Reconstruction of attacks in Proverif

Jules Timmerman Vincent Cheval

2025

Contents

Introduction

Machines all over the world use protocols to communicate with each other. All key systems, from bank payments to messaging, rely on such protocols, making security a key aspect. While manual proofs are possible, automated and computer-assisted proofs allow for more flexibility and ease the process of proving big protocols. Various tools already exist with different characteristics: the amount of automation (ranging from highly automated like CryptoVerif to less automated like Squirrel), or with varying theories supporting them. One such tool is Proverif. It relies on the *symbolic model*. Here, we approximate security proofs by considering that if two names are syntactically different, they are different in the semantics. For example, if a user randomly samples a key k and another key k', we consider that $k \neq k'$, even though in reality there is a small chance for these keys to be the same.

As security of an arbitrary protocol is undecidable, approximations are necessary. Proverif first transforms the protocol from pi calculus to Horn clauses, so it can be used in the logic. The clauses are then saturated through a resolution-based algorithm. These Horn clauses can then be used any number of times to create a derivation of the query, given by the user. The soundness of Proverif guarantees that if no derivations are found, then there is no trace in the protocol satisfying the query. However, Proverif might sometimes find a derivation, but it might be a "false attack" due to the approximations. While an exhaustive exploration of the traces is possible, it is too time-consuming.

Therefore, we use the derivation and other information from the saturation to search the trace space. By checking the derivation tree for key facts representing messages and other parts of the protocol, we slowly step through the protocol by only taking facts that appear in the derivation tree.

Related Works Proverif already implemented such an algorithm previously [1]. However, it had one big limitation: private channel synchronization. It was not able to find attacks when a protocol made extensive use of private channels. Nowadays, private channels are used in every big protocols to model a bunch of constructs, like memory cells. Thus, dealing with them is becoming increasingly important.

Contribution We propose an improved version of the previous algorithm that handles private channel synchronization. We also update the theory behind the algorithm to match the improvements that have been made to Proverif over the last twenty years and deal with new constructs. Lastly, we introduce optimizations that may be used during the saturation as well as on derivation to simplify occurrences.

Outline First, we present the various components of Proverif that leads to reconstruction. Second, we introduce the general structure of the algorithm and explain each parts and how they relate to the previous iteration of the algorithm as well as an example scenario where the newer version finds an attack. Lastly, we state and prove two results that may be used during the saturation to simplify occurrences.

1 Background and notations

This section acts as a background section where *some* concepts relevant for later are introduced. This is mostly a reference for notations more than an actual introduction and explanation of the concepts.

1.1 Traces and process semantics

Definition 1 (Configuration)**.** *A configuration $\kappa, \mathcal{P}, \mathcal{T}, \mathcal{A}$ is defined by κ a phase, $\mathcal{P}$ a multiset of closed processes, $\mathcal{T}$ a set of terms of the form $tbl(M_1 \dots M_n)$, $\mathcal{A}$ a set of terms of the form $D \rhd U$ representing the attacker's knowledge. Intuitively, D is the term that was used to deduce U.*

It is used to represent the current state of a process.

Note that compared to [2], we are omitting synchronization.

Now that we have configurations to represent protocols, we need to define a semantics, so we can step through our protocol execution. The simplest semantics can be found in Figure 3 (page 10) of [2]. We will mostly use the instrumented semantics, which carries additional information. To that end, we introduce *instrumented configurations*

Definition 2 (Instrumented configuration)**.** *An instrumented configuration $\kappa, \mathcal{P}, \mathcal{T}, \mathcal{A}, \Lambda$ is the same as a configuration with the following differences:*

- *Λ is a set of used session identifiers*
- *$\mathcal{P}$ a multiset of* instrumented processes.
- *$\mathcal{T}$ is now a set of pairs of the form (O, tbl) where O is an occurrence term and tbl a table entry. Intuitively, it is the occurrence that created this entry.*

An instrumented process is a process where occurrences and the previous replications / messages were added as information. The transformation from process to instrumented process can be found in Figure 7 (page 30) of [2]. It explains how occurrences are generated.

Now for the semantics.

Definition 3 (Instrumented Semantics)**.** *We create the step relation $\xrightarrow{l}_i$ between configurations, where l is either empty, or one or multiple labels. Many different labels are possible. Among them are:*

- *$pre(O, M)$ for inputs and assignments to record how the variable was instantiated.*
- *$\texttt{event}(ev, O)$ represents the event ev at occurrence O.*
- *$io(M, N)$ to indicate that a message N was sent on channel M.*

The complete semantics can be found in Figure 8 (page 33) of [2]

1.2 Clause generation

The generation of clauses is explained in Figure 10 (page 42) of [2]. We write the generation of clause $[\![P]\!]\kappa\mathcal{H}$ where we use the same notations as the instrumented semantics and $\mathcal{H}$ is a set of facts and formulas used as premises.

Derivation tree After doing a saturation, we may find a faulty clause. This can be used to build a derivation tree. The nodes are labeled with Horn clauses. The edges are labeled with ground facts. A node has one incoming edge and n outgoing edges and represents an instantiation of a clause. The root as one incoming edge labeled with the conclusion of the faulty clause.

1.3 Restricted semantics

To reconstruct attacks faster, [1] introduces a *restricted semantics*. It uses an existing derivation tree $\mathcal{D}$ to guide the exploration of the trace. For example, it only selects inputs or inputs that relate to facts in the derivation tree. It reduces the amount of possibilities and lessens the non-determinism. In some cases, it even becomes fully deterministic and no backtracking is necessary.

