{\mathsf{then}})\{x^{\mathsf{L}} \mapsto M_{\mathsf{L}}\}\} \cup \mathsf{fst}(\mathcal{P}); \Phi_{\mathsf{fst}}; \mathcal{S}_{\mathsf{fst}}; i)$ where M_{L} is such that $tbl(M_{\mathsf{L}}) \in \mathcal{S}_{\mathsf{fst}}$, and $\mathsf{fst}(D)\{x^{\mathsf{L}} \mapsto M_{\mathsf{L}}\}\Downarrow =_{\mathsf{E}} \mathsf{true}$; and
– $K = \mathcal{T}_1(K_0) = (\{i : B'\} \cup \mathcal{T}_1(\mathcal{P}); \Phi; \mathcal{S}; i)$ where:

$$B' = \begin{cases} \mathtt{get}\ tbl(\mathtt{diff}[x^{\mathsf{L}}, \_])\ \mathtt{st}.\ \mathsf{fst}(D)\ \mathtt{in} \\ \quad \mathtt{get}\ tbl(\mathtt{diff}[\_, x^{\mathsf{R}}])\ \mathtt{st}.\ \mathsf{snd}(D)\ \mathtt{in} \\ \quad\ \mathcal{T}_1(B_{\mathsf{then}}) \\ \quad \mathtt{else}\ \mathsf{out}(\_, \mathtt{diff}[\mathsf{bad}_{\mathsf{L}}, \mathsf{bad}_{\mathsf{R}}]) \\ \mathtt{else} \\ \quad \mathtt{get}\ tbl(\mathtt{diff}[\_, x^{\mathsf{R}}])\ \mathtt{st}.\ \mathsf{snd}(D)\ \mathtt{in} \\ \quad\ \mathsf{out}(\_, \mathtt{diff}[\mathsf{bad}_{\mathsf{L}}, \mathsf{bad}_{\mathsf{R}}]) \\ \quad \mathtt{else}\ \mathcal{T}_1(B_{\mathsf{else}}) \end{cases}$$

Let M_{R} be such that $tbl(\mathtt{diff}[M_{\mathsf{L}}, M_{\mathsf{R}}]) \in \mathcal{S}$, and $\mathtt{diff}[M'_{\mathsf{L}}, M'_{\mathsf{R}}]$ be such that $tbl(\mathtt{diff}[M'_{\mathsf{L}}, M'_{\mathsf{R}}]) \in \mathcal{S}$ with $\mathsf{snd}(D)\{x^{\mathsf{R}} \mapsto M'_{\mathsf{R}}\}\Downarrow =_{\mathsf{E}} \mathsf{true}$. Note that the first element exists since we know that $tbl(M_{\mathsf{L}}) \in \mathcal{S}_{\mathsf{fst}}$. Regarding the second one, we know that it exists relying on our hypothesis $K{\downarrow}{\uparrow}$. Therefore, we consider the two following bi-configurations:

– $K'_0 = (\{i : B_{\mathsf{then}}\{x^{\mathsf{L}} \mapsto M_{\mathsf{L}}, x^{\mathsf{R}} \mapsto M'_{\mathsf{R}}\}\} \cup \mathcal{P}; \Phi; \mathcal{S}; i)$; and
– $K' = (\{i : \mathcal{T}_1(B_{\mathsf{then}})\{x^{\mathsf{L}} \mapsto M_{\mathsf{L}}, x^{\mathsf{R}} \mapsto M'_{\mathsf{R}}\}\} \cup \mathcal{T}_1(\mathcal{P}); \Phi; \mathcal{S}; i)$.

Since K_0 is separated, we have that K'_0 is separated too, and we have that $\mathsf{fst}(K'_0) = K'_L$. Relying on the fact that our transformation is compatible with substitution, we deduce that $\mathcal{T}_1(K'_0) = K'$. Then, it is easy to see that $K \xrightarrow{\tau\tau} K'$, and thus $K'{\downarrow}{\uparrow}$.