The main concept behind the restricted semantics is *justification*. The formalism used in [1] is slightly different from now. We explain the idea here using the modern formalism and adapting the notion.

First, we suppose we created a function called $fullocc(.)$ that takes a clause as input and outputs an occurrence term. This occurrence term represents the full occurrence of the conclusion, that is without stripping away the input message or `let` bindings. The sequence of name arguments is the same one as during the instrumentalization. In that, we say that an occurrence term is *complete*. We can build this function during the generation of clauses. Like in [3], we also define an ordering relation, written $\prec_{\mathcal{C}}$, on occurrences that depend on an (initial) configuration $\mathcal{C}$. We say that $o[\tilde{a}] \prec_{\mathcal{C}} o'[\tilde{b}]$ when $\tilde{a}$ is a prefix of $\tilde{b}$ and o' appears in the scope of o in $\mathcal{C}$. Note that we only compare complete occurrence terms, that is, when inputs were not stripped away from the name arguments. We may still compare incomplete occurrences by first converting them to full occurrences similarly as $fullocc(.)$. We may also "extend" the occurrence by writing $(o[\tilde{a}], a') \prec_{\mathcal{C}} O$. This is true if and only if $o[\tilde{a}, a'] \prec_{\mathcal{C}} O$. This notation is useful to check which message should be used as input.

Given $\mathcal{H}$ a set of hypothesis, a derivation tree $\mathcal{D}$, and a clause $\mathcal{H} \to C$, we say that a derivation tree contains a clause $\mathcal{H} \to C$, written $\mathcal{H} \to C \in \mathcal{D}$, when there exists a node in $\mathcal{D}$ labeled $\mathcal{H} \to C$, whose incoming edge is labeled C' and outgoing edges are labeled $F_1, \ldots, F_n$ and such that $\mathcal{H} \to C$ subsumes $F_1 \wedge \ldots \wedge F_n \to C'$. We say that the derivation $\mathcal{D}$ justifies the occurrence term O when there exists a clause $\mathcal{H} \to C \in \mathcal{D}$ such that $O \prec_{\mathcal{C}} fullocc(\mathcal{H} \to C)$.

This concept of justification is then used in premise of the semantics. The original rules can be found in Figure 5 (page 9) of [1]. We give a new version of the old rules in Figure A that adapt the instrumented semantics from [2]. New rules can be found in Section 2.2.2.

2 Improving current algorithm: private channel synchronization

2.1 The problem

The derivation tree outputted by the saturation might not reflect the private channel synchronization needed. Sometimes, we need to go past a $\mathtt{out}(d, x)$ for the rest of the attack. Because the channel d may be private, the attacker cannot simply receive the message and need to find a matching $\mathtt{in}(d, \_)$.

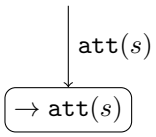
Example 1. *Consider the following process:*

$$P = \mathtt{in}^{o_1}(c, x); \mathtt{in}^{o_2}(d, y) \mid \mathtt{out}(d, s); \mathtt{out}(c, s)$$

The set of generated clause is the following, given that d is private and c is public:

$$\to \mathtt{mess}(d, s)$$
$$\to \mathtt{att}(s)$$

The derivation tree is the following:



but the synchronization needed on channel d is hidden.

2.2 Adding event-like stuff to the clauses

2.2.1 The "complete" but slow way

We want to make the synchronization dependence clear. To that end, we generate new clauses $\mathbb{C}'_{init}$ from the initial protocol according to the following rules:

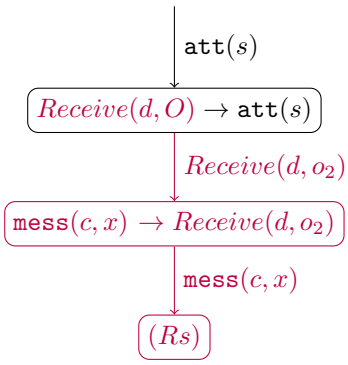
- $\texttt{out}(d, x)$ adds $Receive(d, x_o)$ to the hypothesis of the next clause with x_o a variable that will represent an occurrence term. This means that "to continue to the next point, you *need* to receive data on channel d "
- $\texttt{in}^O(d, x)$ creates a clause $H_1 \wedge \ldots \wedge H_n \rightarrow Receive(d, O)$, so we can match.

An explicit and formal transformation will be given later.

Example 2. *Continuing on Example 1, we highlight in red the new parts of the modified clauses:*

$$\begin{aligned} &\rightarrow \texttt{mess}(d, s) \\ Receive(d, O) &\rightarrow \texttt{att}(s) \\ &\rightarrow Receive(c, o_1) \\ \texttt{mess}(c, x) &\rightarrow Receive(d, o_2) \end{aligned}$$

with the derivation now being (we omit some attacker rules for clarity):



We see how the emphasis is put on private channel synchronization and is shown in the derivation.

To not saturate again from this initial set as this would be too slow, we modify the saturated set of clause. For a clause $\mathcal{R} \in \mathbb{C}_{sat}$, we have a derivation from the base set of clause $\mathbb{C}_{init}$. Each of the clause $\mathcal{R}_i \in \mathbb{C}_{init}$ from that derivation can be associated with a clause $\mathbb{C}'_i \in \mathbb{C}'_{init}$. This allows us to change the saturated set to a set of clauses $\mathbb{C}'$ close to the saturated set but with clauses from $\mathbb{C}'_{init}$. We then saturate $\mathbb{C}'$ to generate the subtrees necessary to derive $Receive(d, O)$.

While this would work, there is a high risk of non-termination by modifying *every* clause like that. As such, we take a more subtle approach.

2.2.2 The subtle approach

This new approach functions as successive rounds, until we either get stuck, find a successful trace or after a certain amount of rounds. After the first saturation, we get a set of saturated clauses $\mathbb{C}_{sat}$ and a clause $\mathcal{R}$ that falsifies the query. We create a set of clauses $\mathbb{C}_{init}^{Receive}$ from the initial process by adding only the clauses where the conclusion is $Receive(d, O)$. We do not add any $Receive(d, O)$ in hypothesis. This is done by applying the new clause generation only to the `in` part of the process.

We present here an overview of the algorithm. Details can be found later. We also give an example execution of the algorithm later in Section 2.2.3.

- ReconstructTrace: We first try to reconstruct a trace from that clause by exploring the restricted semantics. If we get stuck at some point due to private channel synchronization, we continue with ModifyClauseRecv.
- ModifyClauseRecv: We modify the clause $\mathcal{R}$ like in the previous section by changing the base clauses into clause with *Receive* and propagate that change in the hypothesis of $\mathcal{R}$. We also add blocking *Receive* with more information and a restriction. This creates a clause $\mathcal{R}'$ with *Receive* in hypothesis.
- SaturateInvalidate: We then do a saturation of $\mathbb{C}_{sat} \cup \mathcal{R}' \cup \mathbb{C}_{init}^{Receive}$ where $\mathcal{R}'$ is the request clause. The distinction of $\mathcal{R}'$ as a request clause is important to make sure the new clause is not subsumed. In practice, we use a special predicate (for example q-$\mathtt{att}(s)$) that only appears in conclusion of the query. The saturation gives a bunch of clauses that *might* invalidate the query. We keep only these clauses. If there is none, then we output Cannot be proved. We might point to the user where we got stuck.
- We then start a new round using the new clauses.

Thus, the algorithm is as follows:

Algorithm 1 Overview of the ReconstructAttack algorithm

function ReconstructAttack($\mathcal{R}$)
  $\tau \leftarrow$ ReconstructTrace($\mathcal{R}$)
  if τ successful **then return** τ
  else
    $\mathcal{R}' \leftarrow$ ModifyClauseRecv($\mathcal{R}$)
    $(\mathcal{R}'_1 \dots \mathcal{R}'_n) \leftarrow$ SaturateInvalidate($\mathbb{C}_{sat} \cup \mathcal{R}' \cup \mathbb{C}_{init}^{Receive}$)
    ReconstructAttack($\mathcal{R}_i$) for all i

Now for the details.

ModifyClauseRecv This function handles the conversion of the clauses to their variant with *Receive* in hypothesis. We define more clearly the predicates we will be using.

First, we will use $Receive(d, O)$ with d a channel and O an occurrence term. Intuitively, it indicates that we *need* to receive a message on channel d. The occurrence indicates which $\mathtt{in}(d, x)$ is used to receive. Formally, we add the following rule to the generation of clauses:

$$[\![\mathtt{in}^O(M, x); P]\!]\kappa\mathcal{H} = [\![P]\!]\kappa(\mathcal{H} \wedge \mathtt{mess}(M, x) \wedge \mathrm{pre}(O, x)) \cup \{\mathcal{H} \to Receive(M, O)\}$$

Now that we explained how we create *Receive* in conclusion, we need to clarify how we generate it in hypothesis of the clause. The idea is to add a $Receive(M, O)$ in hypothesis for each $\mathtt{out}^{O_{out}}(d, x)$ encountered Formally, we use the following clause generation:

$$\begin{aligned}[\![\mathtt{out}^{O}(M, N); P]\!]\kappa\mathcal{H} \\ = [\![P]\!]\kappa(\mathcal{H} \wedge Receive(M, x_o) \wedge b{-}IO(M, x_o, O)) \\ \cup \{\mathcal{H} \to \mathtt{mess}(M, N)\} \text{ with } x_o \text{ a fresh variable}\end{aligned}$$

Notice the predicate $b{-}IO$, called blocking I/O. We write $b{-}IO(d, O_{in}, O_{out})$ where O_{out} is the occurrence of an output and O_{in} is the occurrence of an input. It is used to carry throughout the saturation information regarding the matching done between inputs and outputs. We also use the following restriction in conjunction with $b{-}IO$:

$$b{-}IO(d, O^1_{in}, O^1_{out}) \wedge b{-}IO(d, O^2_{in}, O^2_{out}) \implies (O^1_{out} = O^2_{out} \iff O^1_{in} = O^2_{in})$$

This means we will not be able to use the same input or output twice.

This new generation of clauses is not used during the initial saturation, but only at specific moments. At the beginning of the algorith, we apply the `in` rule to create $\mathbb{C}^{Receive}_{init}$ while still omitting any *Receive* in hypothesis. The `out` generation is used when we are stuck as follows: we map each clause of the derivation tree (thus that originates from the base generation) to a clause generated with the `out` rule enabled, consequently adding *Receive* in hypothesis. We then apply the resolutions made in the derivation to get a new clause with more hypothesis. This is the returned clause.