The case of the rule Get-E can be done in a similar way. This allows us to conclude. □

Lemma 2. *Let B_0 be a separated biprocess such that $\mathcal{T}_1(B_0){\downarrow}{\uparrow}$, and let $T \in \overline{\mathtt{traces}}(\mathcal{T}_1(B_0))$. There exists $T_{\mathsf{fst}} \in \mathtt{traces}(\mathsf{fst}(B_0))$ such that $T_{\mathsf{fst}} \cong \mathsf{fst}(T)$. A similar result holds for* snd.

Proof. We prove the result for fst, starting with the following claim.

Claim. Let K_0 be a separated bi-configuration, $K = \mathcal{T}_1(K_0)$ such that $K{\downarrow}{\uparrow}$. If $K \xrightarrow{\alpha} K'$ for some α and some K' with a rule different from Get-T and Get-E, then there exists K'_0 a separated bi-configuration such that $K' = \mathcal{T}_1(K'_0)$, and $\mathsf{fst}(K_0) \xrightarrow{\alpha} \mathsf{fst}(K'_0)$. A similar result holds when considering $K \xrightarrow{\tau\tau} K'$ corresponding to the execution of two get instructions in a row. In such a case, we have that $\mathsf{fst}(K_0) \xrightarrow{\tau} \mathsf{fst}(K'_0)$.

Note that the result stated above allows us to prove the statement in the lemma by induction on the length of the execution trace T. Now, it remains to establish this claim. We do a case analysis on the rule(s) used to perform $K \to K'$.

Case In. In such a case, we have that:

– $K_0 = (\{i : \mathtt{in}(c, \mathtt{diff}[x^{\mathsf{L}}, x^{\mathsf{R}}]); P\} \cup \mathcal{P}; \Phi; \mathcal{S}; i)$;
– $K = (\{i : \mathtt{in}(c, \mathtt{diff}[x^{\mathsf{L}}, x^{\mathsf{R}}]); \mathcal{T}_1(P)\} \cup \mathcal{T}_1(\mathcal{P}); \Phi; \mathcal{S}; i)$;
– $\alpha = \mathtt{in}(c, R)$ for some recipe R such that $R\Phi_{\mathsf{fst}}\Downarrow =_{\mathsf{E}} M$, and $R\Phi_{\mathsf{snd}}\Downarrow =_{\mathsf{E}} M'$;
– $K' = (\{i : \mathcal{T}_1(P)\{x^{\mathsf{L}} \mapsto M, x^{\mathsf{R}} \mapsto M'\}\} \cup \mathcal{T}_1(\mathcal{P}); \Phi; \mathcal{S}; i)$.

We consider the following bi-configuration:

$$K'_0 = (\{i : P\{x^{\mathsf{L}} \mapsto M, x^{\mathsf{R}} \mapsto M'\}\} \cup \mathcal{P}; \Phi; \mathcal{S}; i)$$

Since K_0 is separated, we have that K'_0 is separated too. Since $K \xrightarrow{\alpha} K'$ and $K{\downarrow}{\uparrow}$, we have that $K'{\downarrow}{\uparrow}$. Relying on the fact that our transformation is compatible with substitution, we deduce that $K' = \mathcal{T}_1(K'_0)$. Then, it is easy to see that $\mathsf{fst}(K_0) \xrightarrow{\alpha} \mathsf{fst}(K'_0)$.

Most of the cases can be done in a similar way. We focus now on the most interesting ones, corresponding to the execution of two `get` instructions in a row.

Case GET-T *followed by* GET-T. In such a case, we have that:

– $K_0 = (\{i : \mathtt{get}\ tbl(\mathtt{diff}[x^{\mathsf{L}}, x^{\mathsf{R}}])\ \mathtt{st}.\ D\ \mathtt{in}\ B_{\mathsf{then}}\ \mathtt{else}\ B_{\mathsf{else}}\} \cup \mathcal{P}; \Phi; \mathcal{S}; i)$;
– $K = \mathcal{T}_1(K_0) = (\{i : B'\} \cup \mathcal{T}_1(\mathcal{P}); \Phi; \mathcal{S}; i)$ where:

$$B' = \begin{cases} \mathtt{get}\ tbl(\mathtt{diff}[x^{\mathsf{L}}, \_])\ \mathtt{st}.\ \mathsf{fst}(D)\ \mathtt{in} \\ \quad \mathtt{get}\ tbl(\mathtt{diff}[\_, x^{\mathsf{R}}])\ \mathtt{st}.\ \mathsf{snd}(D)\ \mathtt{in} \\ \quad\ \mathcal{T}_1(B_{\mathsf{then}}) \\ \quad \mathtt{else}\ \mathsf{out}(\_, \mathtt{diff}[\mathsf{bad}_{\mathsf{L}}, \mathsf{bad}_{\mathsf{R}}]) \\ \mathtt{else} \\ \quad \mathtt{get}\ tbl(\mathtt{diff}[\_, x^{\mathsf{R}}])\ \mathtt{st}.\ \mathsf{snd}(D)\ \mathtt{in} \\ \quad\ \mathsf{out}(\_, \mathtt{diff}[\mathsf{bad}_{\mathsf{L}}, \mathsf{bad}_{\mathsf{R}}]) \\ \quad \mathtt{else}\ \mathcal{T}_1(B_{\mathsf{else}}) \end{cases}$$

– $K' = (\{i : \mathcal{T}_1(B_{\mathsf{then}})\{x^{\mathsf{L}} \mapsto M_{\mathsf{L}}, x^{\mathsf{R}} \mapsto M_{\mathsf{R}}\}\} \cup \mathcal{T}_1(\mathcal{P}); \Phi; \mathcal{S}; i)$.

We consider the following bi-configuration:

$K'_0 = (\{i : B_{\mathsf{then}}\{x^{\mathsf{L}} \mapsto M_{\mathsf{L}}, x^{\mathsf{R}} \mapsto M_{\mathsf{R}}\}\} \cup \mathcal{P}; \Phi; \mathcal{S}; i)$. Since K_0 is separated, we have that K'_0 is separated too. Since $K \xrightarrow{\tau\tau} K'$, and $K{\downarrow}{\uparrow}$, we have that $K'{\downarrow}{\uparrow}$. Relying on the fact that our transformation is compatible with substitution, we deduce that $K' = \mathcal{T}_1(K'_0)$. Then, it is easy to see that $\mathsf{fst}(K_0) \xrightarrow{\tau} \mathsf{fst}(K'_0)$.

Case GET-T *followed by* GET-E. In such a case, we have that

$$K' = (\{i : \mathtt{out}(c, \mathtt{diff}[\mathsf{bad}_{\mathsf{L}}, \mathsf{bad}_{\mathsf{R}}])\} \cup \mathcal{P}; \Phi; \mathcal{S}; i).$$

It is easy to see that K' diverges, and thus K diverges too, contradicting our hypothesis.

The other cases GET-E followed by GET-E, and GET-E followed by GET-T can be done in a similar way. This allows us to conclude. □

Theorem 2. *Let B_0 be a separated biprocess such that $\mathcal{T}_1(B_0){\downarrow}{\uparrow}$. We have $\mathsf{fst}(B_0) \approx_t \mathsf{snd}(B_0)$.*

Proof. Let $T_{\mathsf{fst}} \in \mathtt{traces}(\mathsf{fst}(B_0))$. By Lemma 1, we know that there exists $T \in \overline{\mathtt{traces}}(\mathcal{T}_1(B_0))$ such $T_{\mathsf{fst}} \cong \mathsf{fst}(T)$. Then, relying on Lemma 2 (considering the case snd), we know that there exists $T_{\mathsf{snd}} \in \mathtt{traces}(\mathsf{snd}(B_0))$ such that $T_{\mathsf{snd}} \cong \mathsf{snd}(T)$. Moreover, since $T \in \overline{\mathtt{traces}}(\mathcal{T}_1(B_0))$ and $\mathcal{T}_1(B_0)\downarrow\uparrow$, we have that $\mathsf{fst}(T) \cong \mathsf{snd}(T)$. This allows us to conclude that $T_{\mathsf{fst}} \cong \mathsf{fst}(T) \cong \mathsf{snd}(T) \cong T_{\mathsf{snd}}$. Hence, the result. □