ReconstructTrace We use the same approach with the restricted semantics. We propose a few adjustments to make it compatible with our strategy and more up to date with current Proverif.

- Regarding *Receive*, we create a rule similar to (I-Comm) called (Res-Comm) that works with *Receive*. A trace is allowed to take a step by matching an input $\mathtt{in}^{O_{in}}(d, x)$ and an output $\mathtt{out}^{O_{out}}(d, y)$ when they match a $b{-}IO(d, O_{in}, O_{out})$ somewhere in the rule. Because this is blocking, we only need to check the hypothesis of $\mathcal{R}$. Formally, this is the rule:

$$\text{Res-Comm1}\ \frac{b{-}IO(M, O_{in}, O_{out}) \in \mathcal{R}}{\mathcal{P} \cup \{(\mathtt{in}^{O_{in}}(M, x); P), (\mathtt{out}^{O_{out}}(M, N); Q)\} \to_r \mathcal{P} \cup \{P\{N/x\}, Q\}}$$

- We also keep the previous rule that let us step through an IO through a message predicate:

$$\text{Res-Comm2}\ \frac{\mathtt{mess}(M, N) \wedge \mathrm{pre}(O_{in}, N) \in \mathcal{R}}{\mathcal{P} \cup \{(\mathtt{in}^{O_{in}}(M, x); P), (\mathtt{out}^{O_{out}}(M, N); Q)\} \to_r \mathcal{P} \cup \{P\{N/x\}, Q\}}$$

- We extend the Let rules, notably useful for tables, with justification.
  - For (I-Let1), we can create (Res-Let1):

$$\text{Res-Let1}\ \frac{\begin{array}{c}\{y_1, \ldots, y_n\} = \{x_i \mid K = K' \wedge \mathtt{itable}(x_i, tbl)\} \quad M = \{x_1, \ldots, x_n\} \\ (\mathcal{C}, \sigma) \models K \qquad \sigma \text{ ground} \qquad dom(\sigma) = \{x_1, \ldots, x_n\} \\ \mathcal{D} \text{ justifies } (O, M\sigma)\end{array}}{\mathcal{P} \cup \{\!\{\mathtt{let}^{O}\ x_1, \ldots, x_n\ \mathtt{s.t.}\ K\ \mathtt{in}\ P\ \mathtt{else}\ Q\}\!\} \to_r \mathcal{P} \cup \{\!\{P\sigma\}\!\}}$$

– For (I-Let2), we apply the same idea, but we check that the tree does not justify this occurrence, meaning it appears nowhere in the tree, thus no pre$(O, .)$ was used.

$$\text{Res-Let2}\ \frac{\forall \sigma, (\mathcal{C}, \sigma) \not\models K \quad M = \{x_1, \ldots, x_n\} \quad \forall \sigma, \mathcal{D} \text{ does not justify } (O, M\sigma)}{\mathcal{P} \cup \{\!\{\texttt{let}^O\ x_1, \ldots, x_n\ \texttt{s.t.}\ K\ \texttt{in}\ P\ \texttt{else}\ Q\}\!\} \to_r \mathcal{P} \cup \{\!\{Q\}\!\}}$$

– For (I-Insert), we use a similar idea.

$$\text{Res-Insert}\ \frac{\mathcal{D} \text{ justifies } O}{\mathcal{P} \cup \{\!\{\texttt{insert}^O\ tbl; P\}\!\}, \mathcal{T} \to_r \mathcal{P} \cup \{\!\{P\}\!\}, \mathcal{T} \cup \{(O, tbl)\}}$$

Note that here, we may need to check for suffixes of O at the predicate $\texttt{itable}(O, tbl)$ appears in $\mathcal{D}$ only if it is used for a `get` later. If we only need to step through this `insert` o access P, the predicate does not appear in $\mathcal{D}$.

- Regarding events, we want to step through an event only if it appears in the derivation. However, Proverif might remove events that *do not appear in the query*. Thus, we are not able to specify directly in the semantics, which is thus far query-agnostic, whether we allow an event to be stepped through. Consequently, we allow all event instructions to be taken and defer the responsibility of correctly stepping to the *step selection function*. More information in Section 2.2.4

$$\text{Res-Event}\ \frac{}{\mathcal{P} \cup \{(\texttt{event}(ev, O); P)\} \to_r \mathcal{P} \cup \{P\}}$$

- Regarding phases, we need to be precise. Intuitively, we want to allow stepping from phase κ to phase $\kappa' > \kappa$ when there is no work left in phase κ. As a second condition, we also need to make sure that all the processes that *should* go to phase κ' are ready to transition. We need to find good ways to check for those two conditions.

  First, we can check if there is important work left in this phase by checking in the derivation tree if there exists some fact that we have not yet encountered in the trace that depends on the current phase. Thus, we suppose we have a function $\texttt{phaseof}(.)$ such that $\texttt{phaseof}(F)$ is the phase of the fact. We can create this function when generating clauses. While exploring the trace, we also keep track of the facts that hold. We express that using trace satisfaction : $\tau \models F$. To summarize, we can express that there should be no more work left to do at phase κ by writing: $\forall F \in \mathcal{D},\ \texttt{phaseof}(F) = \kappa \implies \tau \models F$.

  Now we need to make sure that all processes that should not be dropped are, in fact, not dropped by the phase change.

  Definition 4 (Safe drop)**.** *We say that a process P* can be safely dropped *when no occurrences of P appear in $\mathcal{D}$. Formally:*

$$\forall O \in P,\ \mathcal{D} \textit{ does not justify } O$$

  Thus, we need to check that all processes that are about to be dropped can be safely dropped. We summarize everything in the following rule:

$$\text{Res-Phase}\ \frac{\begin{array}{c}\text{all processes of } \mathcal{P} \text{ do not start with } \texttt{phase}\ \kappa', \kappa' > \kappa \\ \text{all processes of } \mathcal{P}_{>\kappa+1} \text{ start with } \texttt{phase}\ \kappa', \kappa' > \kappa + 1 \\ \forall F \in \mathcal{D},\ \texttt{phaseof}(F) = \kappa \implies \tau \models F \\ \forall P \in \mathcal{P},\ P \text{ can be safely dropped}\end{array}}{\kappa, \mathcal{P} \cup \mathcal{P}_{>\kappa+1} \cup \{\!\{\texttt{phase}\ \kappa + 1; P_i\}\!\}_{i=1}^k \to_r \kappa + 1, \mathcal{P}_{>\kappa+1} \cup \{\!\{P_i\}\!\}_{i=1}^k}$$

2.2.3 Running the algorithm on an example

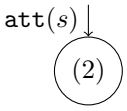
We consider the following process:

$$\begin{aligned} P = \ & \mathtt{out}^{o_1}(d', s);\ \mathtt{out}^{o_2}(c, s) \\ & \mid \mathtt{out}^{o_3}(d, s);\ \mathtt{in}^{o_4}(d', x) \\ & \mid \mathtt{in}^{o_5}(d, x) \end{aligned}$$

The generated clauses after the initial generation are the following:

$$\rightarrow \mathtt{mess}(d', s) \qquad (1)$$
$$\rightarrow \mathtt{att}(s) \qquad (2)$$
$$\rightarrow \mathtt{mess}(d, s) \qquad (3)$$

Note that this set is, in fact, already saturated. Proverif finds a derivation of $\mathtt{att}(s)$ that is composed of a single rule.



We also consider the set $\mathbb{C}_{init}^{Receive}$:

$$\rightarrow Receive(d, o_5) \qquad (4)$$
$$\rightarrow Receive(d', o_4) \qquad (5)$$

First Round We start our algorithm with ReconstructAttack(2). However, there is no trace that can be found in the restricted semantics as it cannot step through $\mathtt{in}^{o_5}(d, x)$. Thus, we run ModifyClauseRecv(2). We obtained the following new clause:

$$b{-}IO(d', x_o, o_1) \wedge Receive(d', x_o) \rightarrow \mathtt{att}(s) \qquad (2')$$

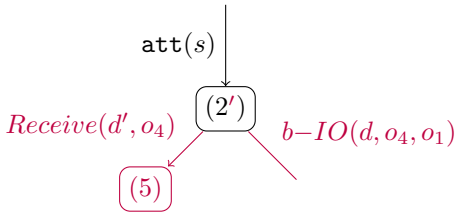
which gives the following set to saturate in SaturateInvalidate:

$$\rightarrow \mathtt{mess}(d', s) \qquad (1)$$
$$b{-}IO(d', x_o, o_1) \wedge Receive(d', x_o) \rightarrow \mathtt{att}(s) \qquad (2')$$
$$\rightarrow \mathtt{mess}(d, s) \qquad (3)$$
$$\rightarrow Receive(d, o_5) \qquad (4)$$
$$\rightarrow Receive(d', o_4) \qquad (5)$$

After saturation, we get the following clause that falsify the query:

$$b{-}IO(d', o_4, o_1) \rightarrow \mathtt{att}(s)$$

Note that the history of this clause is made of (2) and (5). Its derivation tree is the following:



Second Round We now start a new round of our algorithm. We still do not have any successful trace for the same reason as above, so we modify the false clause again. Remember that modifying a clause means that we modify the base clauses in the history and then propagate the results. The history here is made of (2) and (5) so we change those clause to (2′) and (5′):

$$b{-}IO(d', x_o, o_1) \wedge Receive(d', x_o) \rightarrow \texttt{att}(s) \qquad (2')$$
$$b{-}IO(d, y_o, o_3) \wedge Receive(d, y_o) \rightarrow Receive(d', o_4) \qquad (5')$$

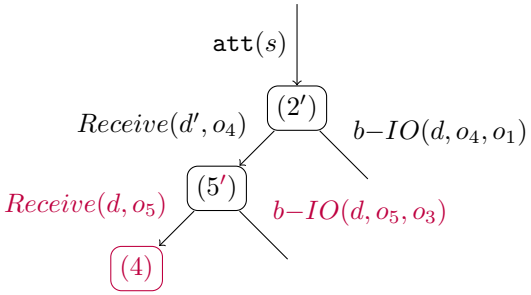
After propagating the changes by redoing the resolutions, we get the following clause $\mathcal{R}'$:

$$b{-}IO(d', o_4, o_1) \wedge b{-}IO(d, y_o, o_3) \wedge Receive(d, y_o) \rightarrow \texttt{att}(s)$$

We saturate and get the following false clause, made of (2′), (5′) and (4):

$$b{-}IO(d', o_4, o_1) \wedge b{-}IO(d, o_5, o_3) \rightarrow \texttt{att}(s)$$

whose derivation is:



Third Round We start our next round. This time, we are able to step through $\texttt{in}^{o_5}(d, x)$ thanks to $b{-}IO(d, o_5, o_3)$ being in the derivation. We find the rest of the trace as well, finishing the reconstruction.

2.2.4 Step Selection Function

While the restricted semantics prohibits moving in the trace however we wish, it does not completely remove non-determinism. We want to step in the trace in a specific order. Let us consider a few examples to illustrate this problem.

Example 3. *Consider the following protocol:*

$$\texttt{insert}\ tbl(0)$$
$$\mid \texttt{get}\ tbl(x)\ \texttt{suchthat}\ x = 0\ \texttt{in}\ \texttt{out}(c, 0)\ \texttt{else}\ \texttt{out}(c, s)$$

Here, to leak s, we need to take the `else` *branch of the* `get`*. If at some point we step through the* `insert`*, then we will not be able to get a trace. We need to prioritize the* `get` *here.*

Example 4. *Consider the following protocol:*

$$\texttt{insert}\ tbl(0)$$
$$\mid \texttt{out}(c, s)$$

with the query $\texttt{att}(s) \implies table(0)$.

Here, we need to be careful not to step through the `insert` $tbl(0)$*, so we get a trace where* $\texttt{att}(s)$ *holds but not* $table(0)$*. This example shows that the query itself changes how we should step while reconstructing the trace.*

We consider a *step selection* function written `ssel`. Intuitively, this function takes the query and the current configuration as parameters and return the step we should take in the current trace reconstruction. We may define various step selection functions with different heuristics.

More formally:

Definition 5 (Step Selection function)**.** *A step selection is a function that takes as input a configuration, a query and a derivation tree and outputs a process. It verifies the following property:* $\texttt{ssel}(\rho, \mathcal{C}, \mathcal{D}) \in \texttt{next\_procs}(\mathcal{C}) \cup \{\bot\}$ *where* $\texttt{next\_procs}(\mathcal{C})$ *is the set of potential processes that we might step through* using the restricted semantics*. It can be seen as the "head" of all the processes in the configuration* $\mathcal{C}$*. It may return* $\bot$ *to indicate that no steps should be taken and reconstruction is stuck.*

Simple Step Selection function We will now describe one simple step selection function, called `simp_ssel`. The idea is to prioritize steps in the premises of the query and delay the steps through the conclusion of the query. The steps that do not affect the end results are also prioritized.

Internally, this function associates weights to each values of $\texttt{next\_procs}(\mathcal{C})$ and outputs the highest weight. If multiple values share the highest weight, one is chosen. We do not specify here which one might get chosen. Phases have a neutral weight.

3 Matching replication occurrences

3.1 Introducing the idea

When instrumenting a process, the `new` are replaced by fresh names containing occurrences. The variables inside the occurrences express the inputs and replications that happened previously in the trace. We write $o(a_1, \ldots, a_n)$ an *occurrence term*, where o is an *occurrence* and $(a_1, \ldots, a_n)$ are called *name arguments*. When the name arguments are still variables, we might call these variables *occurrence variables*

We are particularly interested in replications. When stepping through a replication, the process semantics forces the use of a fresh identifier unique to that replication. Thus, if two names share a replication index, then they come from the same execution trace. The interesting point here is that it allows us to unify the occurrence terms, and more precisely the variables related to name arguments, that appear prior to this replication. Formally, given two names $a[x_1, \ldots, x_n, \lambda, x'_1, \ldots, x'_n]$ and $b[y_1, \ldots, y_n, \lambda, y'_1, \ldots, y'_n]$, if both of them are present in the same clause and that clause does come from a real trace, then we have $\forall i, x_i = y_i$.

We can actually go one step further. In some cases, variables after the shared session index must be equal as well. This is the case for inputs that directly follow the shared replication. Until the next replication, all occurrences will be equal. Note that it is slightly more complicated as we may have both names come from different parts of the process, for example from two parallel processes.

3.2 Preliminaries: formalizing the idea

An occurrence variable x_i is associated with a (unique) occurrence o due to how the transformation to an instrumented process works. We write that occurrence $\texttt{occof}(x_i)$.

We first introduce some notions related to sequences of name arguments and replications:

Definition 6 (Replication Sequence)**.** *Consider a sequence of name arguments $\tilde{a} = [a_1, \ldots, a_n]$ similar to the transformation to instrumented process. We write $\tilde{a}_{|\lambda}$ the longest subsequence of $\tilde{a}$ such that:*

$$\forall a \in \tilde{a}_{|\lambda}, \texttt{occof}(a) \text{ comes from a replication}$$

We call $\tilde{a}_{|\lambda}$ the replication sequence of $\tilde{a}$. Any prefix of $\tilde{a}_{|\lambda}$ is called a replication prefix of $\tilde{a}$.

Definition 7 (Longest Replication Prefix)**.** *Consider two sequences of name arguments $\tilde{a} = [a_1, \ldots, a_n]$ and $\tilde{b} = [b_1, \ldots, b_m]$. We say that $\tilde{a}$ and $\tilde{b}$ share a replication prefix when there exists $\tilde{c}$ such that $\tilde{c}$ is a replication prefix of $\tilde{a}$ and $\tilde{b}$.*

We consider the longest replication prefix between the two sequences. In this context, we write $id_{\tilde{a}}$ the index such that $\tilde{a}_{id_{\tilde{a}}} = \tilde{c}_{-1}$, that is, $id_{\tilde{a}}$ is the index of the last shared replication. We write the same for $\tilde{b}$.

Definition 8 (Input variables of a replication sequence)**.** *Consider a sequence of name arguments $\tilde{a} = [a_1, \ldots, a_n]$. Given an index i, we define the set of input variables of a replication sequence, written $IV(\tilde{a}, i)$, as the following:*

$$IV(\tilde{a}, i) = \{a_k \mid k < i \text{ or } i < k < j, \ \texttt{occof}(a_k) \text{ comes from an input or a } \texttt{let}\}$$

where j is the index of the first replication after i, or n if there is none. This set represents all the input variables that are under replications from the sequence.

We mostly use this notion in conjunction with the longest replication prefix. We extend these notions to sequences of name arguments or variables. Furthermore, we naturally extend these to deal with names instead of sequence of name arguments.

3.3 Main theorem

Theorem 1. *Consider a process P, a clause $\mathcal{R}$ and two names $a[x_1, \ldots, x_n]$ and $b[y_1, \ldots, y_m]$ that are subterms of $\mathcal{R}$. Consider the longest replication prefix between a and b, written $\tilde{c}$, and the associated indices id_a and id_b.*

We have:

$$\forall a' \in IV(a, id_a), \forall b' \in IV(b, id_b), \ \texttt{occof}(a') = \texttt{occof}(b') \implies a' = b'$$

Proof. The main idea of this proof is the tree-like structure of the instrumentalization and the process semantics. Consider $a' \in IV(a, id_a)$ and $b' \in IV(b, id_b)$ and suppose that $\texttt{occof}(a') = \texttt{occof}(b')$. We write that occurrence o.

Consider a trace that steps through the instantation (that is, `in` or `let`) related to a' and b'. We first show that the occurrence term of these two statements is the same. We know that the occurrence is the same by hypothesis, written o. Now we need to show that both sequences of name arguments are equal. In the transformation to instrumented process, we only label the occurrences with the replication indices. Because $a' \in IV(a, id_a)$ and $b' \in IV(b, id_b)$, the replications that may appear in the occurrence terms are prefixes of $\tilde{c}$. In addition, the sequences are the same size by definition of the transformation to instrumented process. Thus, we conclude that both statements are labeled with the same occurrence term.

We now show that there cannot be two different `in` or `let` labeled with the same occurrence term. This is a direct consequence of the process semantics. The sequence of replication is the same for both constructs thus the trace must step through the replications the same way for both. However, the semantic does not allow stepping through an occurrence term multiple time for a given replication sequence. Thus, both instantiations are actually the same.

Both a' and b' come from the same `in` or `let`. This ensures that the value of the name argument will be the same. Thus, $a' = b'$. □

3.4 Going further than clause: derivations

While this simplification is useful and can be used during the saturation, we can take it one step further for the reconstruction of attacks. We can actually apply this results to a whole derivation instead of a single clause, assuming the derivation represents a real trace, which is what we want from our reconstruction. By considering the derivation as a tree, we can then consider all the names that are present inside the tree. This is slightly stronger as we can have names that might get simplified with resolutions that will then appear in the complete derivation.

Theorem 2. *Consider a process P, a derivation $\mathcal{D}$ and two names $a[x_1, \ldots, x_n]$ and $b[y_1, \ldots, y_m]$ that appear in the derivation. Consider the longest replication prefix between a and b, written $\tilde{c}$, and the associated indices id_a and id_b.*

If $\mathcal{D}$ corresponds to a real trace of P, we have:

$$\forall a' \in IV(a, id_a), \forall b' \in IV(b, id_b),\ \mathtt{occof}(a') = \mathtt{occof}(b') \implies a' = b'$$

Proof. The proof here is similar to Theorem 1. We first consider a trace that corresponds to the derivation, then show that the inputs related to the name arguments are identical. □

4 Smarter `let` constructs

We introduce another optimization related to name arguments. This time, we focus on `let` constructs. The key observation is that the `then` and `else` branches are exclusive: a process cannot step through the `then` block and the `else` block for a given replication sequence.

Before formally stating our theorem, we introduce some concepts and writing shorthands. Given a process P and an occurrence o, we write $\mathtt{then}(o)$ (resp. $\mathtt{else}(o)$) the process in the `then` (resp. `else`) branch.

Definition 9 (Occurrence restriction)**.** *Given a sequence of name arguments $\tilde{b}$ and a sequence of occurrences $\tilde{o}$, we write $\tilde{a}_{|\tilde{o}}$ the longest subsequence of $\tilde{a}$ such that:*

$$a \in \tilde{a}_{|\tilde{o}} \iff \mathtt{occof}(a) \in \tilde{o}$$

We say that $\tilde{a}_{|\tilde{o}}$ is the restriction of $\tilde{a}$ to $\tilde{o}$. We extend this notion to use a sequence of name arguments on the restriction side, by mapping $\mathtt{occof}(.)$ on the sequence, and then we extend to occurrence terms. We also extend this to a restriction of an occurrence term.

We can now formally state the theorem:

Theorem 3. *Consider a process P and a clause $\mathcal{R}$. Let O be the occurrence term of a `let` binding. Consider two names $a[x_1, \ldots, x_n]$ and $b[y_1, \ldots, y_m]$ that are subterms of $\mathcal{R}$. Suppose that $a[x_1, \ldots, x_n] \in \mathtt{then}(O)$ and $b[y_1, \ldots, y_m] \in \mathtt{else}(O)$. Thus, we have:*

$$(x_1, \ldots, x_n)_{|O} \neq (y_1, \ldots, y_m)_{|O}$$

Note that this theorem only creates disequalities between session identifiers as they are the only name arguments in the occurrence term of a `let`. This is, in fact, equivalent to a theorem considering "complete" occurrences (similarly as $fullocc(.)$) thanks to Theorem 1.

Proof. This proof uses similar arguments as the proof of Theorem 1. Suppose that $(x_1, \ldots, x_n)_{|O} = (x_1, \ldots, x_m)_{|O}$. As both names are part of the `then` or `else`, their sequence of name arguments *must* be longer than O. Thus, we have a common replication prefix between both names, written

$\tilde{c}$, that is longer than O. When considering a trace, having a shared replication prefix means that it steps through the replication the same way. The replication prefix is longer than O, thus the trace can step all the way until the `let` binding. However, the semantics does not allow us to step through a `then` and a `else` branch with the same replication sequence. Thus, we cannot have the same replication sequence and we get an impossibility, finishing the proof. □

Note that there is a slightly different version of that theorem that deals with occurrence terms instead of names:

Theorem 4. *Consider a process P. Let O be the occurrence term of a* `let` *binding. Consider two fully instantiated occurrence terms $O_1 = o_1[x_1, \ldots, x_n] \in \texttt{then}(O)$ and $O_2 = o_2[y_1, \ldots, y_m] \in \texttt{else}(O)$. Thus, we have:*

$$(x_1, \ldots, x_n)_{|O} \neq (y_1, \ldots, y_m)_{|O}$$

The proof here is omitted as it is similar to the previous theorem.

We claim that theses theorems are equivalent. The idea of the equivalence proof is to slightly alter the process by adding names (resp. events) next to the occurrence (resp. name). In that case, the name arguments sequences are identical.

Conclusion and Future Works

We proposed a proposed a new reconstruction algorithm meant to handle private channels as well as the theory behind it. We also stared implementing said algorithm in the tool and it is showing some success.

We unfortunately did not have time to implement the last two optimizations regarding occurrences.

References

[1] Xavier Allamigeon and Bruno Blanchet. "Reconstruction of Attacks against Cryptographic Protocols". In: *18th IEEE Computer Security Foundations Workshop (CSFW-18)*. Aix-en-Provence, France: IEEE Computer Society, June 2005, pp. 140–154.

[2] Bruno Blanchet et al. *The (almost) complete theory of ProVerif.*

[3] Vincent Cheval and Itsaka Rakotonirina. "Indistinguishability Beyond Diff-Equivalence in ProVerif". In: *2023 IEEE 36th Computer Security Foundations Symposium (CSF)*. 2023, pp. 184–199. DOI: 10.1109/CSF57540.2023.00036.

A Restricted semantics

$$\text{Res-In}\ \frac{D \rhd M \in \mathcal{A} \quad E \rhd N \in \mathcal{A} \quad \mathcal{D} \text{ justifies } (O, N)}{\mathcal{P} \cup \{\!\{\mathtt{in}^O(M, x); P\}\!\}, \mathcal{A} \to_r \mathcal{P} \cup \{\!\{P\{N/x\}\}\!\}, \mathcal{A}}$$

$$\text{Res-Out}\ \frac{D \rhd M \in \mathcal{A} \quad x \text{ a fresh variable} \quad \mathcal{D} \text{ justifies } O}{\mathcal{P} \cup \{\!\{\mathtt{out}^O(M, N); P\}\!\}, \mathcal{A} \to_r \mathcal{P} \cup \{\!\{P\}\!\}, \mathcal{A} \cup \{x \rhd N\}}$$

$$\text{Res-Repl}\ \frac{O\sigma \notin \Lambda \quad dom(\sigma) = \tilde{a} \quad img(\sigma) \subseteq \mathcal{N}_s \quad \mathcal{D} \text{ justifies } O\sigma}{\mathcal{P} \cup \{\!\{\{\!\{!^O_{\tilde{a}} P\}\!\} \cup \mathcal{M}\}\!\}, \Lambda \to_r \mathcal{P} \cup \{\!\{P\sigma, \{\!\{!^O_{\tilde{a}} P\}\!\} \cup \mathcal{M}\}\!\}, \Lambda \cup \{O\sigma\}}$$

where $\mathcal{N}_s$ is the set of session identifiers

$$\text{Res-App}\ \frac{\begin{array}{c} D_i \rhd U_i \text{ for } i \in \{1, \ldots, n\} \quad f \in \mathcal{F}_{c,pub} \cup \mathcal{F}_{d,pub} \\ f(U_1, \ldots, U_n) \Downarrow U \qquad D = f(D_1, \ldots, D_n) \\ \mathtt{att}(U_1) \ldots \mathtt{att}(U_n) \in \mathcal{D} \end{array}}{\mathcal{P}, \mathcal{A} \to_r \mathcal{P}, \mathcal{A} \cup \{D \rhd U\}}$$

$$\text{Res-Gen}\ \frac{\begin{array}{c} M = f_{\mathcal{N}_{pub}}(\lambda) \quad \forall D \rhd U, D \neq M \\ M \in \mathcal{D} \end{array}}{\mathcal{P}, \mathcal{A} \to_r \mathcal{P}, \mathcal{A} \cup \{M \rhd M\}}$$

where $f_{\mathcal{N}_{pub}}$ is a private unary function used to generate fresh names and λ a session identifier

Figure 1: Adapted rules of the Restricted semantics